Course Notes on Database Systems

Rahul Simha

Department of Computer Science
The George Washington University

Chapter 1

Introduction

Course Notes on Database Systems

1.1 CS 321: Database Systems

- **Prerequisites**. CS 313, CS 304, programming in C (you need to know: essentials of algorithm analysis, how to code tree-type data structures). No prior knowledge of databases needed.
- **Course description**. In this course, students will learn what a database is, and why data management requirements in the real world require more than just data structures. We will cover the standard relational model in great detail: the relational model, relational algebra, normal forms, physical storage and access, transaction processing, concurrency, recovery, distributed databases. Time permitting, some advanced topics from among the following will be covered: parallel databases, string processing, compression, spatial data processing. Students will implement assignments in the Oracle database as well as write code to build parts of a database.
- **Course materials**.
  - **Textbook**: No textbook.
  - **Notes/Slides**: Copies of slides used in class to be purchased from Computer Science office.
  - Stuff on reserve in Swem:
    - R.Elmasri and S.Navathe. Fundamentals of Database Systems. *Benjamin-Cummings*, 1994.
    - P.O'Neill. Database: principles, programming, performance. *Morgan-Kaufman*, 1994.
    - Various Oracle books.
  - On-line Oracle documentation.
  - Other items on reserve that may be of interest:

∗ R.Laurini and D.Thompson. Fundamentals of Spatial Information Systems. *Academic Press*, 1992.

∗ F.Preparata and M.I.Shamos. Computational geometry: an introduction. *Springer-Verlag*, 1985.

∗ H.Samet. Applications of Spatial Data Structures. *Addison-Wesley*, 1990.

1.2 What is a database?

- **Def**: *A database is a collection of interrelated data.*

  Q: for what purpose?

- **Def**: *A Database Management System (DBMS) is a collection of interrelated data and a set of programs to create, access and manipulate the data*

- What is an example of a database? Consider:
  - Linked list containing 100 integers.
    - 50 lines of C
    - operations: add, delete nodes

  ⇒ is this a DBMS?

  - If so, what aspect of the linked list program is a DBMS? ⇒ object code, code-while-executing?

  NO! It is not a DBMS!

- There is more to a DBMS:
  - *Large data size.* Modern dbases have gobs of data. e.g. IRS data
    - Approx. 100 million taxpayers, 2000 bytes per person

      ⇒ $100 \times 10^6 \times 2000 = 2 \times 10^{11}$ bytes
  - *Persistence of data.* Data still exists after programs complete execution.

    ⇒ permanent storage on disk
  - *Variety of interrelated data.*

- Is a *file system* a DBMS?
  NOTE: a file system has
  - Large storage capability
  - Persistence of data (non-volatile storage)
  - Variety of interrelated data
- Consider a typical application:
  - University registrar keeps information about students, faculty, courses offered, classrooms etc. e.g,
    - ∗ For each student: id, name, major, grades
    - ∗ For each course: courseid, course name, credits, dept
    - ∗ For each section: courseid, course name, instructor
    - ∗ For each faculty: name, dept, courses, office, telephone
  - What does the registrar do with the data?
    - ∗ Answer queries: "What grade did Smith get in CS526?", "Get all courses taught by Jones"
    - ∗ Insert new info: "Smith got a B+ in CS534", "Classroom Terc 104 not available from 9-10.50"
    - ∗ Delete some info: "Remove Smith and related data"
- Can a file system suffice for the above application? e.g.
  - Store data in various text files
  - Write small C programs to scan data for queries, make insertions and deletions
  - Utilities for sorting, pretty-printing, GUI

  ⇒ If this is what a DBMS is, why a whole course on dbases?
  ⇒ Why are commercial dbases expensive? (over $1,000,000 for full-size Oracle).
  ⇒ Is there more to a DBMS?

1.3 Of Course There's More to a DBMS

- **Control of data redundancy and inconsistency**
  - Unnecessary duplication of data can cause problems:

    ⇒ *Space wastage*: putting a student's postal address with every occurence of the student's name.

    ⇒ *update cost*: if an address changes, we have to find every occurence of old address and update it

COURSE	NAME	ADDRESS
	⋮	
321	Smith, J	123 Rappahannock St, Tappahannack, VA 29127
	⋮	
304	Smith, J	123 Rappahannock St, Tappahannack, VA 29127
	⋮	
423	Smith, J	123 Rappahannock St, Tappahannack, VA 29127
	⋮	

  - Some duplication is necessary
    - difficult to eliminate completely

      ⇒ design becomes convoluted e.g., if we insist *course numbers* appear only once, pointers will be needed to course numbers

      ⇒ deletion is a problem (garbage collection needed)
  - Data must be consistent at all times

    ⇒ Changes must occur completely or not at all

    (What happens if an update program crashes before completion?)

- **Efficient data access**.

- To answer a query about a particular student, should one scan the whole dbase?
- For large amounts of data, searching can be time-consuming.
- NOTE: Access efficiency is evaluated differently in DBMS
  ⇒ minimize disk accesses rather than CPU time

- **Concurrent access**.
  - Registrar's office has many employees that need to access the dbase simultaneously
    ⇒ they should not have to take turns at a single terminal
    ⇒ concurrent access to data is desired
  - Need to be careful:
    - User 1: delete "Jones took CS 313"
      User 2: "List Jones' courses"

    ⇒ need to coordinate actions of different users carefully.

- **Security**.
  - Not every user should be able access all the data
    (students able to add/delete grades?)
  - Control of access to data is needed without having to duplicate data.

- **Integrity constraints**.
  - Grades need to be in the set $\{F, ..., A+\}$.
  - GPA's cannot be negative.
  - A student cannot take a nonexistent course.

  ⇒ A DBMS should enforce broad integrity constraints.

- **Backup and recovery**.
  - Data must be systematically backed up.
  - Recovery in system crashes.
  - Inserts and updates cannot occur partially
    ⇒ either complete it or don't do it at all
- **Metadata**.
  - Data about the data (data types, interrelationships, integrity constraints etc).
  - directory and catalog information.
- **Data and program independence**.
  - Data should be stored in a program-independent fashion
    ⇒ changes in program should not affect access to data
    ⇒ data re-entry should not be needed
- **User and programmer interface**.
  - query language
  - GUI
  - reduced application development time
    (C function library to manipulate data)
  - features that allow data to be re-organized for efficient access
- Let's return to the question: is a file system a DBMS?
  ⇒ clearly, no.

1.4 What is a Data Model?

- *Data Model*: An agreed-upon method for abstractly describing the logical organization of data.

  NOTE: the term data model is often incorrectly used in the literature.

- An analogy: architectural blueprints
  - Blueprints are standardized: any builder can work on a plan drawn by any architect.
    (conventional symbols for windows, doors, plumbing etc)
  - Blueprint ‘standards’ may differ across countries.

- A Data Model is like a blueprint ‘standard’:
  - An agreed-upon way to describe how a dbase is to be organized
  - It permits actual implementation to be independent of this description.
    - ∗ Architectural analogy: blueprint does not specify building materials (can use cement or brick exterior)
    - ∗ Data model: a data model can be implemented in various ways

- *Example*: sets
  - Sets are described using notation like:

$$\begin{aligned} A &\leftarrow \{17, 234, 88\} \\ B &\leftarrow \{45, 88, 129, 564\} \end{aligned}$$

  - Operations are defined on sets: union, intersection etc
  - However, sets can be implemented (in C) in many ways:
    1. arrays
    2. linked lists
    3. various algorithms for union, intersection etc
- In dbases, a Data Model describes the *logical* organization of data, along with operations that manipulate the data.
- In this course, we will cover: the *Relational Model*.

1.5 Various people associated with DBMS's

- **Database Administrator**
  - usually heads a team of dbase professionals
  - involved in purchase and installation of a dbase
  - decides data organization, access priveleges, integrity constraints, customization
  - oversees users, applications programmers
- **Database application programmer**
  - develops applications for end users
  - writes code in a query language (e.g. SQL) and in C/C++
  - writes code for GUI-access or web-access to dbase
- **End user**
  - uses interface for queries (e.g. a bank teller)
  - rarely is able to program in SQL
- **DBMS system programmer**
  - works for a dbase vendor (e.g. Oracle, Sybase etc)
  - Various levels: system (low-level), application (GUI's etc), language (C++/Java), library/tool developer (specialized applications, e.g., GIS)
- **Dbase researcher**
  - works on research problems associated with dbases
    ⇒ dbase design, query processing algorithms, system issues, performance modeling

1.6 Different Ways of Using a Database

- *Using a query language interactively*, e.q. SQL in Postgres

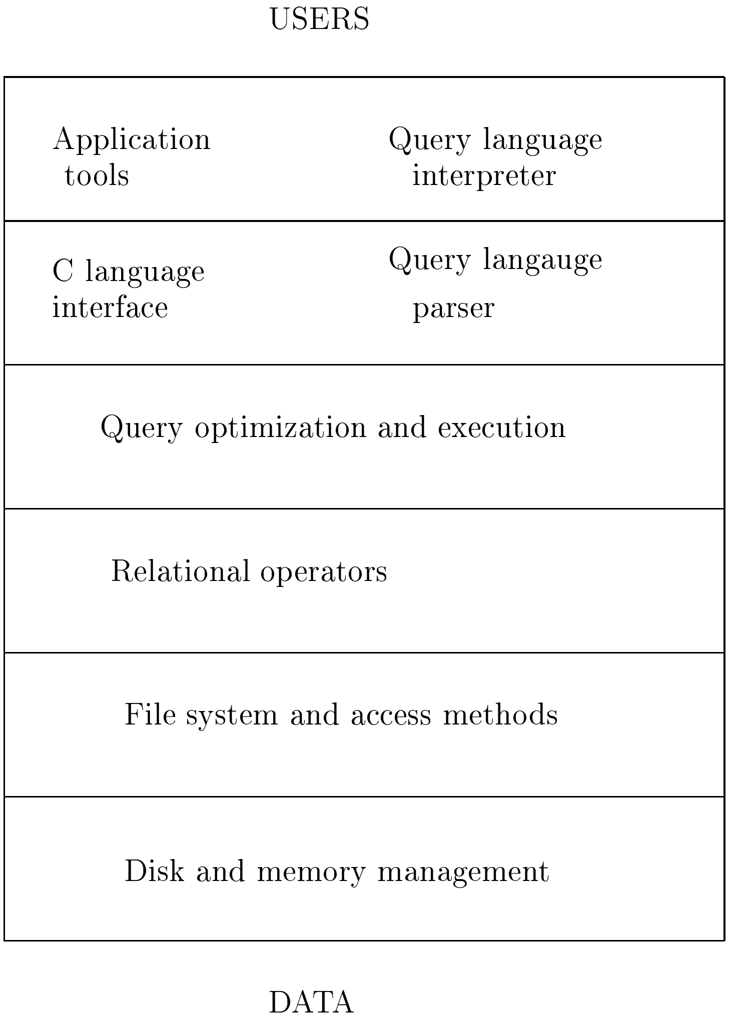
```
% psql test1
psql> \d
  Database    = test1
  +------------------+-----------------+----------+
  |  Owner           |  Relation       |   Type   |
  +------------------+-----------------+----------+
  | simha            | acc_type        | table    |
  | simha            | accounts        | table    |
  | simha            | branch          | table    |
  | simha            | managers        | table    |
  | simha            | transactions    | table    |
  +------------------+-----------------+----------+

pqsl> select * from branch;
   branch        city
   ------------- -----------
   Friendship    Washington
   Downtown      Washington
   Downtown      New York
   Foggy Bottom  Washington
   Adams Morgan  Washington
   Chevy Chase   Washington
   South East    Washington

psql> \q
```

- *Writing query language code separately*, e.g. SQL in Postgres
  - Create a file called `samplequery`:
    `select * from branch;`
  - Execute the query in Postgres:
    `% psql < samplequery`
- *Writing C code using a C-interface (API)*, e.g. in Postgres:
  - Write a C program in a file called `sample.c`

    ```
    #include "libpq-fe.h"
    main () {
      conn = PQsetdb (pghost, pgport, pgoptions, pgtty, dbName);
      str = PQdb (conn);
      res = PQexec (conn, "select * from branch");
      PQprintTuples (res, stdout, 1, 0, 0);
    }
    ```

  - Compile and execute `sample.c`
- *Writing internal code*
  - Source code required
  - Most often written in C/C++
  - Various levels: physical layer, query layer, parser, application layer

1.7 Structure of a DBMS


USERS
Application tools
Query language interpreter
C language interface
Query langauge parser
Query optimization and execution
Relational operators
File system and access methods
Disk and memory management
DATA

- **Application tools**:
  - GUI interfaces, specialized applications
  - Usually written using in C/C++ using library
- **Query language interpreter**:
  - Puts out a prompt and reads in queries

- **Query language parser**:
  - Developed using a parser generator
  - Instead of code generation, generate function calls to lower layer
- **C interface**:
  - Library of C functions to `include` in C programs
  - Calls functions at various layers (depending on library)
- **Query optimizer**:
  - Analyze query and develop execution plan
  - Calls relational operators, lower layer functions
- **Relational layer**:
  - Functions that manipulate data in memory
  - Implementation of high-level "relational operators"
- **File system and access methods**:
  - Opening, writing and reading files
  - Index structures: hashing, B-trees
- **Disk and memory management**:
  - Often, a DBMS uses its own memory management
  - Layout of data on disk (clustering of related data)

1.8 Databases: Then and Now

- Ancient data storage: stone tablets, scrolls, paper, log books etc.
- Early days of computing: storage on cards, paper tape.
- 1950's:
  - first commercial uses of computers
  - census database
  - customer dbase for IBM
- 1960's:
  - IBM and others develop first few DBMS's
  - Based on files, hierarchical and network data models
  - nascent formalization of databases
- 1970's:
  - Relational dbase model (E.F.Codd)
  - significant research in sorting, searching, physical implementation
  - query languages: SQL, QUEL, QBE
  - several popular dbases: System R (IBM), Ingres (Berkeley)
  - transaction processing
- 1980's:
  - Dbase for PC's
  - Many additional features (GUI's, additional power)
  - research on distributed dbases, object-oriented dbases, client-server systems

  - Today's dbase giants: Oracle, Sybase, Informix
  - preliminary research on non-traditional dbases: geographic and spatial systems, image dbases, scientific dbases

- 1990's:
  - Geographic Information Systems (GIS), spatial databases
    * Store maps, city plans, road maps
    * Typical queries:
      "Find the 10 cities nearest nearest to Richmond"
      "Which counties are currently covered by Hurricane Bertha?"
      "Find all residential land parcels within 2 miles of the factory"
    * Try these websites:
      `http://www.mapquest.com`
      `http://tiger.census.gov/`
  - Image databases
    * Store images, video
    * Typical queries:
      "Find all images with a tree in them"
      "Find all images that look like the sample image"
      "Find all aerial images of the Williamsburg area"
    * Try these websites:
      `http://wwwqbic.almaden.ibm.com`
      `http://www-white.media.mit.edu/vismod/demos/photobook`
  - multimedia documents, text and document searching
    * Documents with text, audio and video
    * Typical queries:
      "Find documents with video clips of the President"
      "Find documents with references to Nuclear Disarmament"
    * Try `http://www.thomas.gov`

  - web interfaces
  - parallel dbases
    - How to parallelize query processing?
    - How to distribute data across processors?
    - Load balancing

- Emerging growth areas:
  - Non-traditional dbases:
    - Geographic and spatial dbases, Image dbases
    - Multimedia and temporal dbases (video, audio data), digital libraries
    - Scientific dbases (Medical dbases, CAD dbases, financial data, scientific data, DNA dbases)
    - Dbase tools: GUI-based query specification
    - Object-based, object-relational dbases

1.9 What this course is about

- **Course syllabus**:

  Relational Databases:

  - What is a database?
  - Data Models.
  - Relational databases: relational algebra.
  - Relational databases: SQL.
  - Example: Oracle.
  - Database programming in Oracle.
  - Physical implementation: file structures.
  - Physical implementation: indexing, B-trees, B+-trees, Hashing, Sorting.
  - Query processing.
  - Database design: normalization.
  - Recovery, concurrency.

  Additional topics: one or more of (time-permitting)

  - OLAP and business data processing.
  - Geographic Information Systems.
  - Text and pattern searching, approximate searching.
  - Distributed Databases.
  - Parallel databases.
  - Object-oriented databases.
  - Multimedia and temporal databases
  - Overview of text and data compression.

- **Coursework** (subject to change):
  - Programming Assignment 1: Implement some SQL queries in Oracle (Oracle is a relational dbase supporting SQL and several tools.)
  - Programming Assignment 2: More Oracle programming.
  - Programming Assignment 3: Indices, either B-Trees or Hashing.
  - Mid-term exam, in-class
  - Final project: a dbase application
  - Final exam.
- **Reading/review assignment for first week**
  - Review binary trees, hashing from any data structures book

Chapter 2

Relations and Relational Algebra

Course Notes on Database Systems

2.1 The Relational Data Model

- Recall what a data model is: *a convention for describing the logical organization of data.*
- Other data models: *hierarchical* and *network* models.
- Both the *hierarchical* and *network* models need a detailed understanding of the structure of the data to answer queries (even at a logical level).

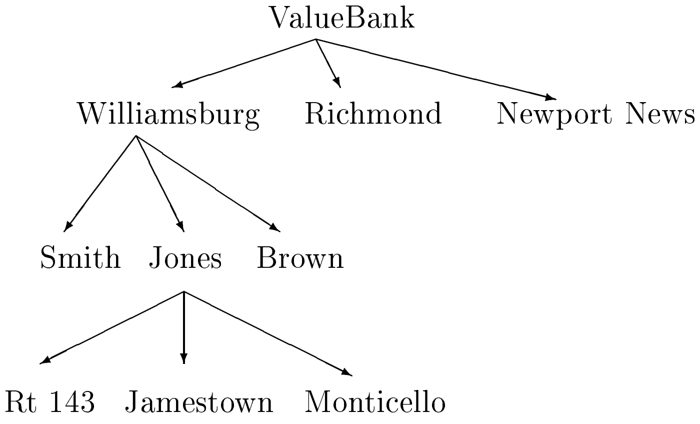
  ⇒ main goal of the relational model:

  - to describe a logical structure that is simple enough to minimize structural or navigational information in answering queries (at a logical level)

  A second goal:

  - to create a mathematically precise framework for the logical representation of data.

- To make sense of these terms, consider this example:
  VALUEBANK has branches in Richmond, Williamsburg, Newport News.
  - Each city has several branches, e.g., in Williamsburg: Monticello Rd, Jamestown Rd, Route 143.
  - Customers have accounts in various branches.
  - One obvious approach is *hierarchical*:



– Requirements: one needs to add/delete amounts, add/delete customers, add/delete branches

NOTE:

– Adding and deleting requires knowledge about *how* the data is stored

⇒ navigational information required

Reasoning about adding/deleting requires navigational information.

- Okay, but what does this really mean?
  - Suppose the hierarchical model City→Customer→Branch was used:
    - If all customers in the Jamestown Branch were deleted, we would lose the information "Jamestown Branch is in Williamsburg"

- In a non-hierarchical or "flat" model, we would store the data as:

CUST	BRANCH
Jones	Jamestown
Jones	Monticello
Jones	Route 143
Smith	Monticello
Brown	Route 143

BRANCH	CITY
Jamestown	Williamsburg
Monticello Rd	Williamsburg
Route 143	Williamsburg

Thus, deleting all "Jamestown" customers (from the first table) will not affect knowledge about the branches.

In other words, as long as we have the second table, any changes to other parts of the dbase will not affect the "branch location" information.

⇒ not necessarily true in the hierarchical model

2.2 Relational Model: Some definitions

- First, we need to review some elementary *set theory*:
  - A *set* is a collection of objects.
  - Usually, in math classes, examples consist of sets of numbers, e.g.

    $$A = \{17, 200, 523\},\ B = \{17, 413, 703, 804\}$$

  - We will consider more general sets, e.g.

    NAME = { Smith, Jones }
    COLOR = { blue, green, red }

  - Recall *cross-product of sets*: e.g.,

    NAME×COLOR = { (Smith,blue), (Smith,green), (Smith,red),
    (Jones,blue), (Jones,green), (Jones,red) }

    In general: $A \times B = \{(i, j) : i \in A, j \in B\}$
  - Cross-products extend to more than two sets: e.g.,
    - if CAR = {Ford, BMW, Toyota} then { (Smith,Toyota,blue), (Jones,BMW,green) } is a *subset* of NAME×CAR×COLOR
  - NOTE: order matters in a tuple
- Some informal definitions:
  - A *relation* is a "sort-of" cross-product of sets, e.g.,

    VEHICLE = NAME×CAR×COLOR

  - A *relation instance* is a particular subset of a relation, e.g.

    { (Smith,Toyota,blue), (Jones,BMW,green) }

    is an instance of VEHICLE.
  - What do we mean by "sort-of"? In a relation, *the order* of the sets is not important. Thus, whereas

    NAME×CAR×COLOR and NAME×COLOR×CAR

    are different sets, they are the same relation.

- We write the relation VEHICLE as:

  VEHICLE (NAME, CAR, COLOR)

  NOTE: While some order must be used in practice, the actual order is not relevant.
- Terminology:

Relation name:	VEHICLE
Attributes:	NAME, CAR, COLOR

- More formal definitions:
  - A *relation schema* $R(A_1, \ldots, A_n)$ consists of a relation name R and a list of attributes (or fields) $A_1, \ldots, A_n$.
  - The *domain* of attribute A_i is:

    $$\text{dom}(A_i) = \text{set of all possible values of attribute } A_i$$

    e.g.,

    $$\text{dom(CAR)} = \{\text{GM, Ford, Toyota, ... }\}$$
  - Values in a domain are *not* divisible: e.g.

    the letter ‘o’ in “Toyota” is not in dom(CAR)

    NOTE: commercial systems allow partial string searches, but the relational theory does not permit domain values to be subdivided.
  - A *tuple* from $R(A_1, \ldots, A_n)$ is an element of $\text{dom}(A_1) \times \ldots \times \text{dom}(A_n)$, e.g.,

    $$\text{(Smith,BMW,red)} \in \text{dom(NAME)} \times \text{dom(CAR)} \times \text{dom(COLOR)}$$
  - A *relation instance* is a set of *distinct* tuples from a relation schema, e.g.,

    (Smith,Toyota,blue),
    (Jones,GM,blue),
    (Brown,BMW,white),
    (Simpson,Ford,white)

    is a relation instance of

    VEHICLE (NAME, CAR, COLOR).

- A special **null** value is allowed in tuples, e.g.,

  (Jones, Honda,**null**)

  to signify lack of knowledge at the current time.

- A relation instance is a *set*. However, it is common to display or think of it as a *table*:

VEHICLE	NAME	CAR	COLOR
	Smith	Toyota	Blue
	Jones	GM	blue
	Brown	BMW	red
	Simpson	Ford	white
	Jones	Honda	**null**

  Here, the *row* correspond to tuples and the *columns* correspond to attributes (or fields).

- Although the term "relation" is broader than a particular instance, it is often used informally to refer to a particular instance.

- A *relational database schema* is a collection of relation schemas.

- A *relational database instance* is a particular instance of a relational database schema.

- Most installations typically have a number of disparate databases. Each database usually has a bunch of relation instances.

  Thus, an airline might have a reservations database with the following relations:

  - PASSENGER (NAME, SSN, FLT_ID, MILES)
    FLIGHT (FLT_ID, FLT_NO, START_APT, END_APT)
    AIRPORT (APT, NAME, CITY)
    CREW (SSN, FLT_ID)

  and may have a separate database for employee information:

  - EMPLOYEE (NAME, SSN, POSITION, SALARY, DEPT)
    MANAGERS (DEPT, NAME, MANAGER_SSN)
    BENEFITS (NAME, SSN, AMOUNT)

- We will use the following notation with regard to relation schema $R(A_1, \ldots, A_n)$:
  - $r(R)$ denotes a relation instance of schema R.
  - $t =< v_1, \ldots, v_n >$ denotes a particular tuple (with value v_i for attribute A_i.)
  - $t[A_i]$ denotes the value of the A_i attribute in tuple t.
    Thus, if $t =< v_1, \ldots, v_n >$ then $t[A_i] = v_i$.
  - If B is a subset of attributes, i.e., $B \subseteq \{A_1, \ldots, A_n\}$,
    $t[B]$ denotes the collection of values in t that correspond to the attributes in B.
  - For example, consider

    VEHICLE (NAME, CAR, COLOR).

    Suppose t =<Smith,Toyota,blue> and B ={CAR,COLOR}. Then:
    - t[CAR] = <Toyota>.
    - $t[B]$ = <Toyota,blue>.

2.3 Constraints on Relations

- For practical reasons, it is useful to allow constraints to be imposed on relations.
- *Domain constraints*:
  - we would like to define attribute domains as tightly as possible
    e.g., dom(AGE) = $\{0, \ldots, 120\}$ is preferable to dom(AGE) = all integers.
- *Key constraints*:
  First, what is a *key*?
  - Recall: a relation instance is a collection of **distinct** tuples.
  - Thus, if t_i and t_j are any two tuples, then $t_1[A_1, \ldots, A_n] \neq t_2[A_1, \ldots, A_n]$.
  - Now, it may be that there is a subset of attributes $B \subseteq \{A_1, \ldots, A_n\}$ such that $t_i[B] \neq t_j[B]$ for every t_i, t_j.
    $\Rightarrow$ such a subset of attributes is *possibly* a *superkey.* For example, in

VEHICLE	NAME	CAR	COLOR
	Smith	Toyota	Blue
	Jones	GM	blue
	Brown	BMW	red
	Simpson	Ford	white
	Jones	Honda	**null**

    with B=$\{NAME, CAR\}$ no two tuples have the same B-values (same NAME,CAR combination).
  - **Def**: A *superkey* on $R(A_1, \ldots, A_n)$ is a subset of attributes $S \subseteq \{A_1, \ldots, A_n\}$ such that $t_i[S] \neq t_j[S]$ for tuples t_i, t_j in *any relation instance* on R.

- Important: the superkey quality must hold for *all* possible relation instances. For example, suppose we have:

VEHICLE	NAME	CAR	COLOR
	Smith	Toyota	Blue
	Jones	GM	blue
	Brown	BMW	red
	Simpson	Ford	white
	Jones	Honda	**null**

  At this time, it appears that (NAME, CAR) form a superkey. But at some later stage we might want to add: <Smith,Toyota,green>.
  ⇒ (NAME, CAR) is not a superkey.

- Suppose we created the relation

  VEHICLE2 (NAME, CAR, COLOR, STATE, LICENSE_PLATE).

  Then,

  (NAME, CAR, STATE, LICENSE_PLATE)

  is a superkey.

- **Def**: A *key* K on $R(A_1, \ldots, A_n)$ is a superkey of R such that removing any attribute from K leaves a set of attributes that is not a superkey.
  ⇒ a key is a *minimal* superkey.

- For example, in

  VEHICLE2 (NAME, CAR, COLOR, STATE, LICENSE_PLATE),

  the attribute set {STATE, LICENSE_PLATE} is a key.

- NOTE: the value of a key (or superkey) uniquely identifies a tuple (since every tuple has a unique key value).

- Remember: a key or superkey is a property of attributes, not a property of a particular relation instance.

- If a relation has many keys, it is convenient to designate one of them as the *primary key.* e.g.,

  VEHICLE3 (NAME, CAR, COLOR, STATE, LICENSE_PLATE, ENGINE_SERIAL) has two keys, {STATE,LICENSE_PLATE} and {ENGINE_SERIAL}.

- In books, it is common to indicate primary keys by underlining the attributes, e.g.,

  VEHICLE3 (NAME, CAR, COLOR, STATE,LICENSE_PLATE, ENGINE_SERIAL)

Finally, some constraints on keys:

1. Relations with no keys should be avoided.
   For example, in

   VEHICLE (NAME, CAR, COLOR)

   if a person can own two green Toyota's, then we would not be able to store both tuples (no duplicates allowed)
   ⇒ should instead use
   VEHICLE (NAME, CAR, COLOR, STATE, LICENSE_PLATE).

2. Primary keys should not contain **null** values.
   ⇒ this is a practical constraint: we need the key to distinguish between tuples

- *Referential integrity constraint*:

  - First, we need the defintion of a *foreign key*:
    - **Def**: A set of attributes F of schema R_1 is a *foreign key* of R_1 with respect to R_2 if F is the primary key for R_2.
    - For example, consider relations
      PASSENGER (NAME, SSN, FLT_ID, MILES)
      FLIGHT (FLT_ID, FLT_NO, START_APT, END_APT)

      Here, FLT_ID is the primary key for FLIGHT.
      ⇒ FLT_ID is a foreign key in PASSENGER.
  - The foreign key (or referential integrity) constraint is: *Given a particular database instance containing relations R_1 and R_2 in which F is a foreign key in R_1 with respect to R_2, then for any tuple t in R_1, there must exist a tuple s in R_2 such that $t[F] = s[F]$.*
  - Informally, if the value 63 occurs as FLT_ID in PASSENGER, there had better be a tuple containing such a FLT_ID value in FLIGHT.

– For example:

PASSENGER	NAME	SSN	FLT_ID	MILES
	Bill	221-66-1234	17	2000
	Al	306-77-1131	$\boxed{63}$	45000
	Tom	636-22-9999	12	55000

FLIGHT	FLT_ID	FLT_NO	START_APT	END_APT
	11	F616	DCA	LGA
	15	F335	DCA	JFK

Here, the value 63 occurs is not to be found in FLIGHT
⇒ the foreign key constraint on PASSENGER is violated.

– Why the constraint? Essentially, if an attribute like FLT_ID in PASSENGER is important enough to be a primary key in FLIGHT, then it likely leads to additional information, which we must be able to get for every tuple in PASSENGER.

– NOTE: the constraint applies to instances of relations.

- To summarize:

1. *Domain constraints*: proper typing of values
2. *Key constraint 1*: A relation should have a key.
3. *Key constraint 2*: Primary keys can't have **null** values.
4. *Foreign key constraint*: A relation shouldn't have a foreign key value that doesn't exist in the foreign relation.

2.4 Relational Algebra: Operations on Relations

In examples below, we will consider the following dbase for McVALUE AIRLINES:

PASSENGER	NAME	SSN	FLT_ID	MILES
	Bill	221-66-1234	17	2000
	Al	306-77-1131	63	45000
	Bob	111-22-3333	15	1600
	Jack	733-55-1122	15	7700
	Tom	636-22-9999	12	55000
	Trent	414-28-5850	11	200
	Dick	223-63-7771	17	64500
	Newt	828-81-6977	12	1570

FLIGHT	FLT_ID	FLT_NO	START_APT	END_APT
	11	F616	DCA	LGA
	12	F71	LGA	DCA
	15	F335	DCA	JFK
	17	F338	JFK	DCA
	63	F15	DCA	JFK

AIRPORT	APT	NAME	CITY
	DCA	National	Washington
	LGA	La Guardia	New York
	JFK	Kennedy	New York

CREW	SSN	FLT_ID
	011-44-2233	11
	313-62-7711	11
	442-11-3313	12
	722-55-1139	15
	011-44-2223	63
	011-44-2223	17
	313-62-7711	17
	442-11-3313	63

EMP	NAME	SSN	POSITION	SALARY	MGRSSN
	Erskine	011-44-2223	Co-Pilot	33,000	313-62-7711
	Hillary	313-62-7711	Pilot	39,000	334-56-9876
	Newt	442-11-3313	Steward	26,000	313-62-7711
	Donna	722-55-1139	Engineer	37,000	334-56-9876
	Scott	221-44-8883	Control Tower	29,000	722-55-1139
	Liz	119-72-3131	Sales	34,000	334-56-9876
	Morris	334-56-9876	CEO	42,960,000	334-56-9876

- **UPDATE operations**:
  - Operations:
    1. Insert a tuple into a relation
    2. Delete a tuple from a relation
    3. Modify a tuple in a relation
  - **Insert**. Consider the McVALUE dbase and the following examples:
    1. `Insert <Rob, 011-44-2223, Accounts> into EMP`
       ⇒ not allowed: it would violate key constraint on SSN (011-44-2223 already exists).
    2. `Insert <Jesse, 666-23-1111, 25, null> into PASSENGER`
       ⇒ not allowed: foreign key constraint violated (25 does not exist as a FLT_ID in FLIGHT).
    3. `Insert <666-23-2223, null> into CREW`
       ⇒ not allowed: **null** not permitted in primary key
    4. `Insert <George, 554-12-1234, Communications> into EMP`
       ⇒ acceptable
  - Similarly, **delete**'s and **modify**'s need to satisfy constraints.

- **RETRIEVAL operations**: the Relational Algebra
  - These operations act on relations and produce relations.
  - The collection of these operations (or operators) is called the *relational algebra.*

- The **select** operation.

- Applies to a single relation.
- A condition (boolean expression) is often specified.
- The result is a *new* relation containing the result of the operation
  ⇒ all those tuples satisfying the condition.
- Notation: $\sigma_{\text{<condition>}}$ (<relation>) .
- Example: $\sigma_{\text{CITY='New York'}}$ (AIRPORT) . The result is the relation:

APT	NAME	CITY
LGA	La Guardia	New York
JFK	Kennedy	New York

- Example: $\sigma_{\text{MILES}>60000}$ (PASSENGER)

NAME	SSN	FLT_ID	MILES
Dick	223-63-7771	17	64500

  NOTE: the result is a relation, not a tuple.
- Example: $\sigma_{\text{MILES}>\ 90000}$ (PASSENGER)

NAME	SSN	FLT_ID	MILES
empty			

  ⇒ empty relation (no tuples satisfied the condition)
- Observation: σ () is *commutative*:

$$\begin{aligned}\sigma_{\text{<cond1>}}\ (\ \sigma_{\text{<cond2>}}\ (\text{R})\ ) &= \sigma_{\text{<cond2>}}\ (\ \sigma_{\text{<cond1>}}\ (\text{R})\ ) \\ &= \sigma_{\text{<cond2>}\ \textbf{and}\ \text{<cond1>}}\ (\text{R})\end{aligned}$$

- NOTE:

1. We will often informally write a query in English before working out the relational algebra needed to satisfy the query.

   For example: "Find all employees who are Pilots"
   ⇒ The solution is: $\sigma_{\text{POSITION='Pilot'}}$ (EMP) .
2. Although not strictly part of standard relational algebra, we will allow new relations to be defined via assignment:

$$\text{PILOTS} := \sigma_{\text{POSITION='Pilot'}}\ (\text{EMP})\ .$$

3. Comparison operators like "<" apply to attributes whose domains are naturally ordered (like numbers or strings).

- The **project** operation.
  - Applies to a single relation.
  - A list of attributes must be specified.
  - The result is a *new* relation containing the result of the operation $\Rightarrow$ all tuples, but only those attributes specified in list.
  - Notation: $\Pi_{\text{<attribute-list>}}$ (<relation>)
  - Example: $\Pi_{\text{NAME,SSN}}$ (EMP) . The result is the relation:

NAME	SSN
Erskine	011-44-2223
Hillary	313-62-7711
Newt	442-11-3313
Donna	722-55-1139
Scott	221-44-8883
Liz	119-72-3131

  - Example: Π_{NAME} (EMP) . The result is the relation:

NAME
Erskine
Hillary
Newt
Donna
Scott
Liz

  - Example: "Create a list of cities served by the airline". Here, one solution is:

$$\Pi_{\text{CITY}} \ \text{(AIRPORT)}$$

which gives:

CITY
Washington
New York

- NOTE: the result above is NOT

CITY
Washington
New York
New York

  That is, duplicates are removed in the result (since duplicate tuples are not allowed).
  However, many commercial dbases leave duplicates in the result.

- Combining relational operators.
  - Since relational operators return relations as results, operators can be combined.
  - For example: "List the names (and only the names) of employees that are pilots". This query can be expressed as:
    $$\Pi_{\text{NAME}} \; (\; \sigma_{\text{POSITION='Pilot'}} \; (\text{EMP}) \;)$$

- The **union** operation.
  - Applies to *two* relations, that are *union-compatible* (defined below).
  - The result is a new relation containing the result of the operation.
    ⇒ the collection of tuples in both relations.
  - Notation: $R_1 \cup R_2$.
  - Suppose we have the relation

CANDIDATE	FNAME	LNAME	OFFICE
	Bill	Clinton	President
	Al	Gore	Vice-President
	Bob	Dole	President
	Jack	Kemp	Vice-President

    Consider the query: "List all names, first or last, in CANDIDATE":
    - $*$ Neither Π_{FNAME} (CANDIDATE) nor Π_{LNAME} (CANDIDATE) is sufficient.
    - $*$ Instead:
      $$\Pi_{\text{FNAME}} \; (\text{CANDIDATE}) \; \cup \; \Pi_{\text{LNAME}} \; (\text{CANDIDATE})$$

- NOTE: the two relations must be *union-compatible*: *Relations* $R_A(A_1, \ldots, A_n)$ *and* $R_B(B_1, \ldots, B_n)$ *are* union-compatible *iff* $\forall i : dom(A_i) = dom(B_i)$.
- For union-compatibility, the *names* of the attributes in R_A and R_B don't have to be identical.
- Example: above, we assumed

$$\text{dom(FNAME)} = \text{dom(LNAME)}$$

- But, what about the attribute names of the result?

  ⇒ The result takes on the attribute names of the first relation.
- Thus, the result of

$$\Pi_{\text{FNAME}}\ \text{(CANDIDATE)} \cup \Pi_{\text{LNAME}}\ \text{(CANDIDATE)}$$

  is

FNAME
Bill
Al
Bob
Jack
Clinton
Gore
Dole
Kemp

- *Re-naming* attributes:
  - Often, it is not satisfactory to use the attribute name from one of the relations. It is common to extend the relational algebra to permit *re-naming* of attributes:

    $$\text{TEMP(ANYNAME)} := \Pi_{\text{FNAME}} \text{ (CANDIDATE)} \cup \Pi_{\text{LNAME}} \text{ (CANDIDATE)}$$

    which produces:

TEMP	ANYNAME
	Bill
	Al
	Bob
	Jack
	Clinton
	Gore
	Dole
	Kemp

  - Multiple attributes can be renamed, as in:

    $$\text{TEMP(FIRSTN, LASTN, GOAL)} := \text{CANDIDATE}$$

- **Intersection** and **Difference** operators:
  - Intersection and difference are defined on union-compatible relations.
  - $R \cap S$ = tuples in *both* R and S.
  - Example: "Find names that occur both as passenger and as staff in the airline dbase":

    $$\Pi_{\text{NAME}} \text{ (PASSENGER)} \cap \Pi_{\text{NAME}} \text{ (EMP)}$$

  - $R - S$ = tuples in R that are *not* in S.
  - Note that $\cup$ and $\cap$ are commutative and associative with respect to each other.

- **Cartesian product**.
  - Applies to two relations.
  - The result is a new relation.
  - Suppose $R(A_1, \ldots, A_n)$ and $S(B_1, \ldots, B_m)$ are two relations. Then $R \times S$ is a relation with attributes $A_1, \ldots, A_n, B_1, \ldots, B_m$ created as follows: for each possible pair of tuples from R and S, concatenate them and place the result in $R \times S$.
  - $|R \times S| = |R||S|$.
  - Example: Consider the following two relations:

R	NAME	CITY
	Lennon	NY
	McCartney	DC
	Harrison	NY
	Starr	LA

S	FLOWER	COLOR
	Rose	red
	Tulip	yellow

Then, the cross-product, $R \times S$ is:

$R \times S$	NAME	CITY	FLOWER	COLOR
	Lennon	NY	Rose	red
	Lennon	NY	Tulip	yellow
	McCartney	DC	Rose	red
	McCartney	DC	Tulip	yellow
	Harrison	NY	Rose	red
	Harrison	NY	Tulip	yellow
	Starr	LA	Rose	red
	Starr	LA	Tulip	yellow

The results are not necessarily meaningful.

– Consider another example:

EMP	NAME	DEPTNO
	Armstrong	3
	Ellington	3
	Bach	1

DEPT	DEPTNUM	DNAME
	3	Jazz
	1	Classical

Consider what we get with the following relational expression:

$$\Pi_{\text{NAME,DNAME}} \ (\ \sigma_{\text{DEPTNO=DEPTNUM}} \ (\text{EMP} \times \text{DEPT}) \ )$$

First, EMP × DEPT gives

EMP×DEPT	NAME	DEPTNO	DEPTNUM	DNAME
	Armstrong	3	3	Jazz
	Armstrong	3	1	Classical
	Ellington	3	3	Jazz
	Ellington	3	1	Classical
	Bach	1	3	Jazz
	Bach	1	1	Classical

Next, $\sigma_{\text{DEPTNO=DEPTNUM}}$ (EMP × DEPT) gives

EMP×DEPT	NAME	DEPTNO	DEPTNUM	DNAME
	Armstrong	3	3	Jazz
	Ellington	3	3	Jazz
	Bach	1	1	Classical

Finally, after the projection we get:

NAME	DNAME
Armstrong	Jazz
Ellington	Jazz
Bach	Classical

We have answered the query: "List employees along with their department names".

– The cross-product operation is rarely used. Instead, a *join* is commonly used.

- The **join** operation.
  - Applies to two relations.
  - The result is a relation.
  - Notation (general form): R $\bowtie_{<\text{condition}>}$ S
  - Recall the previous example

EMP	NAME	DEPTNO
	Armstrong	3
	Ellington	3
	Bach	1

DEPT	DEPTNUM	DNAME
	3	Jazz
	1	Classical

    and the query "List employees along with their department names".

    ⇒ what we really need to do is scan the EMP relation and, for every DEPTNO that occurs, we need to look up the DNAME in the DEPT relation

  - This kind of combining of information in two tables using a common attribute (DEPTNO and DEPTNUM, above) is very common in dbase applications.
  - A special operator, called a *join*, was created just for this purpose.
  - Example (above): The result of EMP $\bowtie_{\text{DEPTNO=DEPTNUM}}$ DEPT is:

NAME	DEPTNO	DEPTNUM	DNAME
Armstrong	3	3	Jazz
Ellington	3	3	Jazz
Bach	1	1	Classical

    Thus, a **join** is like a cross-product, except that the *join condition* is applied to filter out non-matching tuples.

  - The appearance of both DEPTNO and DNUM is a waste (since both entries are identical for each tuple)

- The **natural join** operator (denoted $*$) removes duplicate attributes but *assumes that the join attributes have the same name.* Thus,

  $$\text{DEPT2 (DEPTNO, DNAME)} \ := \ \text{DEPT}$$
  $$\text{RESULT} \ := \ \text{EMP} * \text{DEPT2}$$

  produces

RESULT	NAME	DEPTNO	DNAME
	Armstrong	3	Jazz
	Ellington	3	Jazz
	Bach	1	Classical

- If two or more attributes have same names in R and S then $R * S$ 'joins' on *all* these common attributes.
  ⇒ that is, tuples are matched when *all* these attribute values are equal.

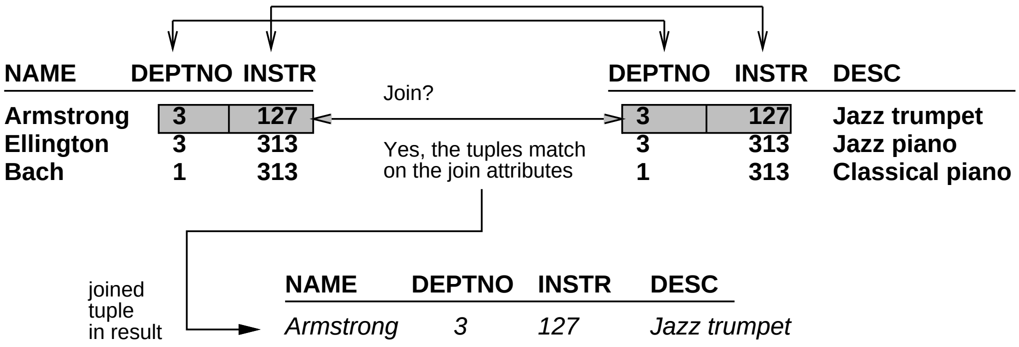
- Joins can be constructed using two nested loops (one for each relation). For example, consider the natural join of

EMP3	NAME	DEPTNO	INSTRUMENT
	Armstrong	3	127
	Ellington	3	313
	Bach	1	313

DEPT3	DEPTNO	INSTRUMENT	DESC
	3	127	Jazz trumpet
	3	313	Jazz piano
	1	474	Classical violin
	1	313	Classical piano

  The steps in joining are:

  - ∗ Step 1:

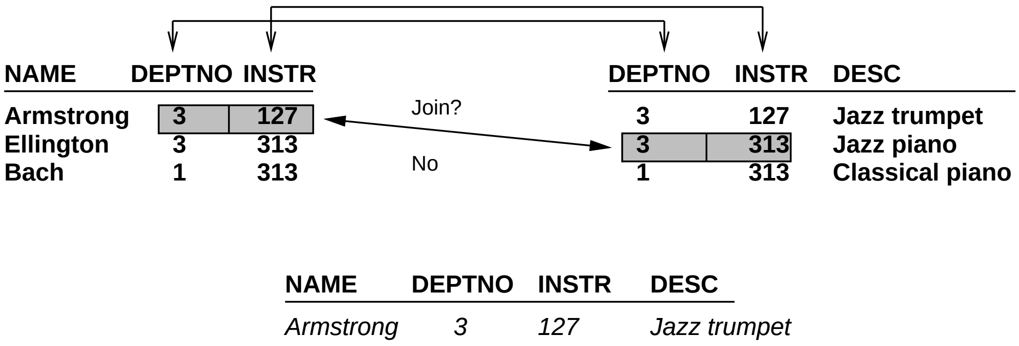


NAME	DEPTNO	INSTR
Armstrong	3	127
Ellington	3	313
Bach	1	313

DEPTNO	INSTR	DESC
3	127	Jazz trumpet
3	313	Jazz piano
1	313	Classical piano

NAME	DEPTNO	INSTR	DESC
Armstrong	*3*	*127*	*Jazz trumpet*

∗ Step 2:

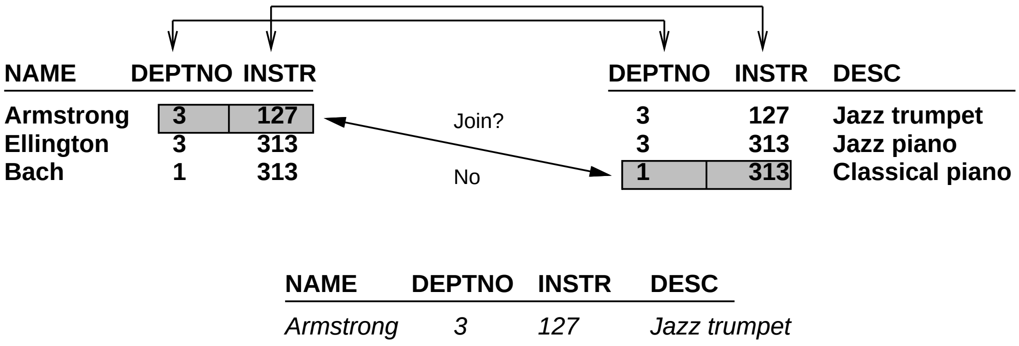


NAME	DEPTNO	INSTR
Armstrong	3	127
Ellington	3	313
Bach	1	313

DEPTNO	INSTR	DESC
3	127	Jazz trumpet
3	313	Jazz piano
1	313	Classical piano

NAME	DEPTNO	INSTR	DESC
Armstrong	*3*	*127*	*Jazz trumpet*

∗ Step 3:

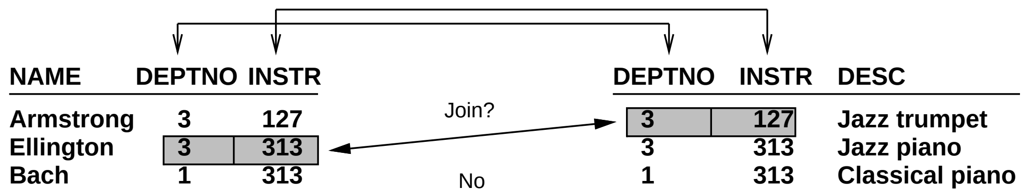


NAME	DEPTNO	INSTR
Armstrong	3	127
Ellington	3	313
Bach	1	313

DEPTNO	INSTR	DESC
3	127	Jazz trumpet
3	313	Jazz piano
1	313	Classical piano

NAME	DEPTNO	INSTR	DESC
Armstrong	*3*	*127*	*Jazz trumpet*

∗ Step 4:

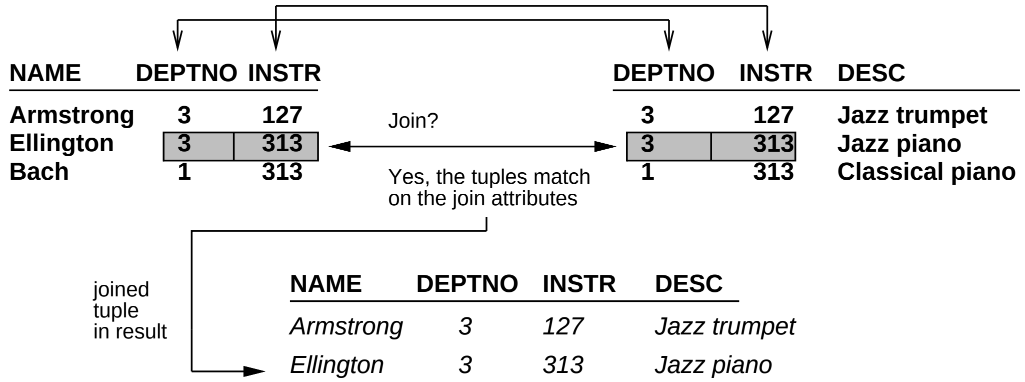


NAME	DEPTNO	INSTR
Armstrong	3	127
Ellington	3	313
Bach	1	313

DEPTNO	INSTR	DESC
3	127	Jazz trumpet
3	313	Jazz piano
1	313	Classical piano

NAME	DEPTNO	INSTR	DESC
Armstrong	*3*	*127*	*Jazz trumpet*

∗ Step 5:

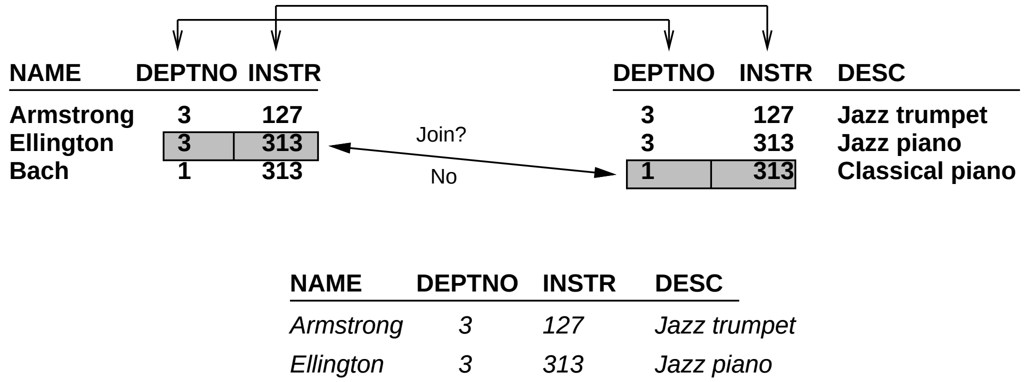


NAME	DEPTNO	INSTR
Armstrong	3	127
Ellington	3	313
Bach	1	313

DEPTNO	INSTR	DESC
3	127	Jazz trumpet
3	313	Jazz piano
1	313	Classical piano

NAME	DEPTNO	INSTR	DESC
Armstrong	*3*	*127*	*Jazz trumpet*
Ellington	*3*	*313*	*Jazz piano*

∗ Step 6:



NAME	DEPTNO	INSTR
Armstrong	3	127
Ellington	3	313
Bach	1	313

DEPTNO	INSTR	DESC
3	127	Jazz trumpet
3	313	Jazz piano
1	313	Classical piano

NAME	DEPTNO	INSTR	DESC
Armstrong	*3*	*127*	*Jazz trumpet*
Ellington	*3*	*313*	*Jazz piano*

... and so on. Finally, EMP3∗DEPT3 produces:

EMP3∗DEPT3	NAME	DEPTNO	INSTR	DESC
Armstrong	3	127	Jazz trumpet	
Ellington	3	313	Jazz piano	
Bach	1	313	Classical piano	

– NOTE:

∗ A join (natural or otherwise) can produce the empty relation, if no matches occur.

∗ A natural join between two relations that have no attributes in common (e.g., $R(A, B)$ and $S(C, D)$) produces an empty relation.

∗ The general join ($\bowtie$) is sometimes called a **theta-join**.

- The **division** operator.
  - Applies to two relations.
  - The result is a relation.
  - Let $R(A), S(B)$ and $Q(C)$ be three relations with attribute sets A, B and C respectively such that
    1. $B \subset A$
    2. $C = A - B$

    Then, $Q = R \div S$ if Q is the largest relation such that $Q \times S \subseteq R$.
  - Example:

STUDENT	NAME	COURSE
	Louis	CS 131
	Louis	CS 141
	Duke	CS 131
	Johann	CS 131
	Johann	CS 141

ALLCOURSES	COURSE
	CS 131
	CS 141

* Note that one of the conditions for division is satisfied:
  {COURSE} ⊂ {NAME, COURSE}
* Thus, the result of
  STUDENT ÷ ALLCOURSES

  should have the schema RESULT(NAME).
* Consider

RESULT1	NAME
	Duke

The cross-product RESULT1 × ALLCOURSES produces

RESULT1 × ALLCOURSES	NAME	COURSE
	Duke	CS 131
	Duke	CS 141

in which <Duke, CS 141> is *not* a tuple in STUDENT.
$\Rightarrow$ RESULT1 cannot be the quotient.

∗ Consider

RESULT2	NAME
	Louis

The cross-product RESULT2 × ALLCOURSES produces

RESULT2 × ALLCOURSES	NAME	COURSE
	Louis	CS 131
	Louis	CS 141

which *is* a subset of STUDENT.

⇒ is RESULT2 the largest relation that satisfies this property?

∗ Consider

RESULT	NAME
	Louis
	Johann

The cross-product RESULT × ALLCOURSES produces

NAME	COURSE
Louis	CS 131
Louis	CS 141
Johann	CS 131
Johann	CS 141

which is a subset of STUDENT.

⇒ it is the largest such relation.

⇒ RESULT = STUDENT ÷ COURSE.

∗ Notice that the operation solves the query: “Find the names of students who have taken *all* courses”.

∗ The appearance of “all” in a query is often an indicator that division may be appropriate.

- The **outer join** operators.
  - Similar to a join, but asymmetric.
  - Consider this example:

EMP	NAME	DEPTNO
	Armstrong	3
	Ellington	3
	Bach	5

DEPT	DEPTNO	DNAME
	3	Jazz
	4	Country

Here, the join EMP$*$DEPT produces:

RESULT	NAME	DEPTNO	DNAME
	Armstrong	3	Jazz
	Ellington	3	Jazz

⇒ the tuple <Bach,5> is not represented in the result.

  - Sometimes it is desirable to include tuples from one or the other relations in a join, *even if no match occurs*
    ⇒ if no match occurs, use a **null**
  - The **left outer join** includes all tuples from the "left" relation, using **null**'s where necessary.

The left outer join, EMP $\overset{L}{\bowtie}$DEPT, produces:

EMP $\overset{L}{\bowtie}$DEPT	NAME	DEPTNO	DNAME
	Armstrong	3	Jazz
	Ellington	3	Jazz
	Bach	5	**null**

  - The **right outer join** includes all tuples from the "right" relation, using **null**'s where necessary.

The right outer join, EMP $\overset{R}{\bowtie}$DEPT, produces:

EMP $\overset{R}{\bowtie}$DEPT	NAME	DEPTNO	DNAME
	Armstrong	3	Jazz
	Ellington	3	Jazz
	null	4	Country

  - A **full outer join** is the union of the the left and right outer joins.

2.5 Examples of Queries

The following examples are based on the McVALUE AIRLINES dbase:

- **Example 1**:
  - *Query*: "Find the names and social security numbers of all passengers who have accumulated at least 50,000 miles".
  - *Analysis*: All the information is in PASSENGER. We need to select a subset of tuples and project only some attributes.
  - *Solution*:
    $\text{RESULT} := \Pi_{\text{NAME,SSN}} \; (\; \sigma_{\text{MILES}>50000} \; (\text{PASSENGER}) \;)$
  - *Result*: $\sigma_{\text{MILES}>50000}$ (PASSENGER) produces

NAME	SSN	FLT_ID	MILES
Tom	636-22-9999	12	55000
Dick	223-63-7771	17	64500

Projecting out NAME and SSN gives:

RESULT	NAME	SSN
	Tom	636-22-9999
	Dick	223-63-7771

- **Example 2**:
  - *Query*: "List names and ssn's of all passengers flying on Flight F338"
  - *Analysis*: The names of passengers and flight data are in different relations, PASSENGER and FLIGHT. We need to match FLT_ID in these two relations.
    ⇒ use a join.
  - *Solution*:

$$\text{RESULT} := \Pi_{\text{NAME,SSN}}\ (\ \sigma_{\text{FLTNO=F338}}\ (\text{PASSENGER} * \text{FLIGHT})\ )$$

  - *Result*: PASSENGER $*$ FLIGHT produces

NAME	SSN	FLT_ID	MILES	FLT_NO	START_APT	END_APT
Bill	221-66-1234	17	2000	F338	JFK	DCA
Al	306-77-1131	63	45000	F15	DCA	JFK
Bob	111-22-3333	15	1600	F335	DCA	JFK
Jack	733-55-1122	15	7700	F335	DCA	JFK
Tom	636-22-9999	12	55000	F71	LGA	DCA
Trent	414-28-5850	11	200	F616	DCA	LGA
Dick	223-63-7771	17	64500	F338	JFK	DCA
Newt	828-81-6977	12	1570	F71	LGA	DCA

From this, $\sigma_{\text{FLTNO=F338}}$ (PASSENGER $*$ FLIGHT) produces

NAME	SSN	FLT_ID	MILES	FLT_NO	START_APT	END_APT
Bill	221-66-1234	17	2000	F338	JFK	DCA
Dick	223-63-7771	17	64500	F338	JFK	DCA

The attributes NAME and SSN are projected out to give:

RESULT	NAME	SSN
	Bill	221-66-1234
	Dick	223-63-7771

  - Note alternative solution:

$$\text{RESULT} := \Pi_{\text{NAME,SSN}}\ (\ \text{PASSENGER} * \sigma_{\text{FLTNO=F338}}\ (\text{FLIGHT})\ )$$

Q: which one might be more efficient to implement?

- **Example 3**:
  - *Query*: “List names and ssn’s of all passengers flying into National airport”
  - *Analysis*: To see if a particular passenger is flying into National airport, we’ll have to use the FLT_ID to look up the ENDAPT in FLIGHT, then use the ENDAPT value to look up NAME in AIRPORT.

    ⇒ a join between PASSENGER and FLIGHT will be needed, as well as a join between FLIGHT and AIRPORT.
  - *Solution*: Let us break this into four steps:
    1. First, we extract National airport from AIRPORT:

       $$\text{NAT} := \sigma_{\text{NAME='National'}}\ (\text{AIRPORT})$$

NAT	APT	NAME	CITY
	DCA	National	Washington

    2. Second, we find all the FLT_ID’s that correspond to flights that terminate at National:

       $$\text{NAT\_FLTS} := \text{FLIGHT} \bowtie_{\text{ENDAPT=APT}} \text{NAT}$$

NAT_FLTS	FLT_ID	FLT_NO	START_APT	END_APT	APT	NAME	CITY
	12	F71	LGA	DCA	DCA	National	Washington
	17	F338	JFK	DCA	DCA	National	Washington

    3. Third, we only need the FLT_ID’s:

       $$\text{NAT\_FLTID} := \Pi_{\text{FLT\_ID}}\ (\text{NAT\_FLTS})$$

NAT_FLTID	FLT_ID
	12
	17

    4. Last, we join with PASSENGER and project out NAME and SSN:

       $$\text{RESULT} := \Pi_{\text{NAME,SSN}}\ (\text{PASSENGER} * \text{NAT\_FLTID})$$

- *Result*:

RESULT	NAME	SSN
	Bill	221-66-1234
	Tom	636-22-9999
	Dick	223-63-7771
	Newt	828-81-6977

- **Example 4**:
  - *Query*: "List all flights (FLTNO) that have either a passenger named Newt or a crew member named Newt"
  - *Analysis*: We need to find flights containing passengers called Newt and flights containing a crew member called Newt and put these together.

    ⇒ a join between PASSENGER and FLIGHT, a join between EMP, CREW and FLIGHT, and a union.
  - *Solution*:
    1. First, we obtain FLTNO's of Newt-passengers:

       $\text{NEWT\_PASS} := \Pi_{\text{FLTNO}} \; (\; \sigma_{\text{NAME='Newt'}} \; (\text{PASSENGER} * \text{FLIGHT}) \;)$

       Note that PASSENGER $*$ FLIGHT gives us:

NAME	SSN	FLT_ID	MILES	FLT_NO	START_APT	END_APT
Bill	221-66-1234	17	2000	F338	JFK	DCA
Al	306-77-1131	63	45000	F15	DCA	JFK
Bob	111-22-3333	15	1600	F335	DCA	JFK
Jack	733-55-1122	15	7700	F335	DCA	JFK
Tom	636-22-9999	12	55000	F71	LGA	DCA
Trent	414-28-5850	11	200	F616	DCA	LGA
Dick	223-63-7771	17	64500	F338	JFK	DCA
Newt	828-81-6977	12	1570	F71	LGA	DCA

From which, $\sigma_{\text{NAME='Newt'}}$ (PASSENGER $*$ FLIGHT) produces

NAME	SSN	FLT_ID	MILES	FLT_NO	START_APT	END_APT
Newt	828-81-6977	12	1570	F71	LGA	DCA

Projecting out the FLT_NO gives:

NEWT_PASS	FLT_NO
	F71

2. Next, FLT_NO's for Newt-crew.
Since neither CREW nor EMP contain FLTNO, we need to get FLT_ID first:

$$\text{NEWT\_CREW\_FID} := \Pi_{\text{FLT\_ID}} \; (\text{CREW} * \sigma_{\text{NAME='Newt'}} \; (\text{EMP}) \;)$$

Note that $\sigma_{\text{NAME='Newt'}}$ (EMP) gives:

NAME	SSN	POSITION	SALARY	MGRSSN
Newt	442-11-3313	Steward	26,000	313-62-7711

Then, CREW $*$ $\sigma_{\text{NAME='Newt'}}$ (EMP) results in:

FLT_ID	NAME	SSN	POSITION	SALARY	MGRSSN
12	Newt	442-11-3313	Steward	26,000	313-62-7711
63	Newt	442-11-3313	Steward	26,000	313-62-7711

Finally, projecting out FLT_ID gives:

NEWT_CREW_FID	FLT_ID
	12
	63

3. Now we can get a list of FLTNO's by joining with FLIGHT:

$$\text{NEWT\_CREW} := \Pi_{\text{FLT\_NO}} \; (\text{NEWT\_CREW\_FID} * \text{FLIGHT})$$

NEWT_CREW	FLT_NO
	F71
	F15

4. Finally, compute the desired result as a union:

$$\text{RESULT} := \text{NEWT\_PASS} \cup \text{NEWT\_CREW}$$

– *Result*:

RESULT	FLTNO
	F71
	F15

- **Example 5**:
  - *Query*: "List names of crew members who visit all airports".
  - *Analysis*: The word 'all' suggests the use of division.
  - *Solution*:

1. To use division, we first need to create a relation that has NAME and APT. Since CREW only has FLT_ID's and SSN's, we start by matching FLT_ID's with airports:

$$\begin{aligned} \text{F1} &:= \Pi_{\text{FLT\_ID,STARTAPT}}\ (\text{FLIGHT}) \\ \text{F2} &:= \Pi_{\text{FLT\_ID,ENDAPT}}\ (\text{FLIGHT}) \\ \text{F(FLT\_ID,APT)} &:= \text{F1} \cup \text{F2} \end{aligned}$$

   This gives:

F	FLT_ID	APT
	11	DCA
	12	LGA
	15	DCA
	17	JFK
	63	DCA
	11	LGA
	12	DCA
	15	JFK
	17	DCA
	63	JFK

2. Next, we match crew SSN's against these FLT_ID's:

$$\text{CREW\_APT} := \Pi_{\text{SSN,APT}}\ (\text{F} * \text{CREW})$$

   First, F $*$ CREW gives:

F ∗ CREW	FLT_ID	APT	SSN
	11	DCA	011-44-2233
	11	DCA	313-62-7711
	12	LGA	442-11-3313
	15	DCA	722-55-1139
	17	JFK	011-44-2233
	17	JFK	313-62-7711
	63	DCA	011-44-2233
	11	LGA	011-44-2233
	11	LGA	313-62-7711
	12	DCA	442-11-3313
	15	JFK	722-55-1139
	17	DCA	011-44-2233
	17	DCA	313-62-7711
	63	JFK	442-11-3313

Second, the projection of SSN and APT gives:

CREW_APT	APT	SSN
	DCA	011-44-2233
	JFK	011-44-2233
	LGA	011-44-2233
	DCA	313-62-7711
	JFK	313-62-7711
	LGA	313-62-7711
	LGA	442-11-3313
	DCA	442-11-3313
	JFK	442-11-3313
	DCA	722-55-1139
	JFK	722-55-1139

Note: some duplicates are removed.

3. For the division, we will need a list of all airports:

$$\text{APTS} := \Pi_{\text{APT}} \ (\text{AIRPORT})$$

APTS	APT
	DCA
	LGA
	JFK

4. Now we can perform the division:

$$\text{ALLAPT\_CREW} := \text{CREW\_APT} \div \text{APTS}$$

This produces:

ALLAPT_CREW	SSN
	011-44-2233
	313-62-7711
	442-11-3313

5. Finally, we associate names:

$$\text{RESULT} := \Pi_{\text{NAME}} \ (\text{ALLAPT\_CREW} * \text{EMP})$$

- *Result*:

RESULT	NAME
	Erskine
	Hillary
	Newt

- **Example 6**:

  - *Query*: “List names and ssn’s of all employees not on any flight”.
  - *Analysis*: it’s easy to find ’flying’ employees from CREW. The remainder can be found by using set difference.
  - *Solution*:

1. First, the employees that fly:

$$\text{FLIERS} := \Pi_{\text{SSN}} \ (\text{CREW})$$

This gives:

FLIERS	SSN
	011-44-2233
	313-62-7711
	442-11-3313
	722-55-1139

2. Next, the list of SSN’s of all employees:

$$\text{ALLSSN} := \Pi_{\text{SSN}} \ (\text{EMP})$$

This gives:

ALLSSN	SSN
	011-44-2223
	313-62-7711
	442-11-3313
	722-55-1139
	221-44-8883
	119-72-3131
	334-56-9876

3. Now we can get the ssn's of nonfliers:

$$\text{NONFLIERS} := \text{ALLSSN - FLIERS}$$

   This produces:

NONFLIERS	SSN
	221-44-8883
	119-72-3131
	334-56-9876

4. Finally, the names (in addition to ssn's):

$$\text{RESULT} := \Pi_{\text{NAME,SSN}} \; (\text{NONFLIERS} * \text{EMP})$$

– *Result*:

RESULT	NAME	SSN
	Scott	221-44-8883
	Liz	119-72-3131
	Morris	334-56-9876

- **Example 7**:

– *Query*: "Find the names of all airports that occur only as a START_APT or only as an END_APT" (thereby identifying a problem)

– Analysis: Use intersection to find all airports that occur as both. Remove from union of all airports using set difference.

– *Solution*:

1. First, project out the airport codes:

$$\begin{array}{lll} \text{START} & := & \Pi_{\text{START\_APT}} \; (\text{FLIGHT}) \\ \text{END} & := & \Pi_{\text{END\_APT}} \; (\text{FLIGHT}) \end{array}$$

   This gives:

START	START_APT
	DCA
	LGA
	JFK

END	END_APT
	LGA
	DCA
	JFK

2. Next, obtain the relevant airports:

$$\text{BAD\_APTS} := (\text{START} \cup \text{END}) - (\text{START} \cap \text{END})$$

   Empty result.

3. Finally, join with AIRPORT to obtain names:

$$\text{RESULT} := \Pi_{\text{NAME}} \; (\text{AIRPORT} * \text{BAD\_APTS})$$

4. *Result*: The result happens to be empty:

RESULT	NAME
	empty

- **Example 8**:
  - *Query*: "Identify all crew members who are listed both as a passenger and a crew member for the same flight".
  - *Analysis*: this information can be obtained from PASSENGER and CREW. A natural join matches NAME and FLT_ID, giving us the result.
  - *Solution*:

$$\text{RESULT} := \Pi_{\text{NAME}} \; (\text{PASSENGER} * \text{CREW})$$

- **Example 9**:
  - *Query*: "Identify all crew members (SSN's) who take a flight as a passenger but not as a crew member".
  - Analysis: the natural join above cannot be *projected* or *selected* to get this result. It forces both NAME and FLT_ID to be matched.
  - *Solution*: use a theta-join (where the join condition is specified):

$$\text{RESULT} := \Pi_{\text{SSN}} \; (\text{ CREW} \bowtie_{\text{SSN=SSN,FLT\_ID}\neq\text{FLT\_ID}} \; \text{PASSENGER })$$

- For more examples, read Elmasri/Navathe, sections 6.3-6.7, or Chapter 2 of O'Neil, or Chapter 8 of Ramakrishnan.

2.6 Relational Calculus: Another Way to Specify Queries

- The **relational algebra** is a *formal language* for specifying queries on relations.
- The **relational calculus** is another formal language.
- The relational calculus comes in two flavors:
  1. **tuple relational calculus**
  2. **domain relational calculus**
- The difference between relational algebra and relational calculus:
  - In *relational algebra*: expressions are **procedural**
    ⇒ expressions provide the order of computations
    For example, in
    $$\Pi_{\text{NAME,SSN}} \;(\; \sigma_{\text{MILES}>50000} \;(\text{PASSENGER})\;)$$
    one scans PASSENGER and extracts tuples in which MILES>50000. Then, among these tuples the NAME,SSN attributes are reported.
  - In *relational calculus*: expressions are **non-procedural**
    ⇒ expressions only specify the desired result
    For example, in tuple relational calculus the above query would be written as:
    $$\{\; x.\text{NAME},\, x.\text{SSN} \mid \text{PASSENGER}(x) \text{ and } x.\text{MILES}>50000 \;\}$$
    Here x is a *tuple-variable*.
- General form of a query in tuple relational calculus:

$$\{\; x.A_1, \ldots, x.A_n \mid \text{<condition>}(x_1, \ldots, x_n, x_{n+1}, \ldots, x_m) \;\}$$

- What does it all mean? Let's look at an example.
  Consider the example:

  $$\{\ x.\text{NAME},\ x.\text{SSN} \mid \text{PASSENGER}(x) \text{ and } x.\text{MILES}{>}50000\}$$

  - Think of the the variable x as a for-loop variable that scans through tuples in PASSENGER, as in:

    for x := <first-tuple> **to** <last-tuple>

  - As x scans the tuples, the condition on the right is checked (x.MILES>50000). If the condition is satisfied, then output the NAME and SSN attributes to a RESULT relation.
  - The condition PASSENGER(x) simply binds the variable x to the relation PASSENGER.

- **Example 2**:
  - Query: "List names and ssn's of all passengers flying on Flight F338"
  - Solution:

```
{ x.NAME, x.SSN |
  PASSENGER(x) and
  ( (∃y) ( FLIGHT(y) and y.FLT_ID=x.FLT_ID
            and y.FLTNO=F338 )
  )
}
```

  - Explanation:
    Use variable x to scan tuples in PASSENGER. For each such tuple, use variable y to scan through FLIGHT and check for a match on FLT_ID. When such a match occurs, check FLTNO=F338.

- **Example 3**:
  - Query: “List names and ssn’s of all passengers flying into National airport”
  - Solution:

$$
\begin{array}{l}
\{\ x.\text{NAME},\ x.\text{SSN}\ |\\
\quad \text{PASSENGER}(x)\ \text{and}\\
\quad (\ (\exists y)\ (\ \text{FLIGHT}(y)\ \text{and}\ x.\text{FLT\_ID}=y.\text{FLT\_ID}\\
\qquad\qquad\quad \text{and}\ (\exists z)\ (\ \text{AIRPORT}(z)\ \text{and}\ z.\text{NAME}=\text{‘National’}\\
\qquad\qquad\qquad\qquad\qquad \text{and}\ z.\text{APT}=y.\text{ENDAPT})\\
\qquad\qquad)\\
\quad)\\
\}
\end{array}
$$

  - Explanation:
    Scan through PASSENGER (x) and match FLT_ID’s with FLIGHT (y). For each such match, scan through AIRPORT (z) and match ENDAPT of y with ‘National’.

2.7 Relational Algebra: Summary

- The tuple relational calculus is simply another formal mechanism for specifying queries.
- The domain relational calculus is similar (just slightly more cumbersome).
- The relational calculus (in any flavor) and the relational algebra are equivalent in terms of *expressive power*
  ⇒ Every query that can be expressed in one language can be expressed in the other.
  (Proof: see the book *Relational Database Theory* by P.Atzeni et al.)
- A query language is said to be *relationally complete* if it can express any query that can be expressed in relational algebra.
- Most commercial query languages (like SQL) are relationally complete.
- Are there simple queries expressible in English that are *not* expressible in relational calculus?
  - Consider the relation:

ANCESTOR	NAME	PARENT
	John, Jr.	Robert
	Robert	Rose
	John	Rose
	Ted	Rose
	⋮	⋮

  - By tracing back two parents, we see that "Rose" is an ancestor of "John, Jr.".
  - The query "Find the ultimate ancestor of John, Jr. (oldest in the chain of ancestors)" is NOT expressible in relational algebra.

  - Intuitively, there are a finite number of 'variables' in any relational calculus expression. By choosing an ancestral chain longer than this number, we will not be able to perform enough 'matches'.

- Usefulness of relational theory:
  - Neither the relational algebra nor the relational calculus are typically found in commercial dbases.
  - Then why study relational algebra? Several reasons:
    1. Commercial query languages like SQL are based on the relational algebra and relational calculus
    2. The query optimizer of a commercial language, typically translates a user-query into an internal form similar to relational algebra.
       ⇒ useful in query optimization.
    3. It is the first step in formalizing the theory of relations.
       ⇒ it will be useful later (*normalization* and schema design).
    4. It is compact and mathematically precise
       ⇒ leads to an accurate characterization of its expressive power.
       ⇒ many useful properties can be formally proven.

- Recall the hierarchical model from an earlier example:
  The Relational Model has become more popular because:
  - it is a flat model
  - operators on relations return relations and so can be combined easily
  - it is powerful enough to express several queries of interest
  - it is easy to work with in formal proofs

- What's missing?
  The relational algebra is only a set of operators to manipulate relations.
  ⇒ need to be able to insert values, print data etc.
  ⇒ commercial languages do that and more.

Chapter 3

SQL: A Query Language for Relational Databases

Course Notes on Database Systems

3.1 SQL: A Language for Relational Databases

- Formal query languagues: Relational algebra and relational calculus.
- Real-world query languages: SQL, QUEL, QBE (among others).
- Most commonly used: SQL (pronounced either “see-kwell” or “ess-cue-ell”)
- SQL: originally called SEQUEL (Structured English QUEry Language), 1974-1976 IBM.
- ANSI SQL standard, 1986
- SQL2 standard, 1992 – also called SQL-92. (current work on SQL3).
- SQL:
  - lets you create and delete relations;
  - lets you specify queries on existing relations;
  - lets you specify domain, key and foreign constraints;
  - has support for security, transaction management and remote access;
- We first consider queries. Then we will look at creating relations, and other issues such as constraints.

3.2 The Basic SQL Query

- What is an SQL ‘program’?
  - SQL code can be typed interactively into an interpreter.
  - SQL code can reside in text files and be compiled (as most high-level languages are).
  - SQL statements can be input from within other programming languages (like C).
- SQL terminology:

SQL	RELATIONAL ALGEBRA
table	relation
row	tuple
column	attribute

- The basic form of an SQL *query* statement:

```
select   [distinct] <attribute list>
from     <relation list>
[where   <condition>] ;
```

NOTE:

- The SQL keywords above are: **select**, **from**, **where** and **distinct**.
- **select** and **from** clauses are required.
- The **where** clause and **distinct** are optional.
- Other optional clauses can be added (such as **group by**). We will cover these later.

- An example:

select	NAME, SSN
from	PASSENGER
where	MILES > 50000;

NOTE:

- This expresses the query: “Find the names and SSN’s of all passengers who’ve accumulated more than 50000 miles”.
- In relational algebra:

$$\text{RESULT} := \Pi_{\text{NAME,SSN}} \; (\; \sigma_{\text{MILES}>50000} \; (\text{PASSENGER}) \;)$$

- The result is the relation (from the McVALUE Airlines dbase):

NAME	SSN
Tom	636-22-9999
Dick	223-63-7771

- In *contrast* to relational algebra, SQL’s **select** denotes the ‘project’ operator (Π) in relational algebra
  ⇒ it describes which attributes are desired in the result.
- The **where** clause corresponds to relational algebra’s ‘select’ (σ) operator.
- The **from** clause specifies the relations in the query.
- Interpretation:

select the attributes	NAME, SSN
from the relations	PASSENGER
where this condition holds:	MILES > 50000;

- Here, we are using boldface to denote SQL keywords.
- The queries are written consistent with standard style.
  ⇒ In practice, SQL statements can be written in different ways, e.g.,

```
select name, ssn from passenger where
   miles > 50000;
```

- Another example:
  - Query: “List names and ssn’s of all passengers flying on Flight F338”
  - In relational algebra:

    $$\text{RESULT} := \Pi_{\text{NAME,SSN}} \left(\sigma_{\text{FLTNO=F338}} \left(\text{PASSENGER} * \text{FLIGHT} \right) \right)$$

  - In SQL:

    ```
    select  NAME, SSN
    from    PASSENGER, FLIGHT
    where   PASSENGER.FLT_ID = FLIGHT.FLT_ID
            and FLIGHT.FLTNO = ‘F338’
    ```

    ⇒ This is how a join is done in SQL.
  - How to ‘read’ the SQL statement:
    - ∗ Note that the relational algebra query can be written as $\Pi_{\text{NAME,SSN}}(\sigma_{\text{FLTNO=F338}}(\sigma_{\text{PASSENGER.FLT\_ID=FLIGHT.FLT\_ID}}(\text{PASSENGER} \times \text{FLIGHT})))$.
    - ∗ Think of computing the cross-product of the relations in the **from** clause.
    - ∗ Then, apply the condition in the **where** clause to each tuple in the cross-product to select tuples.
    - ∗ Finally, project the attributes in the **select** clause to get the result relation.
    - ∗ Observe: *the join condition is explicitly specified in the* **where** *clause.*
    - ∗ NOTE: in an actual dbase, query optimization will try to prevent expensive cross-product computations.

- Duplicates:
  - While relational algebra forbids duplicate tuples, SQL allows them.
  - To force removal of duplicates: use **distinct**.
  - Example: "List all destination airports"

```
select   distinct END_APT
from     FLIGHT
```

  - Result:

END_APT
LGA
DCA
JFK

  - In contrast, the query

```
select   END_APT
from     FLIGHT
```

    will result in duplicates:

END_APT
LGA
DCA
JFK
DCA
JFK

3.3 SQL by Example

In examples below, we will use the McVALUE AIRLINES dbase.

PASSENGER	NAME	SSN	FLT_ID	MILES
	Bill	221-66-1234	17	2000
	Al	306-77-1131	63	45000
	Bob	111-22-3333	15	1600
	Jack	733-55-1122	15	7700
	Tom	636-22-9999	12	55000
	Trent	414-28-5850	11	200
	Dick	223-63-7771	17	64500
	Newt	828-81-6977	12	1570

FLIGHT	FLT_ID	FLT_NO	START_APT	END_APT
	11	F616	DCA	LGA
	12	F71	LGA	DCA
	15	F335	DCA	JFK
	17	F338	JFK	DCA
	63	F15	DCA	JFK

AIRPORT	APT	NAME	CITY
	DCA	National	Washington
	LGA	La Guardia	New York
	JFK	Kennedy	New York

CREW	SSN	FLT_ID
	011-44-2233	11
	313-62-7711	11
	442-11-3313	12
	722-55-1139	15
	011-44-2223	63
	011-44-2223	17
	313-62-7711	17
	442-11-3313	63

EMP	NAME	SSN	POSITION	SALARY	MGRSSN
	Erskine	011-44-2223	Co-Pilot	33,000	313-62-7711
	Hillary	313-62-7711	Pilot	39,000	334-56-9876
	Newt	442-11-3313	Steward	26,000	313-62-7711
	Donna	722-55-1139	Engineer	37,000	334-56-9876
	Scott	221-44-8883	Control Tower	29,000	722-55-1139
	Liz	119-72-3131	Sales	34,000	334-56-9876
	Morris	334-56-9876	CEO	42,960,000	334-56-9876

- **Example 1**.

  - Query: “List names and ssn’s of all passengers flying into National airport”

  - In relational algebra:

$$\begin{array}{l} \text{NAT} := \sigma_{\text{NAME}=\text{‘National’}}\ (\text{AIRPORT}) \\ \text{NAT\_FLTS} := \Pi_{\text{FLT\_ID}}\ (\ \text{FLIGHT} \bowtie_{\text{END\_APT}=\text{APT}}\ \text{NAT}\ ) \\ \text{RESULT} := \Pi_{\text{NAME,SSN}}\ (\text{PASSENGER} * \text{NAT\_FLTS}) \end{array}$$

  - In SQL:

```
select   PASSENGER.NAME, SSN
from     PASSENGER, FLIGHT, AIRPORT
where    AIRPORT.NAME='National'
         and AIRPORT.APT = FLIGHT.END_APT
         and FLIGHT.FLT_ID = PASSENGER.FLT_ID
```

  - Note the qualification of NAME in the **select** clause:

    PASSENGER.NAME

  - The following would be ambiguous:

```
select   NAME, SSN
from     PASSENGER, FLIGHT, AIRPORT
where    AIRPORT.NAME='National'
         and AIRPORT.APT = FLIGHT.END_APT
         and FLIGHT.FLT_ID = PASSENGER.FLT_ID
```

    NAME appears as an attribute in both PASSENGER and AIRPORT.

3.4 Aliases

- Aliases are permitted (and encouraged) in SQL.
- **Example 2**.
    - Consider the previous query: "List names and ssn's of all passengers flying into National airport"
    - The SQL for this query can be written as:

```
select  P.NAME, P.SSN
from    PASSENGER P, FLIGHT, AIRPORT
where   AIRPORT.NAME='National'
        and AIRPORT.APT = FLIGHT.END_APT
        and FLIGHT.FLT_ID = P.FLT_ID
```

    - Here, the relation name PASSENGER is *aliased* to P.
    - It is considered good style (for clarity and brevity) to alias *all* relations:

```
select  P.NAME, P.SSN
from    PASSENGER P, FLIGHT F, AIRPORT A
where   A.NAME='National'
        and A.APT = F.END_APT
        and F.FLT_ID = P.FLT_ID
```

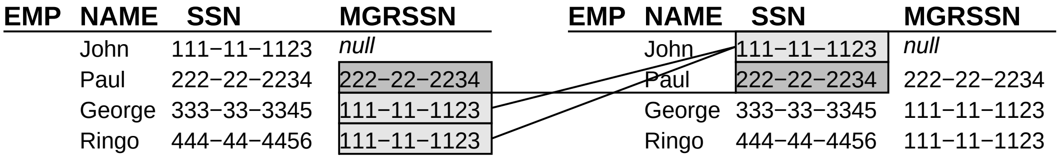
- Sometimes, aliasing is *necessary* even when using a single relation:
  - Example: consider the relation EMP:

EMP	NAME	SSN	MGRSSN
	John	111-11-1123	**null**
	Paul	222-22-2234	222-22-2234
	George	333-33-3345	111-11-1123
	Ringo	444-44-4456	111-11-1123

  - Query: “List employee names along with their manager’s names”
  - Solution:

```
select   E1.NAME, E2.NAME
from     EMP E1, EMP E2
where    E1.MGRSSN = E2.SSN
```

  - Analysis:



EMP	NAME	SSN	MGRSSN
	John	111−11−1123	*null*
	Paul	222−22−2234	222−22−2234
	George	333−33−3345	111−11−1123
	Ringo	444−44−4456	111−11−1123

EMP	NAME	SSN	MGRSSN
	John	111−11−1123	*null*
	Paul	222−22−2234	222−22−2234
	George	333−33−3345	111−11−1123
	Ringo	444−44−4456	111−11−1123

  - Result:

E1.NAME	E2.NAME
Paul	Paul
George	John
Ringo	John

- Renaming attributes in results:
  - Attributes can be renamed in results.
  - Example:
    Query: “List all destination airports”

```
select   distinct END_APT as DEST_AIRPORT
from     FLIGHT
```

3.5 Union, Intersection and Set Difference

- SQL has constructs for union (**union**), intersection (**intersect**) and set difference (**except**).
- **Example 3**.
  - Query: "List all flights (FLTNO) that have either a passenger named Newt or a crew member named Newt"
  - Solution:

```
( select  F.FLTNO
  from    PASSENGER P, FLIGHT F
  where   P.FLT_ID = F.FLT_ID
          and P.NAME = 'Newt' )
  union
( select  F.FLTNO
  from    EMP E, CREW C, FLIGHT F
  where   E.NAME = 'Newt'
          and E.SSN = C.SSN
          and C.FLT_ID = F.FLT_ID );
```

  - Here, we obtained the Newt-passengers:

```
( select  F.FLTNO
  from    PASSENGER P, FLIGHT F
  where   P.FLT_ID = F.FLT_ID
          and P.NAME = 'Newt' )
```

  and 'union-ed' it with the Newt-crewmembers:

```
( select  F.FLTNO
  from    EMP E, CREW C, FLIGHT F
  where   E.NAME = 'Newt'
          and E.SSN = C.SSN
          and C.FLT_ID = F.FLT_ID )
```

  NOTE: a join with FLIGHT was needed to get the FLTNO for each crewmember.

- **Example 4.**
  - Query: "List names and ssn's of all employees not on any flight".
  - First, let's obtain ssn's of all nonflying employees:

```
( select   E.SSN
  from     EMP E )
  except
( select   C.SSN
  from     CREW C );
```

  - This can be refined to solve the original query: (to include names):

```
( select   E.NAME, E.SSN
  from     EMP E )
  except
( select   E2.NAME, C.SSN
  from     CREW C, EMP E2
  where    C.SSN = E2.SSN );
```

- **intersect** (intersection) is similar to **union** and **except**, e.g.,
  - Query: "List all flights (FLTNO) that have both a passenger named Newt *and* a crew member named Newt"
  - Solution:

```
( select      F.FLTNO
  from        PASSENGER P, FLIGHT F
  where       P.FLT_ID = F.FLT_ID
              and P.NAME = 'Newt' )
  intersect
( select      F.FLTNO
  from        EMP E, CREW C, FLIGHT F
  where       E.NAME = 'Newt'
              and E.SSN = C.SSN
              and C.FLT_ID = F.FLT_ID );
```

3.6 Obtaining All Tuples or All Attributes

- To simply obtain *all* tuples in a relation, omit the **where** clause.
- **Example 5**.
  - Query: "Obtain a list of employee names"
  - Solution:

```
select  E.NAME
from    EMP E;
```

- To obtain *all* attributes, use ∗.
- **Example 6**.
  - Query: "Obtain all information about employees"
  - Solution:

```
select  ∗
from    EMP;
```

- **Example 7**.
  - Query: "Obtain all information about airports in New York"
  - Solution:

```
select  ∗
from    AIRPORT A
where   A.CITY = 'New York';
```

3.7 Nested Queries

- SQL allows you to nest queries.
- Most common ways to nest queries involve the keywords: **in**, **not in**, **exists**, **not exists**.
- **Example 8**.
  - Query: "List names and ssn's of all passengers flying on Flight F338"
  - Previous solution:

    ```
    select  P.NAME, P.SSN
    from    PASSENGER P, FLIGHT F
    where   P.FLT_ID = F.FLT_ID
            and F.FLTNO = 'F338'
    ```

  - Solution using nested queries:

    ```
    select  P.NAME, P.SSN
    from    PASSENGER P
    where   P.FLT_ID in      (  select  F.FLT_ID
                                from    FLIGHT F
                                where   F.FLTNO = 'F338' );
    ```

  - Explanation: think of a tuple-scanning variable for each of the **select**'s. For each tuple in the outer select, the entire inner relation is created and scanned.
    ⇒ for each tuple in PASSENGER, compute the 'F338' FLT_ID's from FLIGHT and check containment.
  - If it seems like a waste to re-compute the inner **select** here, it is.
    ⇒ Previous solution is better.

- **Example 9**.
  - Query: "List flights mistakenly assigned to airports *not* served by McVALUE".
  - Solution (for destination airports):

```
select   F.FLTNO
from     FLIGHT F
where    F.END_APT not in   (  select   A.APT
                               from     AIRPORT A ) ;
```

  - Note that

```
select   F.FLTNO
from     FLIGHT F, AIRPORT A
where    F.END_APT <> A.APT;
```

  does not work.

- **Example 10**.
  - Query: "List flights assigned to valid destination airports served by McVALUE".
  - Solution (for destination airports):

```
select   F.FLTNO
from     FLIGHT F
where    exists        (  select   A.APT
                          from     AIRPORT A
                          where    A.APT = F.END_APT);
```

  - In using **exists**, the inner **select** is computed to see if it produces a *non-empty* relation.
  - Note that each outer tuple is checked against each inner tuple.

- Similarly, **not exists** returns true if the nested query result is *empty*.

- **Example 11**.

  - Query: "List names of passengers who fly into all airports".
  - Solution:

```
select   P.NAME
from     PASSENGER P
where    not exists        ((  select   A.APT
                               from     AIRPORT A )
                               except
                            (  select   F.END_APT as APT
                               from     FLIGHT F, PASSENGER P2
                               where    P2.SSN = P.SSN
                                        and F.FLT_ID = P2.FLT_ID ));
```

  - Explanation:
    - The first nested query simply obtains a list of all airports:

```
( select   A.APT
  from     AIRPORT A )
```

    - The outer query scans through PASSENGER.
    - The second nested query does a join between PASSENGER and FLIGHT to obtain each passenger's airports.
    - The passenger in each tuple created in the second nested query is matched (via P2.SSN=P.SSN) with the outer passenger scan.
    - Now, if the list of airports is not complete then the *set-difference* will give a *non-empty* result.
    - The non-empty result is checked using **not exists**.

3.8 Aggregate Functions

- Consider the query: “Count the number of employees”.
  - Relational algebra has no mechanism for counting the number of tuples.
  - Some books define an *extended relational algebra* that permits counting, computing sums and maximums etc,
    ⇒ these are called *aggregate functions*.
- Aggregate functions in SQL.
  - SQL supports five aggregate operators:
    1. **max** (<ATTR>)
       ⇒ the maximum value in column ATTR
    2. **min** (<ATTR>)
       ⇒ the minimum value in column ATTR
    3. **sum** ([**distinct**] ATTR)
       ⇒ the sum of all (unique, if **distinct** is specified) values in column ATTR
    4. **avg** ([**distinct**] ATTR)
       ⇒ the average of all (unique, if **distinct** is specified) values in column ATTR
    5. (a) **count** ([**distinct**] ATTR)
       ⇒ the number of (unique, if **distinct** is specified) non-null values in column ATTR
       (b) **count** ($*$)
       ⇒ the number of tuples

- **Example 12**.
  - Query: “Count the number of employees”
  - Solution:

```
select   count(∗)
from     EMP;
```

- **Example 13**.
  - Query: “Count the number of passengers with mileage > 50000”
  - Solution:

```
select   count(∗)
from     PASSENGER P
where    P.MILES > 50000;
```

- **Example 14**.
  - Query: “Find the average salary of employees that fly.”
  - Solution:

```
select   avg (E.SALARY)
from     EMP E
where    E.SSN in          (  select   C.SSN
                              from     CREW C );
```

- NOTE: aggregate functions cannot be used with other attributes in the same **select** clause.
  - For example, the following is illegal:

```
select   E.NAME, min (E.SALARY)
from     EMP E
```

  (because it does not make sense).

  - One exception: aggregates *can* be used with other attributes when the query contains the **group by** clause.

- **Example 15**.
  - Query: "Find the passenger with the most mileage."
  - Solution:

```
select  P.NAME
from    PASSENGER P
where   P.MILES =        (  select  max(P2.MILES)
                            from    PASSENGER P2 );
```

  - Note that the following is illegal:

```
select  P.NAME, max(P.MILES)
from    PASSENGER P;
```

  - The keyword **all** is sometimes used:

```
select  P.NAME
from    PASSENGER P
where   P.MILES >=  all  (  select  P2.MILES
                            from    PASSENGER P2 );
```

3.9 The group by Clause

- An aggregate function is computed on all tuples in a relation. For example:

  ```
  select  avg(E.SALARY)
  from    EMP E
  ```

  computes the average salary of all employees.

- We can obtain the average salary of a particular department using the **where** clause:

  ```
  select  avg(E.SALARY)
  from    EMP E
  where   E.DEPT='crew'
  ```

  In this case, the average salary of crew members is reported.

- What if we want the average salary of each department?

  One option: write a separate SQL statement for each department.

- Using the **group by** clause:

  ```
  select    avg(E.SALARY)
  from      EMP E
  group by  E.DEPT
  ```

  This produces a list of average salaries of the departments.

  However, only the averages are reported
  ⇒ cannot match with department names!

- To match with department names:

  ```
  select    E.DEPT, avg(E.SALARY)
  from      EMP E
  group by  E.DEPT
  ```

  Here, an attribute is allowed in the **select** clause along with an aggregate operator.

  Rule: attributes that appear in a **group by** clause may appear in the **select** clause.

- Sometimes we need to specify conditions on groups as a whole.

  For example, consider this query: “Find the average salaries of those departments with at least 4 employees”.

  The **having** clause can be used to achieve this result:

  ```
  select    E.DEPT, avg(E.SALARY)
  from      EMP E
  group by  E.DEPT
  having    count (*) > 3
  ```

- Consider a more complex query: “List employees in departments with average salaries larger than 50,000”

  Solution:

  ```
  select  E.NAME
  from    EMP E
  where   E.DEPT in (  select    E2.DEPT
                       from      EMP E2
                       group by  E2.DEPT
                       having    avg(E2.SALARY) > 50000)
  ```

- Sometimes we want to join attributes from other relations and want them to appear along with the aggregate.

  Solution: include those attributes in the **group by** clause.

Example: "List departments with average salaries larger than 50,000 along with department names".

Solution:

```
select     E.DEPTNO, D.DNAME, avg(SALARY)
from       EMP E, DEPT D
where      E.DEPTNO = D.DEPTNO
group by   E.DEPTNO, D.DNAME
having     avg(SALARY) > 50000;
```

Note that the attributes that appear in the output also need to appear in the **group by** clause.

3.10 More on SQL

- Pattern search: to perform wildcard searches, use the percentage symbol and the **like** keyword, e.g.,

```
select   *
from     EMP E
where    E.NAME like 'Sm%';
```

- A list of items can be specified using the **in** keyword, e.g.,

```
select   *
from     EMP E
where    E.NAME in ('Smith','Jones');
```

- Output can be sorted by one or more attributes using the **order by** clause, e.g.,

```
select     *
from       EMP E
order by   E.NAME, E.FNAME
```

- Comments in SQL are specified by using two dashes. Everything after the two-dash symbol up to the end of the line is a comment, e.g.,

```
-- File: test1.sql
-- This file prints out names and SSN's of employees in Sales
select E.NAME, E.SSN     -- Name, ssn attributes
from EMP E               -- EMP is aliased to E
where E.DEPT='Sales';    -- Specify department as Sales
```

3.11 Null Values in SQL

- SQL allows **null**'s as values.
- SQL has a special comparison operator for **null**'s: the **is null** operator.
- **Example 16**.
  - Query: "List all the tuples in EMP whose salary field is **null**"
  - Solution:

```
select   *
from     EMP E
where    E.SALARY is null;
```

- How are comparisons made with a **null** field?
  - Example:

```
select   E.NAME
from     EMP E
where    E.SALARY < 1000
         or E.SALARY > (  select   E2.SALARY
                          from     EMP E2
                          where    E2.NAME = 'Smith' );
```

  - What happens if Smith's salary is **null**?
    ⇒ What does E.SALARY > **null** evaluate to?
- Comparison rule for **null** values:
  - For any value x, and operator $<op> \in \{=, <, >, \leq, \geq, +, -, /, *\}$: the expression

    $$x <op> \textbf{null}$$

    evaluates to the value *unknown* (not an SQL keyword).
  - For example, 5.67 ≤**null** evaluates to *unknown*.

- Logic rules for *unknown* values:
  - Let T denote *true*, F denote *false* and U denote *unknown*.
  - The rule (truth-table) for NOT is:

x	NOT(x)
T	F
F	T
U	U

  - The truth table for OR is:

x	y	x OR y
T	T	T
T	F	T
F	T	T
F	F	F
T	U	T
F	U	U
U	T	T
U	F	U
U	U	U

  - The truth table for AND is:

x	y	x AND y
T	T	T
T	F	F
F	T	F
F	F	F
T	U	U
F	U	F
U	T	U
U	F	F
U	U	U

  - Thus, for example, ((T AND U) OR T) evaluates to *true*.

- How **null**'s and *unknown*'s are treated in SQL:
  - The **where** clause must evaluate to *true* for a tuple to be included in the result.
  - Example: Consider the following relation:

AIRPORT	APT	NAME	CITY
	DCA	National	Washington
	LGA	La Guardia	New York
	JFK	Kennedy	New York
	null	Heathrow	London
	null	Gatwick	London

The SQL statement:

```
select   A.APT, A.NAME
from     AIRPORT A
where    A.CITY=‘London’ or A.APT=‘LON’;
```

produces the result

APT	NAME
null	Heathrow
null	Gatwick

whereas the SQL statement

```
select   A.APT, A.NAME
from     AIRPORT A
where    A.CITY=‘London’ and A.APT=‘LON’;
```

produces the empty relation.

- What about duplicates with **null**'s? Consider

```
select   A.APT, A.CITY
from     AIRPORT A
where    A.CITY='London';
```

This produces:

APT	CITY
null	London
null	London

whereas

```
select   distinct A.APT, A.CITY
from     AIRPORT A
where    A.CITY='London';
```

produces

APT	CITY
null	London

NOTE: the tuples are considered equal even though equality between **null**'s evaluates to *unknown*.

3.12 Creating Relations

- Relations (tables) are created in SQL using the **create table** statement.
- Example:

```
create table PASSENGER  (  NAME     char(20),
                           SSN      char(20),
                           FLT_ID   integer,
                           MILES    integer );
```

- Example:

```
create table FLIGHT  (  FLT_ID      integer,
                        FLTNO       char(8),
                        START_APT   char(3),
                        END_APT     char(3) );
```

- Example:

```
create table AIRPORT  (  APT    char(3),
                         NAME   char(10),
                         CITY   char(15) );
```

- Tables can be delete using the **drop table** statement, e.g.,

```
drop table FLIGHT
```

3.13 Specifying Constraints

- Domain constraints are specified by using a **check** clause in the **create table** statement, e.g.,

```
create table PASSENGER  (  NAME             char(20),
                           SSN              char(20),
                           FLT_ID           integer,
                           MILES            integer,
                           check (MILES > 0) );
```

- Primary keys can be specified by the keywords **primary key** in the **create table** statement:

```
create table PASSENGER  (  NAME                 char(20),
                           SSN                  char(20),
                           FLT_ID               integer,
                           MILES                integer,
                           primary key (SSN) );
create table AIRPORT    (  APT                  char(3),
                           NAME                 char(10),
                           CITY                 char(15),
                           primary key (APT, NAME) );
```

Recall: primary keys cannot have **null** values.

- Specific fields can be prohibited from allowing **null**'s.
- Specific fields can be prohibited from allowing duplicate values.
- Example:

```
create table PASSENGER  (  NAME              char(20) not null,
                           SSN               char(20),
                           FLT_ID            integer,
                           MILES             integer,
                           primary key (SSN) );
```

- Example:

```
create table FLIGHT  (  FLT_ID            integer,
                        FLTNO             char(8),
                        START_APT         char(3),
                        END_APT           char(3),
                        unique (FLT_ID) );
```

- A foreign key can be specified, e.g.,

```
create table PASSENGER  (  NAME                              char(20),
                           SSN                               char(20),
                           FLT_ID                            integer,
                           MILES                             integer,
                           primary key (SSN),
                           foreign key (FLT_ID)
                           references FLIGHT (FLT_ID) );
```

3.14 Insertion, Deletion and Modification

- A tuple can be inserted into a relation using the **insert into** statement:

```
insert into   <relation name>
values        <attribute values>
```

- Example:

```
insert into   FLIGHT
values        (23, 'F723', 'DCA', 'OHA');
```

- Example:

```
insert into   FLIGHT
values        (45, 'F414', 'OHA', 'DCA');
```

- Insertions can be done using the output from **select**, e.g.,
- Example:

```
create table DC_AIRPORTS   (   APT     char(3),
                               NAME    char(10) );


insert into DC_AIRPORTS   select   A.APT, A.NAME
                          from     AIRPORT A
                          where    A.CITY='Washington';
```

- Deletions are specified using the **delete from** statement:

```
delete from  <relation name>
where        <condition>
```

- Example:

```
delete from  PASSENGER
where        MILES < 5;
```

- Example:

```
delete from  PASSENGER
where        NAME is null;
```

- Example:

```
delete from  PASSENGER
where        FLT_ID not in  (  select  F.FLT_ID
                               from    FLIGHT F );
```

- Individual or groups of tuples can be modified using the **update** statement:

```
update  <relation name>
set     <modification>
where   <selection condition>
```

- Example:

```
update  PASSENGER P
set     P.SSN=’828-81-6975’
where   P.SSN=’828-81-6977’ and P.NAME=’Newt’;
```

- Example:

```
update  PASSENGER P
set     P.MILES = P.MILES + 100
where   P.FLT_ID = 12;
```

3.15 Insertions, Modifications and Constraints

- Suppose we define a primary key on PASSENGER:

```
create table PASSENGER  (  NAME                   char(20),
                           SSN                    char(20),
                           FLT_ID                 integer,
                           MILES                  integer,
                           primary key (SSN) );
```

Then, the following SQL statements result in error:

- Example using insert:

```
insert into   PASSENGER
values        ('Jim', '221-66-1234', 63, 100);
```

⇒ can't allow duplicate primary key values.

- Example using update:

```
update   PASSENGER P
set      P.SSN=null
where    P.NAME='Newt';
```

⇒ can't allow **null**'s in primary key attributes.

These inserts and updates are *rejected*.

- Consider the following foreign key constraint:

```
create table PASSENGER   (  NAME                       char(20),
                            SSN                        char(20),
                            FLT_ID                     integer,
                            MILES                      integer,
                            primary key (SSN),
                            foreign key (FLT_ID)
                            references FLIGHT (FLT_ID) );
```

  Then, the following changes violate the constraint:

  – Example with insert:

```
insert into  PASSENGER
values       ('Jim', '455-98-0101', 9, 200)
```

  ⇒ FLT_ID=9 does not exist in FLIGHT
  ⇒ *reject* insertion.

  – Example with delete:

```
delete from  FLIGHT F
where        F.FLT_ID = 11
```

  ⇒ All FLT_ID=11 values in PASSENGER point to nothing
  ⇒ reject deletion?

- When a foreign key is deleted there are several options that can be specified in the **create table** statement:

  1. Delete all tuples that refer to the value
     ⇒ delete all PASSENGER tuples with FLT_ID=11.

     ```
     create table PASSENGER  (  NAME                            char(20),
                                SSN                             char(20),
                                FLT_ID                          integer,
                                MILES                           integer,
                                primary key (SSN),
                                foreign key (FLT_ID)
                                references FLIGHT (FLT_ID) )
                                on delete cascade );
     ```

  2. Don't allow the deletion of the foreign key
     ⇒ reject above deletion.

     ```
     create table PASSENGER  (  NAME                            char(20),
                                SSN                             char(20),
                                FLT_ID                          integer,
                                MILES                           integer,
                                primary key (SSN),
                                foreign key (FLT_ID)
                                references FLIGHT (FLT_ID) );
     ```

3. Set reference to **null**
   ⇒ set FLT_ID to **null** in PASSENGER.

```
create table PASSENGER   (  NAME                          char(20),
                            SSN                           char(20),
                            FLT_ID                        integer,
                            MILES                         integer,
                            primary key (SSN),
                            foreign key (FLT_ID)
                            references FLIGHT (FLT_ID)
                            on delete set null );
```

4. Set reference to a default value
   ⇒ set FLT_ID to some default value in PASSENGER.

```
create table PASSENGER   (  NAME                          char(20),
                            SSN                           char(20),
                            FLT_ID                        integer default 11,
                            MILES                         integer,
                            primary key (SSN),
                            foreign key (FLT_ID)
                            references FLIGHT (FLT_ID)
                            on delete set default );
```

- Relations can be deleted with constraint specifications, e.g.,

  - **drop table** FLIGHT **cascade**
    – will delete the FLIGHT relation.
    – all references to tuples in FLIGHT will be modified to maintain referential integrity.

  - **drop table** FLIGHT **restrict**
    – will be dropped only if not referenced.

- Attributes can be added or removed from relations, e.g.,

  - **alter table** EMP **add** BOSS_NAME char(15);
    – adds attribute BOSS_NAME, a string of max 15 chars.

  - **alter table** FLIGHT **drop** FLT_ID **cascade**
    – attribute FLT_ID is removed from FLIGHT.
    – references are removed.

- Constraints can be named (and later, dropped), e.g.,

```
create table PASSENGER  (  NAME                  char(20),
                           SSN                   char(20),
                           FLT_ID                integer default 11,
                           MILES                 integer,
                           primary key (SSN),
                           constraint FLIGHT_ID
                           foreign key (FLT_ID)
                           references FLIGHT (FLT_ID)
                           on delete set default );
```

⋮

```
alter table        PASSENGER
drop constraint    FLIGHT_ID
```

3.16 Security

- The database administrator can grant access privileges to users and dbase programmers using the **grant** statement, e.g.,
  - **grant insert, delete on** PASSENGER **to** SIMHA
  - **grant select on** PASSENGER **to** SIMHA
  - **grant insert on** PASSENGER **to** SIMHA **with grant option**
- Privileges can be revoked using the **revoke** statement.

3.17 Views in SQL

- Suppose we have a relation called EMPLOYEE with attributes:

  EMPLOYEE (NAME, SSN, ADDR, SALARY)

  – The database administrator does not want programmers to have access to SALARY information.
  – One option: create two relations:

    EMP (NAME, SSN, ADDR)
    EMP_SAL (SSN, SALARY)

    And allow programmers access only to EMP.
  – Note: space is wasted.

- Most DBMS's allow *views* or *virtual relations* to be created.
  – In the above example, a virtual relation (or view) such as

    EMP (NAME, SSN, ADDR)

    could be created.
  – Programmers will go ahead and program assuming that a relation called EMP (with attributes NAME, SSN and ADDR) exists.
  – System will take commands given on EMP and convert them to commands on EMPLOYEE.

- Creating a view in SQL (usually done by database administrator):

```
create  view    EMP
as      select  NAME, SSN, ADDR
        from    EMPLOYEE;
```

Later on, a programmer can treat EMP like any other relation:

```
select   NAME, DNAME
from     EMP, DEPARTMENT
where    DNO=DNUMBER
```

- Although a programmer may treat a *view* as any other relation, updates are sometimes restricted because of potential ambiguity.

  Suppose we have 3 tables:

  EMPLOYEE (NAME, SSN)
  WORKS_ON (SSN, PNO)
  PROD_NAME (PNO, PRODUCT)

  For example:

EMPLOYEE	NAME	SSN
	Smith	123456789
	⋮	⋮

WORKS_ON	SSN	PNO
	123456789	5
	⋮	⋮

PROD_NAME	PNO	PRODUCT
	⋮	⋮
	4	Coke
	5	Pepsi
	⋮	⋮

Next, suppose we create a view: EMP_PRODUCT (NAME, PRODUCT) (employee name and the product s/he is working on). In SQL:

```
create  view    EMP_PRODUCT (NAME, PRODUCT)
as      select  NAME, PRODUCT
        from    EMPLOYEE, WORKS_ON, PROD_NAME
        where   EMPLOYEE.SSN=WORKS_ON.SSN
                and WORKS_ON.PNO=PROD_NAME.PNO
```

Assume that the programmer only sees the relation EMP_PRODUCT

From the table, we see that *Smith* works on product *Pepsi*. Suppose now, Smith is asked to work on *Coke*. To update, the programmer does:

```
update  EMP_PRODUCT
set     PRODUCT='Coke'
where   NAME='Smith'
```

To realize this modification, the system must actually update the relations that constitute the *view* EMP_PRODUCT.

There are two ways of achieving this change:

1. Change PNO to 4 in WORKS_ON.
2. Change PRODUCT to 'Coke' for PNO=5 in PROD_NAME.

– Ambiguity!

Note: here the intention is clear, we should change PNO to 4 in WORKS_ON. But in other cases, it is not so obvious.

3.18 Embedded SQL and SQL Libraries

- SQL is only a language for manipulating relations.
- Most real applications require other features, e.g., writing to files, windows, adding report-quality word-processing etc.
- Commercial DBMS's often provide their own languages. For example, Oracle has its own language.
  - Real world applications are written in these languages.
  - These languages provide a mechanism for *embedding* SQL (to manipulate relations).
- Dbase vendors often provide a mechanism to embed SQL in programming languages such as C. For example,

```
struct emp_struct {
 char name[20];
 int  ssn;
 int  dept_no;
}
struct emp_struct *emp_ptr;
.
.
.
EXEC SQL
  select NAME, SSN, DNO from EMPLOYEE where NAME=‘Smith’
END EXEC SQL;
/* Returns a tuple into struct pointed to by emp_ptr */
```

- In C-embedded SQL, the dbase vendor provides a C compiler that translates the embedded SQL into appropriate function calls to a relational

operator library.
⇒ some query optimization can be done at compile time.

- Often, it is desirable to make direct SQL calls from a C program, e.g.,

```
struct emp_struct {
 char name[20];
 int  ssn;
 int  dept_no;
}
struct emp_struct *emp_ptr;
.
.
.
call_sql_interp (emp_ptr, ’select NAME, SSN, DNO from EMPLOYEE
                 where NAME=‘Smith’’);
/* Returns a tuple into struct pointed to by emp_ptr */
```

⇒ an SQL library is provided (by the dbase vendor).

3.19 SQL: Summary

- SQL is a language for manipulating relations.
- SQL is based on relational algebra operators, adding ‘syntactic sugar’ where needed.
- SQL is itself usually executed from some other language (via embedding).
- SQL provides:
  - *Data definition capabilities* – commands for defining schemas, relations, integrity constraints, default specifications.
  - *Data manipulation capabilities* – commands for updating and manipulating relations (e.g. **select** command).
  - *View capabilities* – commands for creating views and the support software to allow programming based on views.
  - *Authorization* – commands for controlling access.
  - *Integrity checking* – mechanisms for automatic integrity checking.
  - *Transaction control* – (to be explained later).

Chapter 4

Physical Implementation of Databases

Course Notes on Database Systems

4.1 Introduction

- Consider the following relations and query:
  - Relations FLIGHT and AIRPORT:

FLIGHT	FLT_ID	FLT_NO	START_APT	END_APT
	11	F616	DCA	LGA
	12	F71	LGA	DCA
	15	F335	DCA	JFK
	17	F338	JFK	DCA
	62	F15	DCA	JFK

AIRPORT	APT	NAME	CITY
	DCA	National	Washington
	LGA	La Guardia	New York
	JFK	Kennedy	New York

  - SQL query:

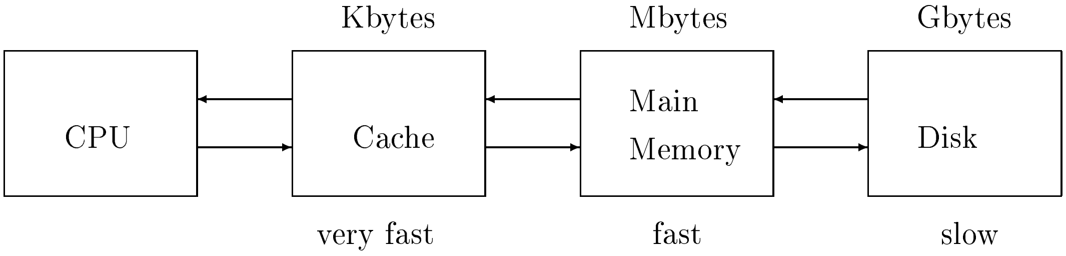
```
select   F.FLTNO
from     FLIGHT F, AIRPORT A
where    F.END_APT = A.APT
         and A.CITY = 'New York';
```

- Several implementation questions arise:
  - How is the query actually computed?
  - How and where are relations stored?

- A simple (but naive) answer:
  - Store each relation in a (Unix) file.
    ⇒ tuples are stored in order of insertion
  - Write C code to perform basic relational algebra operations.
  - For example, to compute Π_{APT} (AIRPORT) :
    - read from file `airport` containing tuples of AIRPORT.
    - scan through tuples and extract APT field to create result tuples.
    - write RESULT tuples to file `result`.
- Why is this naive?
  - The above method will work but will not be efficient for large data sets.
  - Large data sets imply many disk accesses
    ⇒ disk accesses are time-consuming
    ⇒ should try to minimize disk access
    ⇒ need to understand low-level storage details

4.2 Hardware Review

Basic system architecture:



Cache memory:

- Fast access times (approx 50 nanoseconds per access)
- Expensive
- Little or no software control over cache contents
- Small (KBytes), not large enough for DBMS data
- Volatile

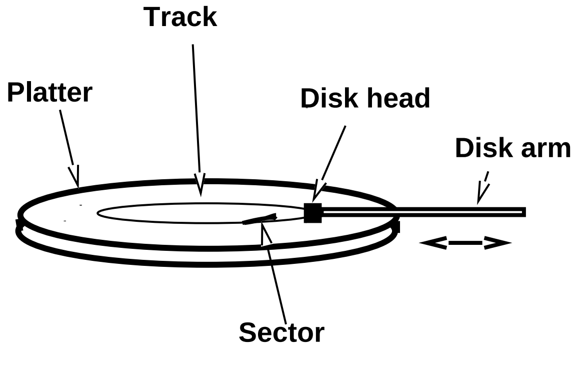
Main memory:

- Moderately fast (100-500 nanoseconds per access)
- Relatively inexpensive
- Plenty of software control (via OS)
- Large enough (MBytes) for code, but not all data
- Volatile

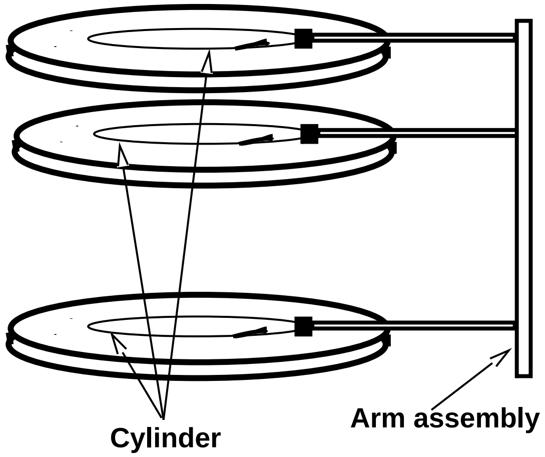
Disk:

- Slow (10-100 milliseconds per access)
- Inexpensive
- Software control available (via disk controller)
- Large data storage (GBytes)
- Nonvolatile
  ⇒ since data must live beyond execution time of dbase tools, data must be stored on disk.

4.3 A Typical Disk or Disk System



SINGLE DISK

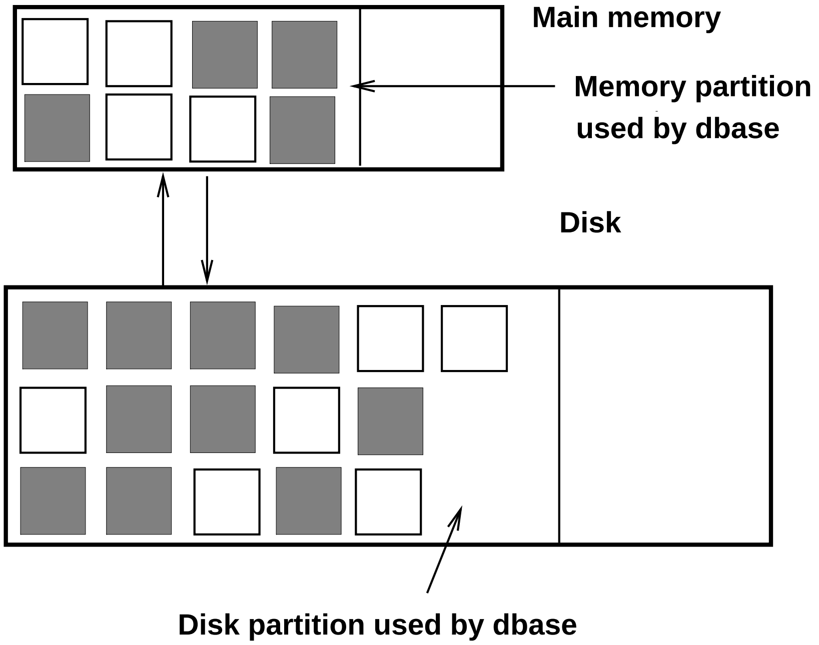


DISK SYSTEM

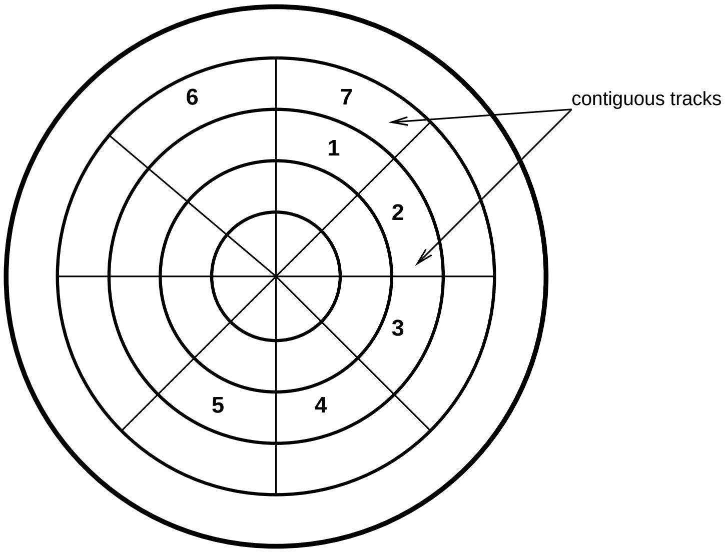
- Read/write head moves across magnetic surface.
- Discrete number of tracks (cylinders).
  Seek time is the time for the head to move to the desired track.
- Each track is divided into sectors.
- Platter rotates at high speed.
  Latency is the time required for the head to be positioned over the desired sector.
- A *block* is the unit of access for a disk
  ⇒ always read or write an *entire* block.
- Blocks are also called disk pages.
- Block sizes are usually programmable.
- Blocks have id's (block addresses).

- Disk I/O driver can translate between block addresses and the appropriate <track,sector> combination.
- Why is a disk access slow compared to a main memory access?
  ⇒ seek time + latency + block transfer time
- Incorporating tape storage:
  - Sometimes, disks are too small (e.g., scientific data)
    ⇒ use mag tapes
  - Tapes are cheap and reliable, but *very slow*.
  - Tapes may require manual handling.
  - Tapes are used primarily for backups.

4.4 Interaction Between Disk and Memory



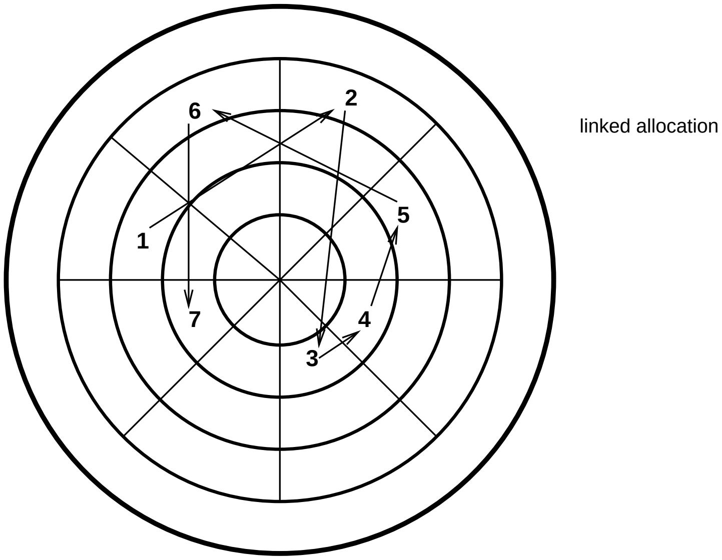
- Data is stored on disks in fixed-size blocks.
- The *disk manager* manages the allocation of blocks on disk.
  - Again, it is possible to use the disk I/O driver in the OS.
  - Most DBMS's prefer to manage allocation on disk.
  - It is preferable to allocate contiguous blocks for some files.
    ⇒ more efficient access is possible
  - The disk manager keeps track of free blocks on disk.
- Methods of storing disk blocks:
  Consider a file of n blocks that is to be written to disk.
  Contiguous allocation:
  - Always store blocks of a file contiguously (in adjacent tracks).



- Advantages: Smaller seek and latency.
- Disadvantages: Difficult to expand file. May not be able store some large files.

Linked allocation:

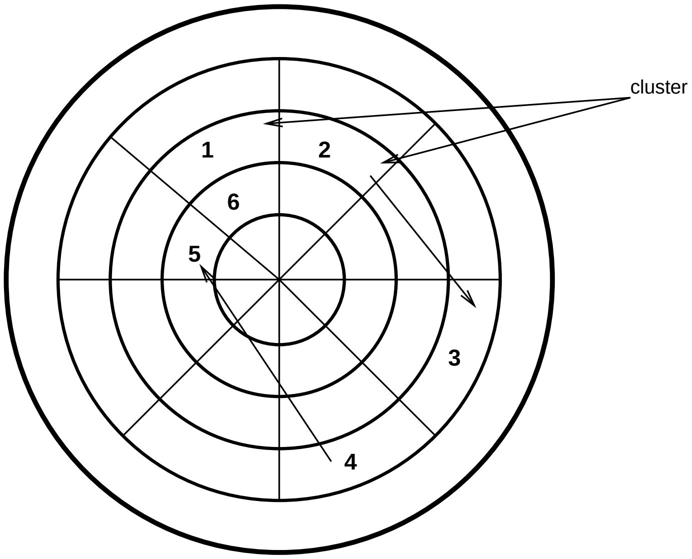
- Store blocks anywhere, but link them (keep pointer to next block in current block).



- Advantages: Simple. Uses disk space efficiently. Easy to expand and contract files.
- Disadvantage: Lot of seek time incurred if blocks are all over the place.

Clustered allocation:

- A compromise: group blocks into clusters and link them.
  ⇒ clustered blocks are stored contiguously, but clusters can be anywhere on disk.



A cluster of blocks is sometimes called an *extent*.

- For dbase programs to work on the data, data has to be brought into main memory.
  - Blocks are read into main memory from disk.
  - Blocks are written to disk from main memory.
  - One option is to use the OS memory management system (virtual memory).

- Most often, dbases do their own memory management.
  ⇒ a large buffer in main memory is acquired from the OS and used for keeping blocks in memory.

- The *buffer manager* is a program that manages memory for the dbase.
  - The buffer manager is defined by its *replacement* policy.
    ⇒ when a block is brought into a full memory, some block in memory has to be written out
  - Standard OS replacement policy: Least Recently Used (LRU) (or some variation of LRU).
  - Dbase considerations are often different:
    - ∗ A DBMS often needs to *pin* some important, frequently accessed blocks (such as directory blocks).
    - ∗ Query analysis and optimization permits some predictability in disk accesses.
      ⇒ *prefetching* can be done (overlap seeks with computation).
    - ∗ A DBMS needs to force writing a block to disk (for persistence).
  - When a particular block is requested (by a higher-level program):
    - ∗ If the block is already in main memory, return the start address of the block.
    - ∗ Otherwise, if memory is full, select a block for replacement and write it to disk.
    - ∗ Bring requested block into memory and return start address.
  - Sometimes, in client-server mode, a block must also be transferred to a client.
  - The buffer manager keeps track of whether blocks have been written into (using a *dirty bit*).
  - When concurrency is permitted, the buffer manager allows blocks to be locked and unlocked.

4.5 Storing Relations

- A DBMS typically runs its own file system.
- Most often, all tuples in a relation are stored in a file.
- Conceptually, a file is a collection of tuples.
- Physically, a file is a collection of disk blocks
  ⇒ a disk block may contain one or more tuples.
- Sometimes, a large tuple may span multiple blocks.
- At the physical-level, a tuple is sometimes called a *record*.
- Two types of records:
  - *Fixed-size records*
  - *Variable-size records*.
- Consider the following relation:

```
create table PASSENGER  (  NAME    char(6),
                           SSN     char(11),
                           FLT_ID  integer,
                           MILES   integer );
```

Assume integers are 32-bits (4 bytes) long
⇒ 25 bytes are required for each tuple.
⇒ 2 records/block if blocksize is 64 bytes.

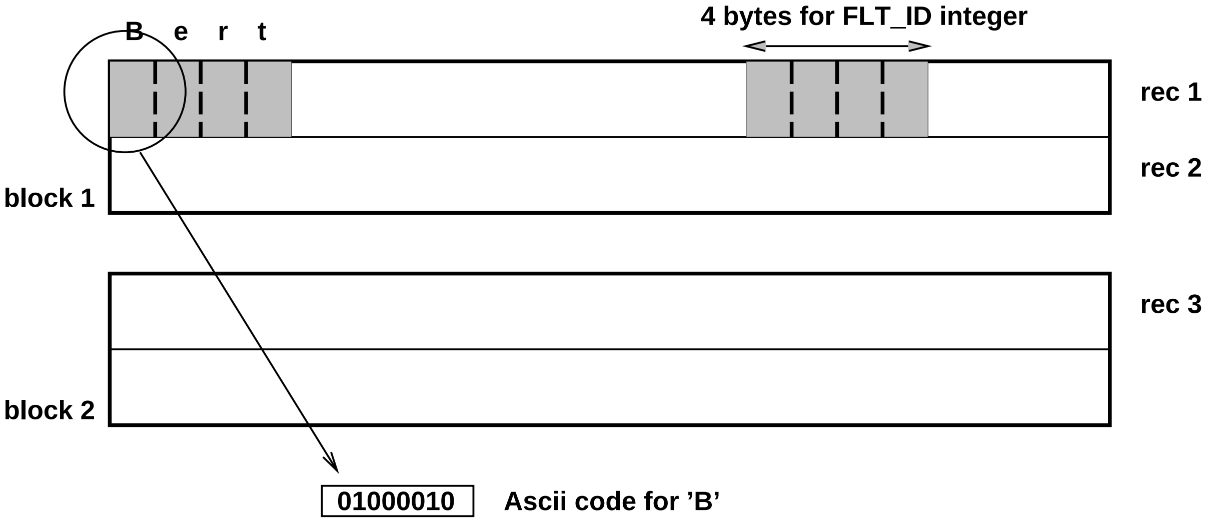
- Suppose we have the following data (high-level conceptual view):

PASSENGER	NAME	SSN	FLT_ID	MILES
	Bert	111-22-3333	34	550
	Ernie	222-33-4444	56	750
	Kermit	444-55-6666	78	950

Low-level conceptual view:

```
<’Bert’, ’111-22-3333’, 34, 550>
<’Ernie’, ’222-33-4444’, 56, 750>
<’Kermit’, ’444-55-6666’, 78, 950>
```

Physical view:

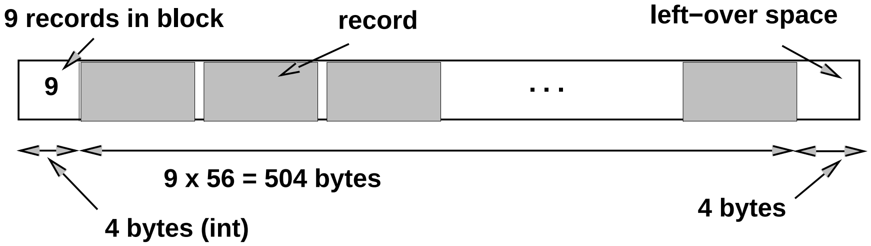


4.6 Fixed-Size Records

- Retrieving a field from within a fixed-size record:
  - Use the offset of the field within the tuple.
  - In the above example, the SSN field's offset is 20 bytes.
- Storage of fixed-size records in blocks.
  - For example, suppose

$$\begin{aligned} \text{record size} &= 56 \text{ bytes} \\ \text{block size} &= 512 \text{ bytes} \end{aligned}$$

    ⇒ 9 records per block.
  - Typically, one stores the *number* of records before the records, e.g.,



  - Most often, a higher-level program will ask for a particular block of a particular file: "Get me the first block in file PASSENGER", e.g.,

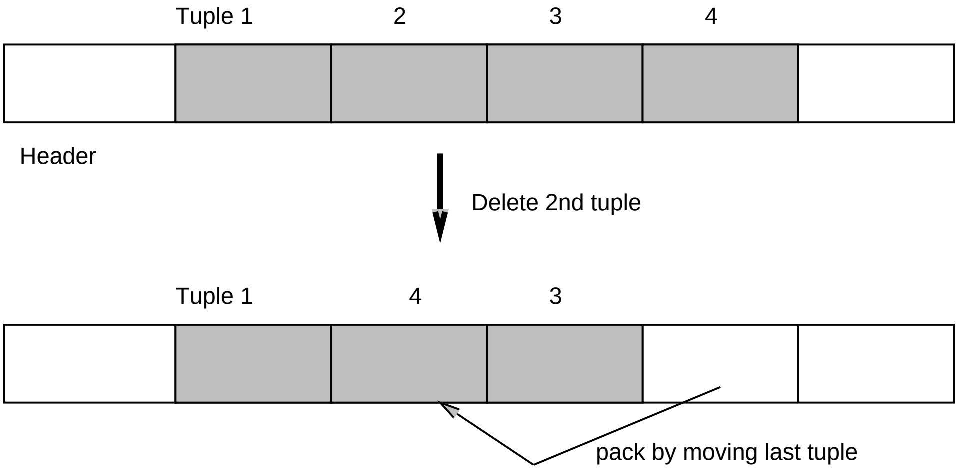
```
blk_num = 1;
char * buf = (char *) get_block ('passenger', blk_num);
```

  - A *file manager* translates the file name and block number within the file to a disk block number.
  - The buffer manager's job is to retrieve the block and return the start address.

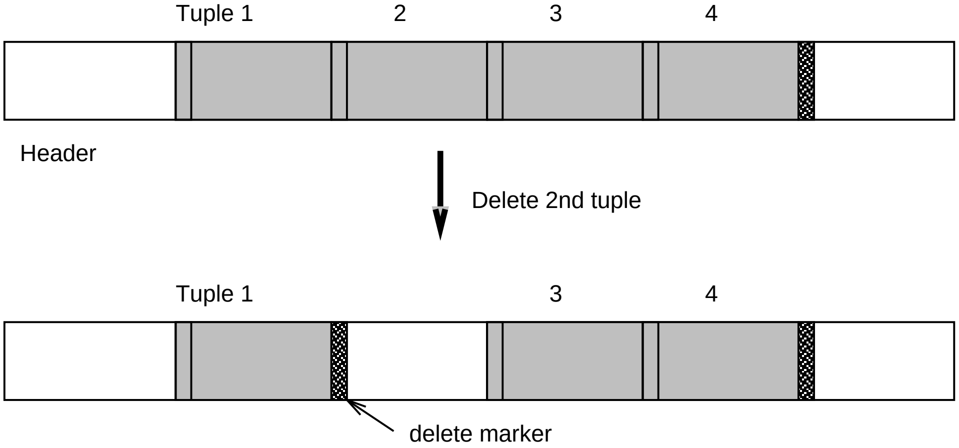
- Thus, the 4-th tuple can then be retrieved knowing the record size, e.g.,

```
char * tuple = buf + (4 - 1) * rec_size;
// Thus, tuple[261] is the 6th byte of the 4th tuple
```

- What about deletions?
  - **Packed-block** option: replace deleted tuple with last tuple in block.

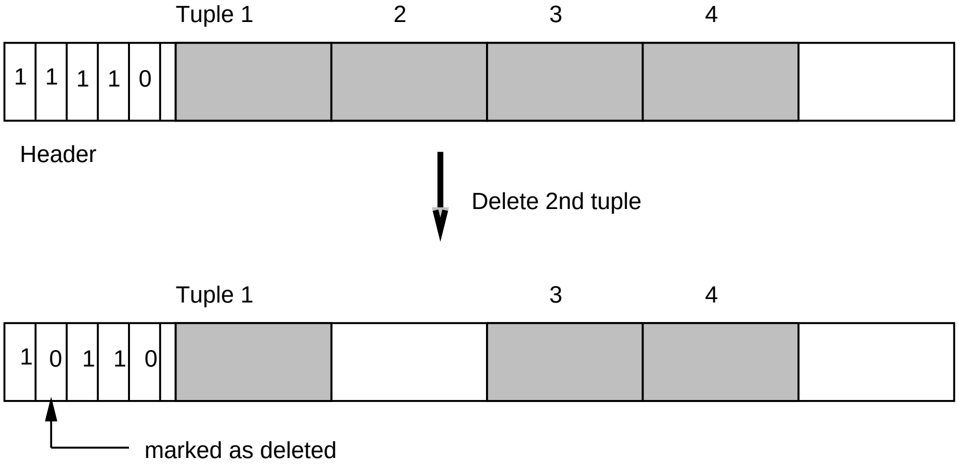


    * Advantages: Simple.
    * Disadvantages: It's a problem if other programs keep pointers to tuple-offsets within blocks (as some B-tree algorithms do).
    * Another disadvantage: Rearranging the tuples may destroy sort-order.
  - **Delete-marker** option: use a special bit pattern (if available) to mark a tuple as deleted, e.g., use all 1's.



  - Advantages: Simple. Solves external-pointer problem.
  - Disadvantages: requires special bit pattern.

- **Valid bit-array** option: for a block that can hold n tuples, use an n-bit array, where 0⇒deleted.



  - Advantages: solves external-pointer problem. No special bit-pattern required.
  - Disadvantages: Overhead required for bit-array and overhead needed during each access.

- Since overwhelming cost is disk access cost, extra computation within memory (e.g., moving tuples around) is not terrible.

4.7 Variable-Size Records

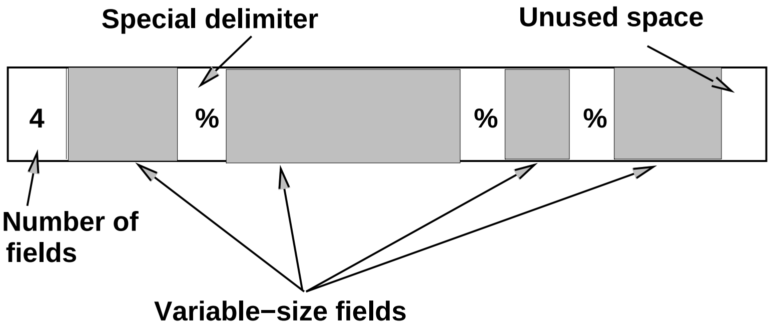
- Variable-size records.
  - Recall: every tuple in a relation has the same number of attributes.
  - Then, how do variable-sized records arise?
  - Consider the following relation:

```
create table BOOK  (  TITLE           char(100),
                      FIRST_AUTHOR    char(50),
                      OTHER_AUTHORS   char(500),
                      RECENT_REVIEW   char(20000) );
```

  Here, some books may have short values for OTHER_AUTHORS and RECENT_REVIEW.
  ⇒ waste of space to use 25000 bytes per tuple
  ⇒ can save space by using variable-size fields
  ⇒ variable-size records.

- Retrieving a field from within a variable-size record:

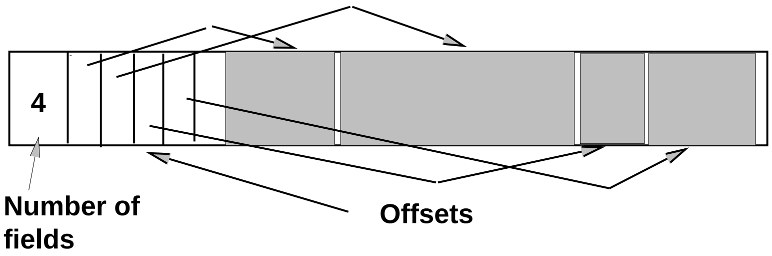
  Option 1: use special delimiters:
  - A special character or string between tuples:



  - Advantages: Simple.
  - Disadvantages: Special delimiter required.

Option 2: Keep offsets to start of each field:

– The first part of the record has offsets to the various records.



– Advantages: No delimiter required. **Null**'s can be easily handled (use zero-offset).

– Disadvantages: Space and time overheads.

- Storage of variable-size records in blocks.

Packed-block method:

– This does not work for variable-size records, since record boundaries are unknown.

Marker method:

– Use special inter-record markers.

– Advantages: Simple.

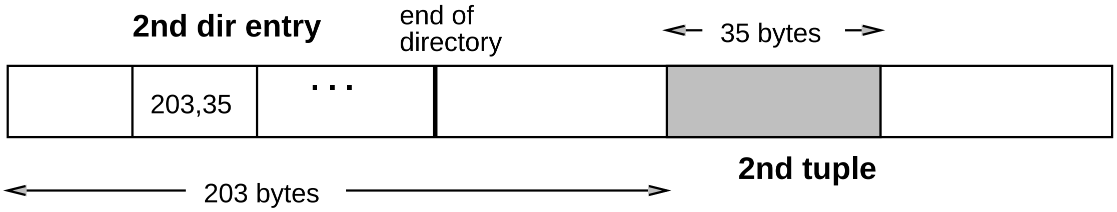
– Disadvantages: Special marker symbol required.

Bit-array method:

– Does not work since record boundaries are not known.

Directory method:

- Store a directory of `(rec_offset, rec_length)` pairs in each block.
- To obtain k-th record, look up k-th non-zero entry in directory, compute offset (using `rec_offset`) and retrieve `rec_length` bytes.



Note: `rec_length` is needed if a new record is inserted into a larger available space.

- Advantages: Most flexible scheme. Has all the advantages of the bit-array method for fixed-size records.
- Disadvantages: Substantially more storage and some computation during access.

- General problem: what if block size is too small for a tuple?
  ⇒ a tuple may *span* several blocks.
  ⇒ only makes life more difficult.

4.8 File Systems

- A single block may not be large enough for all the tuples in a relation
  ⇒ use a file (a collection of blocks).
- Main issues in file design:
  - Blocks can be anywhere on disk
    ⇒ need to keep track of block id's of blocks in each file.
  - Records can be anywhere in file
    ⇒ need a method for organizing records in a file.
- Operations on files a DBMS file system needs to support:
  - **Insert** a record.
  - **Delete** a particular record.
  - **Delete** records that satisfy a given condition.
  - **Modify** a particular record.
  - **Scan** all records. Often, a scan is initiated (**startscan**) and records are obtained one-by-one (**scannext**).
  - **Search** for records that satisfy a condition.
  - **Range search**: when the condition is a range of values.
  - **Equality search**: when the condition is an equality condition.
  - **Reorganize**. Files that leave deletions in place often need to be 'packed'.

  Think of a file system as a C library of functions that can be called for the above tasks.

- Three basic types of files:
  - *Heap files*: records are unordered
  - *Sorted files*: records are ordered by some field.
  - *Hashed files*: records are clustered by a hashing function.

4.9 Heap Files

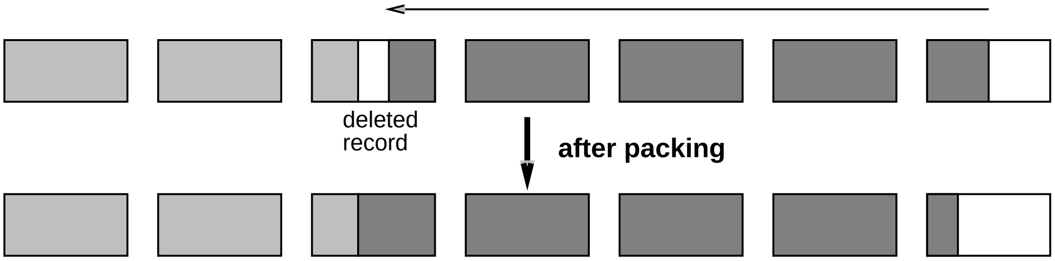
- In a *heap file*, records are unordered.
- File operations using a heap file:

  Insert a record:

  - Simply insert record in last block, if space available.
  - If new block required, request one from disk manager.

  Delete a record:

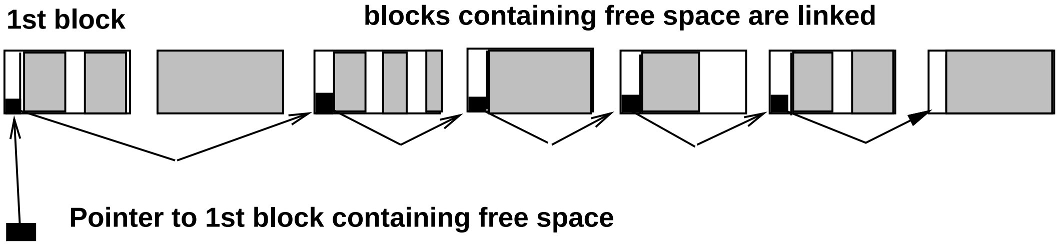
  - Option 1: delete record, then 'pack' remainder of file



    - ∗ Advantages: Simple. No space overhead. No space wasted.
    - ∗ Disadvantages: Time-consuming.

  - Option 2: delete record, pack only occasionally.
    - ∗ 'Occasionally' can mean 'every time total deleted space is larger than a threshold'.
    - ∗ Advantages: Simple. No space overhead. Wasted space can be controlled.
    - ∗ Disadvantages: 'Occasionally' time-consuming.

  - Option 3: delete record, chain deleted space.


1st block
blocks containing free space are linked
Pointer to 1st block containing free space

  - Blocks with deleted space can be added to either end of chain.
  - Key idea: when new records are inserted, try blocks in chain.
  - Advantages: Uses space more efficiently (especially with fixed-size records).
  - Disadvantages: More complex. Long search may be needed for inserting variable-size records. Space overhead for pointers.
- Option 4: delete record, update directory of blocks.
  - Maintain a directory of disk blocks for each file.
  - One entry per block in the directory.
  - Directory entry contains (block number, available space).

⋮ ⋮
(block 5, 35 bytes)
⋮ ⋮

  - The directory itself may span several blocks (it must be stored on disk).
  - On insertion, check directory for existing blocks with space.
  - Since directory entries are typically smaller than tuples, hopefully the directory won't occupy too many blocks.
  - Advantages: Less search required for variable-size tuples. Most flexible scheme.
  - Disadvantages: Space overhead. Occasionally time-consuming.

Scan and any type of **search**:

- For heapfiles, any search requires a complete scan.
- Exception: if it is known that only one record exists to match a condition (e.g., equality search on a key), the search can be stopped when record is found.

4.10 Sorted Files

- In a *sorted file*, records are ordered on some field.
- The field (or group of fields) is usually the *primary key* of the relation.
- Some DBMS's (e.g., Postgres, Oracle) use an internal tuple-id (integer) for each tuple
  $\Rightarrow$ files are typically sorted by tuple-id unless otherwise specified.
- File operations using a sorted file:

  Equality search on sort field:

  - Sorted files were devised to speed up search.
    $\Rightarrow$ use binary search on sort field
    $\Rightarrow$ if $b = \#$ blocks, search time is $O(\log b)$ instead of $O(b)$
  - Note: to retrieve blocks in the middle, a directory is required:

WHICH BLOCK	ACTUAL BLOCK NUM
⋮	⋮
13	4578
⋮	⋮

  - Advantages: Speed.
  - Disadvantages: Directory required. Insertion is difficult (see below).

  Insert a record:

  - Option 1: Make space to insert.
    - ∗ Find right place to insert a record to keep sorted order.
    - ∗ Push back remaining records to create space (and write those blocks back).

    - ∗ Write new record.
    - ∗ Advantages: Keeps sorted order.
    - ∗ Disadvantages: Insert is very expensive.
  - – Option 2: Insert in temporary overflow area.
    - ∗ Insert record in overflow block(s).
    - ∗ Periodically reorganize file: sort the overflow blocks, then merge-sort with main file.
    - ∗ To search: use binary search in main file, sequential search in overflow blocks.
    - ∗ Advantages: Most inserts are fast. Overflow area can be controlled.
    - ∗ Disadvantages: Frequent changes (inserts/deletes) are time-consuming. Search may be slow if overflow area is large.

Delete a record:

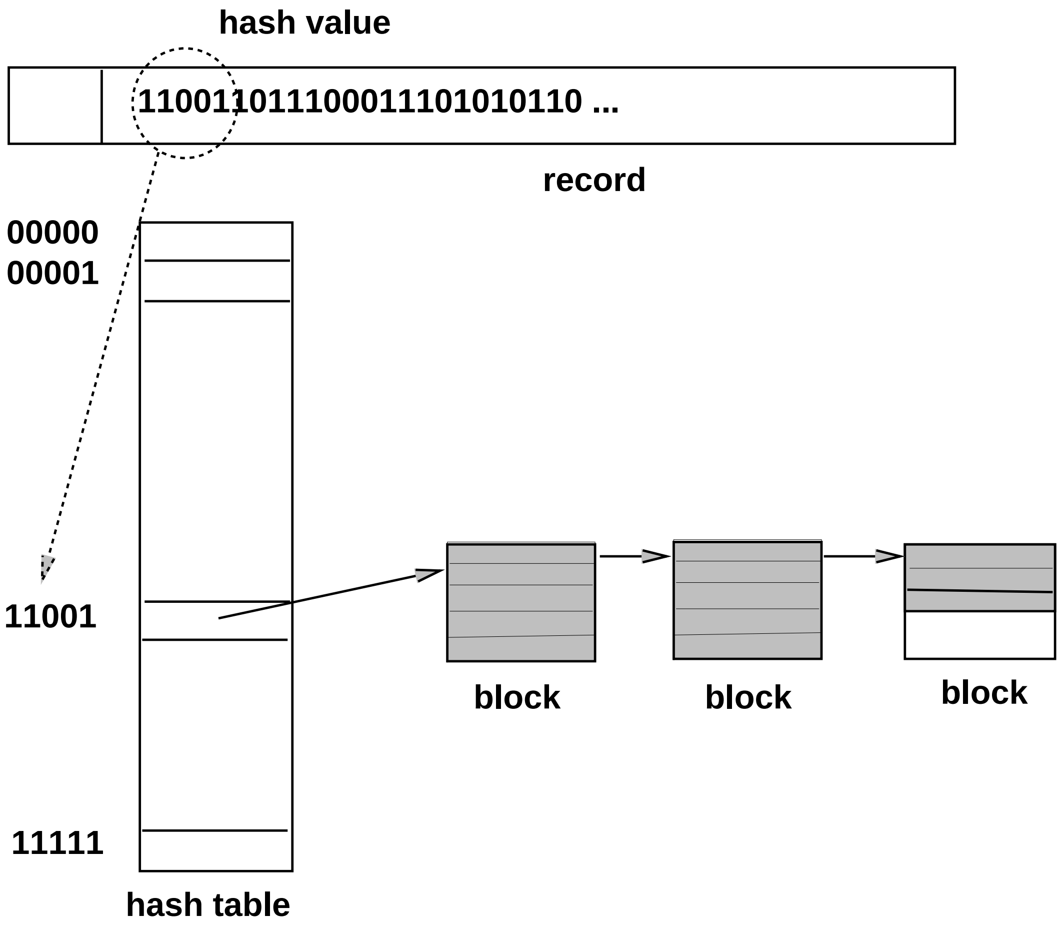
- – Option 1: delete record, then pack file.
  - ∗ Advantages: No space wasted.
  - ∗ Disadvantages: Time-consuming.
- – Option 2: delete record, pack file occasionally.
  - ∗ Advantages: Simple. No space overhead. Wasted space can be controlled.
  - ∗ Disadvantages: 'Occasionally' time-consuming.
- – Note: chaining deleted space is useless since records must be in sorted order.

Range search:

- – If range search is on sort-key, find first tuple and then scan until out of range.
  ⇒ cannot be more efficient.
- – Range search (or plain search) on any other fields requires a complete scan.

4.11 Hashed Files

- In a *hashed file*, records are distributed among several 'buckets'.
- Key ideas in simple hashing:



– A hashing function is applied to each record
  ⇒ a mapping function from the bits in the record to integers
  e.g, pick the first 5 bits in the record
  ⇒ maps records to the integers $0, 1, 2, \ldots, 31$.

- Each integer is associated with a *bucket*: the integer is the *bucket number*.
- Initially, a block is assigned to each bucket
  ⇒ as records get added to the bucket they are placed in the block.
- Later, when the first block assigned to a bucket fills up, additional blocks are used.
- The hash table itself is a (small) collection of blocks.
- Usually: try to pick hashing function so that records spread evenly across buckets.
- Some buckets could have long overflow chains
  ⇒ resembles heap file.
- Dynamic hashing methods handle non-uniformity (to be covered later).

- File operations using a hashed file:

  Insert a record:

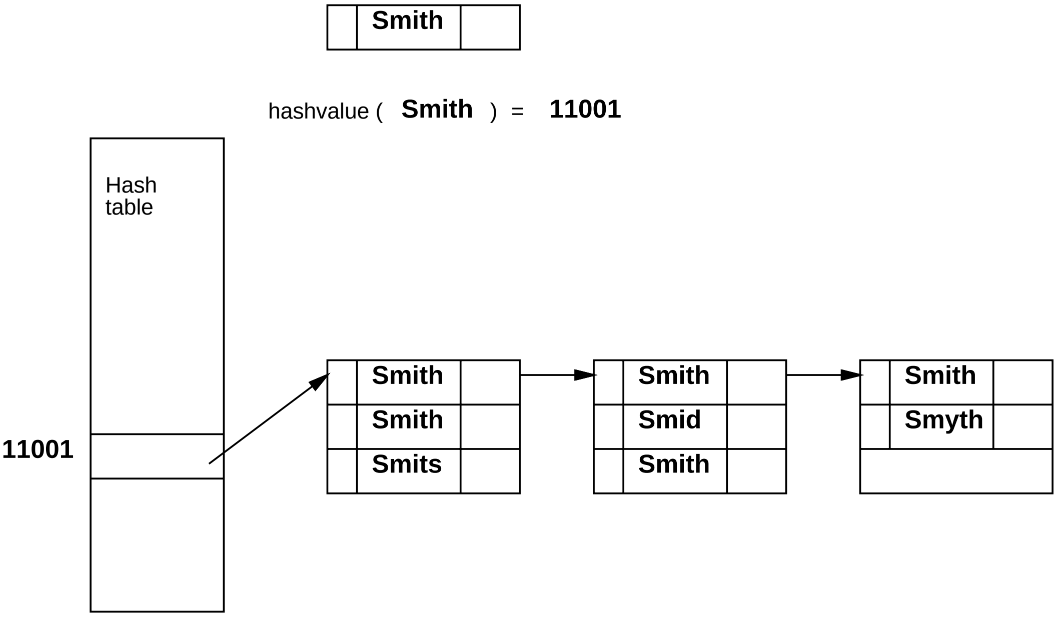
  - Compute hash value and insert in appropriate bucket.
  - New disk block may be needed if all disk blocks in bucket are full.

  Delete a record:

  - Remove record from disk block.
  - Return disk block to disk manager if empty.
  - No need to keep track of deleted space.

  Equality search:

  - Compute hash value of equality argument.
  - Search disk blocks in corresponding bucket.
    ⇒ if there are few blocks per bucket, few disk accesses are needed.
  - Example:



Other searches and scans:

- Scan all blocks in hashed file.

4.12 File Structures: A Summary

- Key ideas in the three file structures:

  Suppose b =number blocks in file.
  Heap files:

  - Plus: Simple. Fast insertion.
  - Minus: Any search is slow ($O(b)$).
  - Best if file scans are common or insertions are frequent.

  Sorted files:

  - Plus: Search on sort-key is fast ($O(\log b)$). Range search is fast (on sort-key)
  - Minus: Insertion is slow. Any other search is slow.
  - Best if range search is needed often.

  Hashed files:

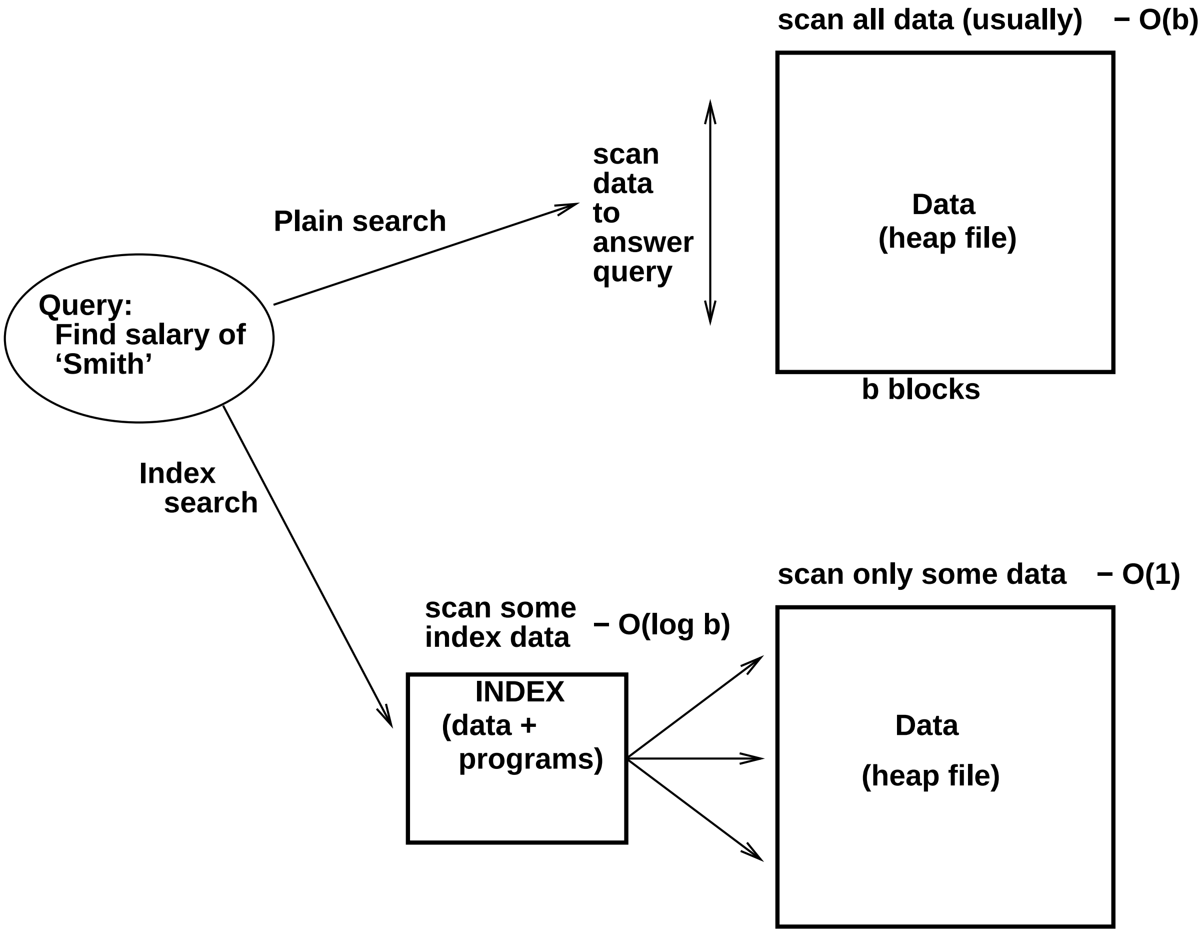
  - Plus: Insertion or search on hash-key is fast. ($O(1)$ if properly distributed).
  - Minus: Complex. Any other search is slow. Space overhead. Unbalanced buckets degrade performance. Range search is slow.
  - Best if equality search is needed on hash-key.

- In practice:

  - Sort files are rarely used, since indices such as B^+-trees provide for fast search, insert and range search.
  - Hash files are rarely used for direct storage. However they are often used as temporary files during *join* computations.
  - Unless otherwise specified, heap files are most often used.
  - Most systems allow the creation of indices to speed up file access.

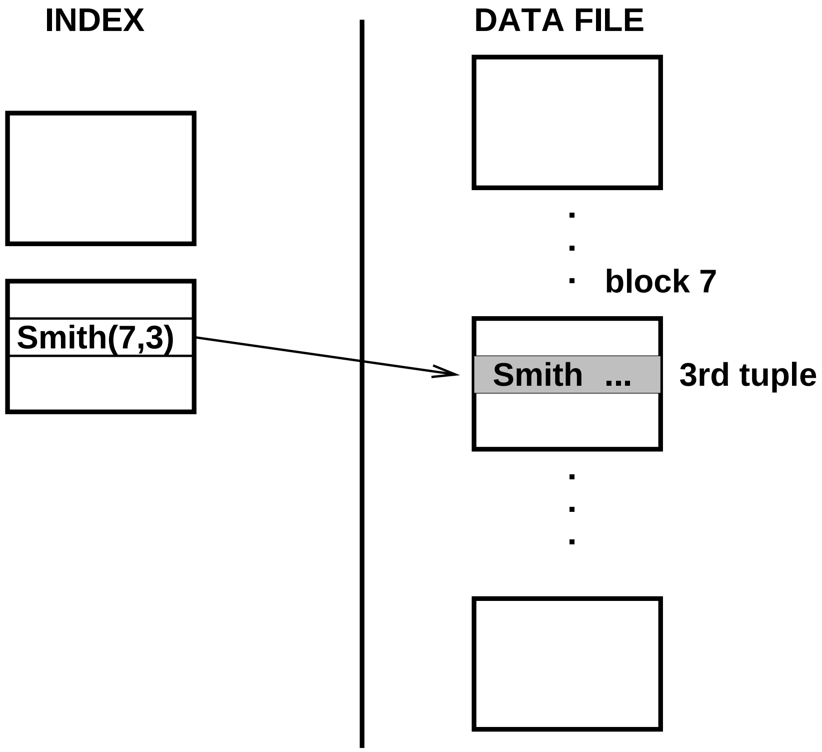
4.13 What is an Index?

- Consider a relation such as EMP (NAME, SSN, SALARY).


scan all data (usually) – O(b)
Query: Find salary of 'Smith'
Plain search
scan data to answer query
Data (heap file)
b blocks
Index search
scan some index data – O(log b)
INDEX (data + programs)
scan only some data – O(1)
Data (heap file)

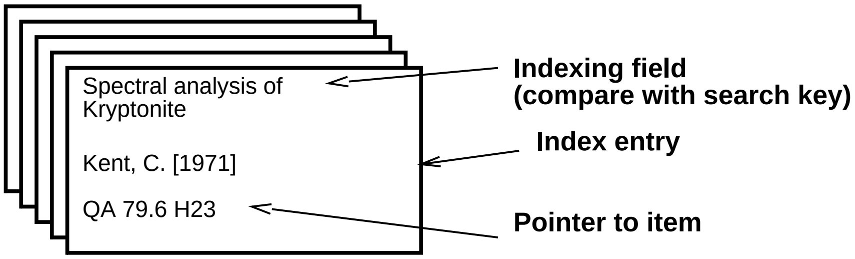
- An index is:
  1. Some data extracted (copied) from the data file placed in a useful data structure (leaving the data file intact).
  2. Programs that manipulate the index data.
- Key idea: narrow down the search to a few pieces of the original data file.

- Indices usually contain 'pointers' to data in the data file.



- A *search-key* is a value that defines a search:
  - In the query "Find the salary of Smith", the string `'Smith'` is the search-key.
  - In the query "Find all employees with a salary greater than 10000", the integer `10000` is the search-key.
  - A search-key is the value used in searching.
  - Unfortunately, the word "key" is being overloaded here.
  - Sometimes, a search-key is also a *key* for the associated relation.

- The main idea behind an index can be found in a library card catalog:



Since each index entry (card) is *smaller* than the data (book) itself, searching the index (card catalog) is faster than search the data file (library) itself.

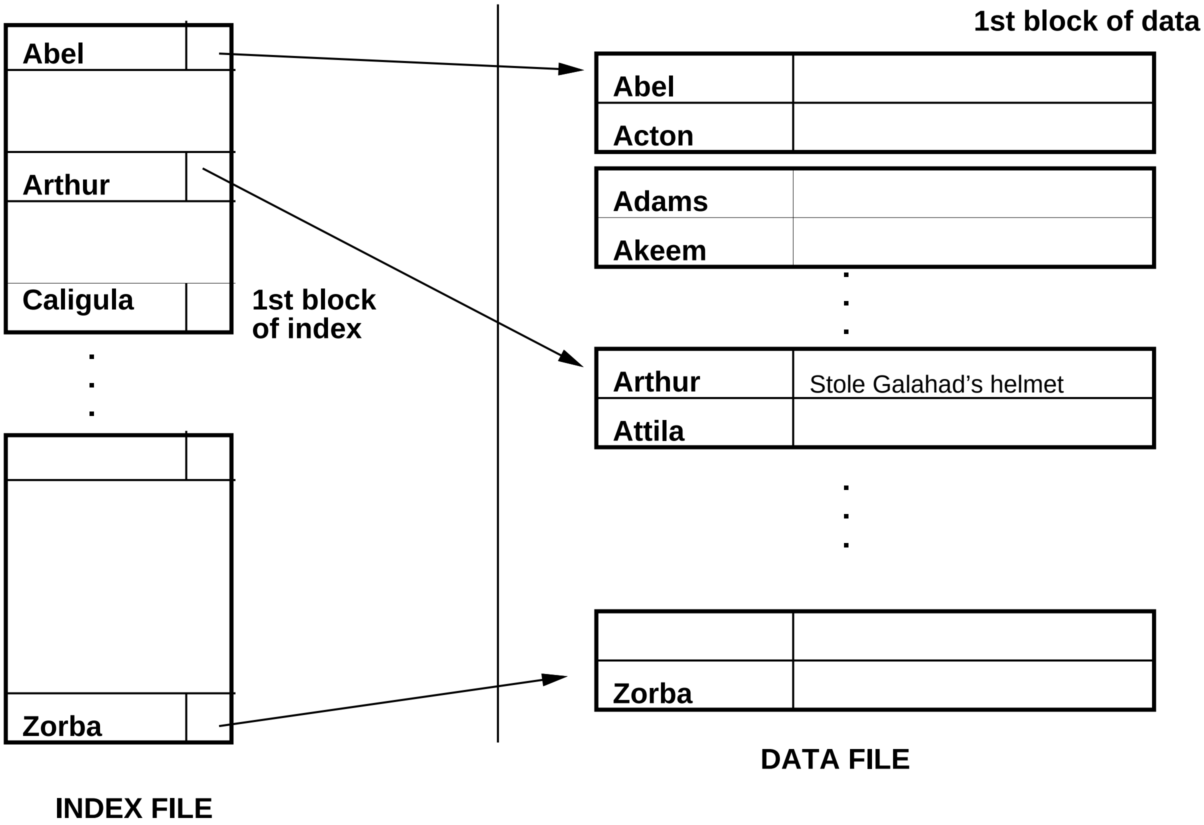
4.14 A Simple Index

- Suppose we have a data file and a search key.
  - Create a file of *index records*.
  - Each index record will have a possible search-key value and a pointer to the appropriate data record.
  - A ‘pointer’ to a data record is really the *block number* of the block in which the data resides, and the *tuple number* within the block.
- Example: consider the following relation:

```
create table BLACKMAIL  (  NAME           char(20),
                           DIRTY_SECRET   char(480) );
```

Suppose the block size is 1024 bytes:

- 500 bytes per data record
  ⇒ 2 data records per block.
- Suppose a pointer requires 5 bytes (4 bytes for block number and 1 byte for tuple number)
  ⇒ 25 bytes per index record
  ⇒ 40 index records per block.



Consider the query: "Find Arthur's dirty secret."

Without index:

- Suppose "Arthur" is the 10th record in the data file
  ⇒ 5 blocks of data file must be read.

Using an index:

- Since "Arthur" is 10th entry
  ⇒ occurs in block 1 of index
  ⇒ 1 block of index file, 1 block of data file
  ⇒ 2 blocks to obtain final record

This may not seem to be much of an improvement. Consider instead the query: find information about "Ulysses".

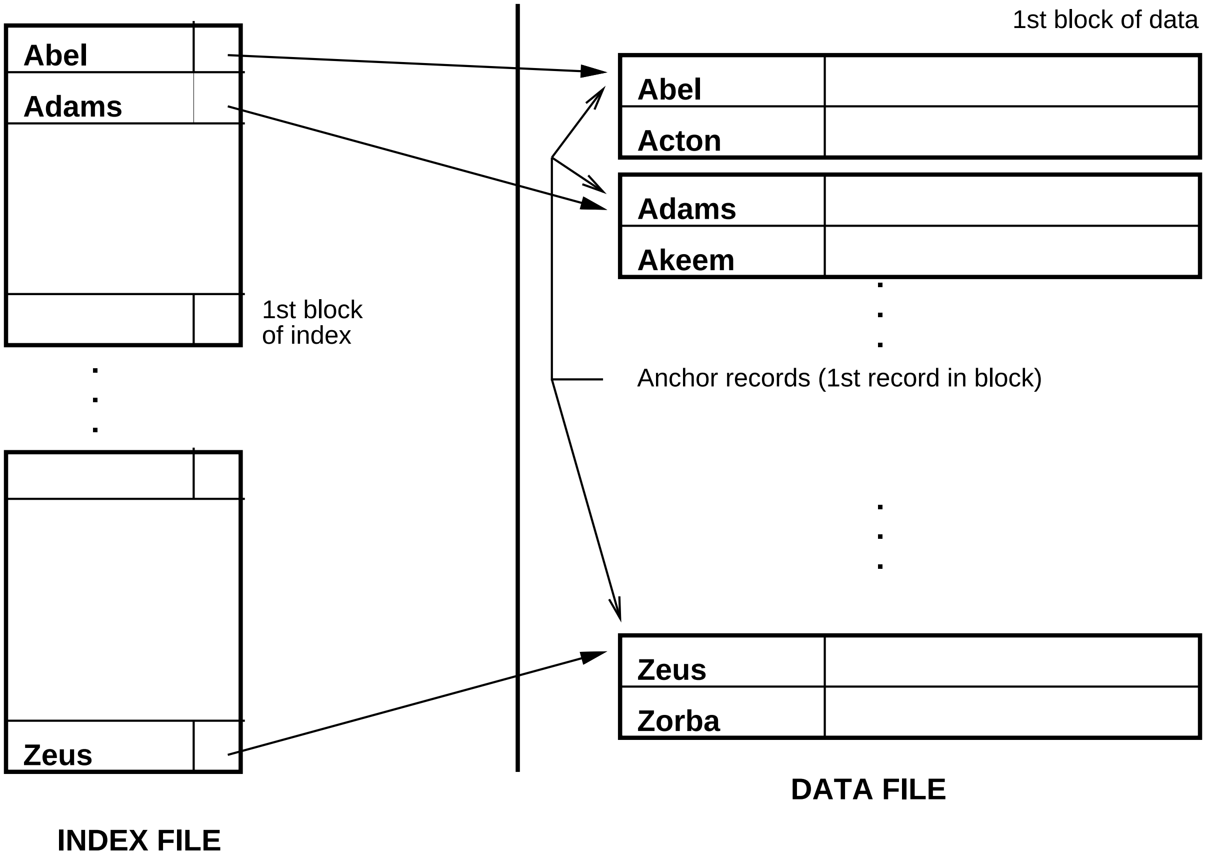
- Suppose "Ulysses" occurs as 30,000-th record
  ⇒ 15,000 blocks accessed in data file scan (no index).

$\Rightarrow$ 30,000/40 = 750 blocks accessed in index file
$\Rightarrow$ 751 block accesses using index.

- The impact of sorting:
  - Suppose

$$\begin{aligned} b &= \text{number of data file blocks} \\ b_i &= \text{number of index file blocks} \end{aligned}$$

  - The index file is typically sorted
    $\Rightarrow$ binary search can be used.
  - If data file is sorted, binary search can be used in direct method.
    $\Rightarrow$ ratio of performance is $O(\log b)/O(\log b_i)$.
  - If data file is not sorted (most common case), then ratio is $O(b)/O(\log b_i)$.
- If the data file is sorted, the index can be made smaller by using *anchor* records (first record in a block):



Since only the correct block needs to be located, knowing that a data item lies between two successive anchor record key-values is enough to determine the block.

In the example:

– 2 data records per block
  ⇒ 15,000 anchor records.
  ⇒ 15,000 index entries
  ⇒ 15,000/40 = 375 index file blocks

⇒ example of a *non-dense* index.

4.15 Types of Indices

- Note: “Indexes” and “Indices” are both valid.
- An index that has one entry for each data record is called a *dense index*. Otherwise, a *non-dense index*.
- The set of attributes corresponding to the search-key is called the *indexing field*.

  For example, one can create an index for the combination (NAME, SSN). Then, the bits corresponding to (NAME,SSN) in each record constitute the *indexing field*.
- A *primary index* is an index such that the indexing field includes the primary key of the data file.

  Note: in some books, a *primary index* additionally requires the data file to be sorted.
- A *secondary index* is any index that is not a primary index.

  For example, consider

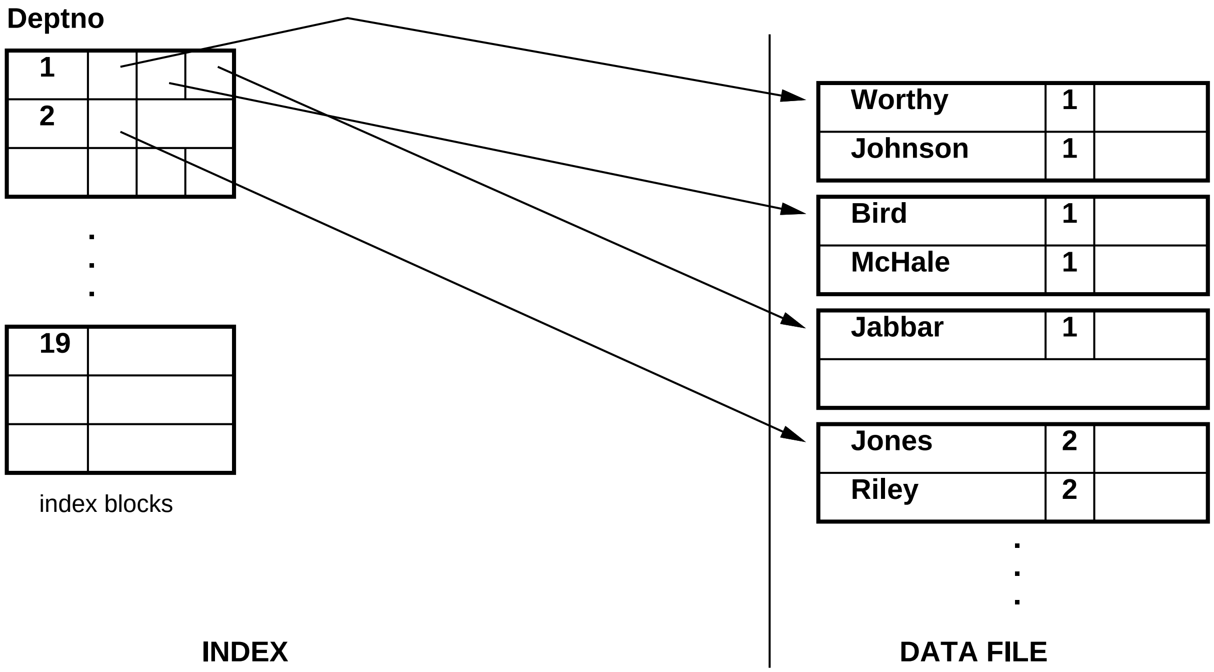
  STUDENT (NAME, SSN, STUDENT_ID, ADDR).

  Then, an index on SSN is a primary index. An index on STUDENT_ID is a secondary index.
- A *clustering index* is a secondary index on a non-key set of attributes.

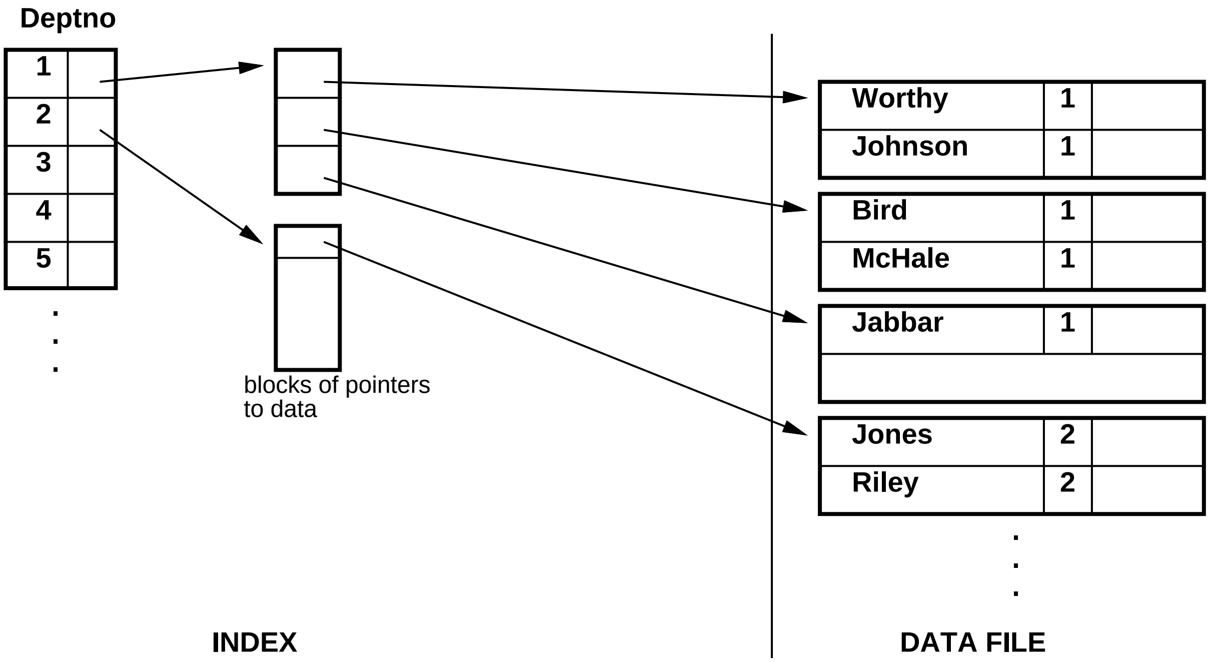
  For example, consider

  EMP (NAME, DEPTNO, SSN, ADDR)

  An index can be created on DEPTNO by *clustering* together data records having the same DEPTNO:



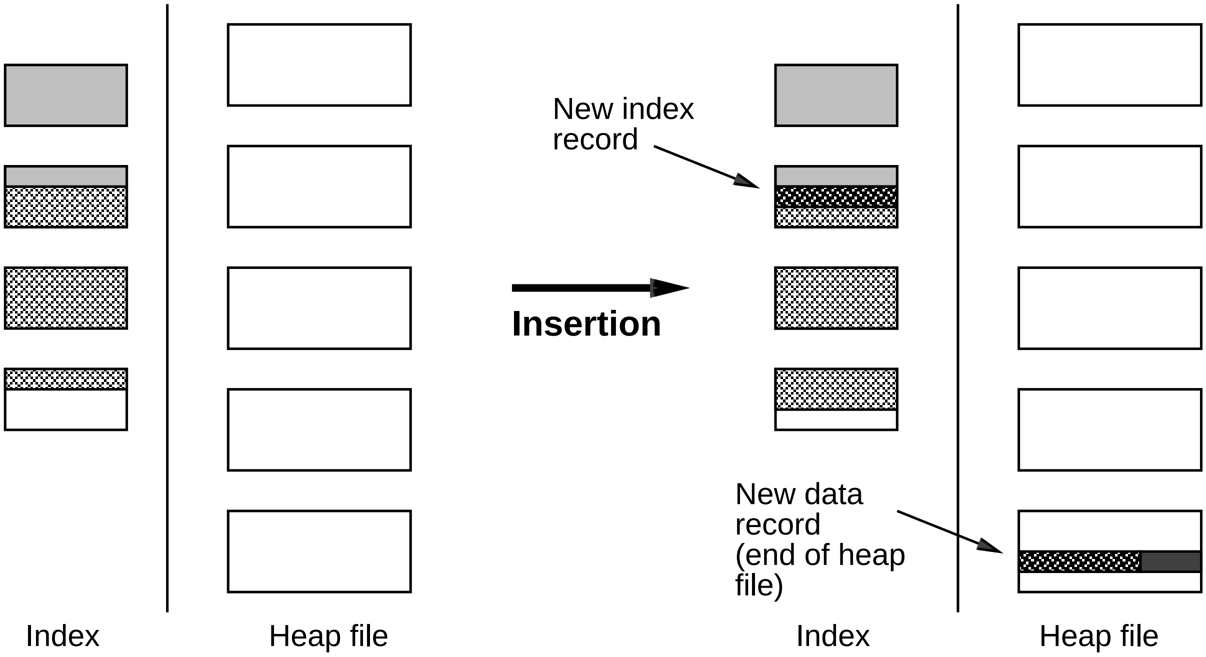
Sometimes, if there are too many values, separate blocks of pointers (block numbers) are used:



- Insertion and deletion:

  Insertion:

  - Index file is sorted
    ⇒ new index record needs to be inserted in proper place.
  - If data file is a heap file, insertion of data record is easy.



  - For a sorted data file, data record needs to be in correct place.
    ⇒ either squeeze record in or use overflow blocks. If a non-dense index is used
    ⇒ anchor records may change in re-packing
    ⇒ large part of index may need to be re-built.
  - Overflow blocks can be used for index file as well
    ⇒ periodic re-organization needed.
  - In both index and data files, space can be initially reserved for future inserts.
  - Insertion into a clustering index is easy: data file is a heap file. New block pointers are simply added to collection of pointers.

Deletion:

- Deletion from data file follows earlier techniques.
- If anchor record is deleted, new anchor record must be designated and written into index entry.
- Several options for deleting index record:
  - Delete and pack
    $\Rightarrow$ time-consuming, but no space wastage
  - Delete and leave space
    $\Rightarrow$ could waste space and later slow down access.

- Primary vs. Secondary indices:
  - Recall: a primary index is an index on a primary key.
  - A secondary index is an index on any other set of attributes.
  - If the data file is sorted by primary key, the primary index can be made *non-dense*
    $\Rightarrow$ fewer index entries
    $\Rightarrow$ binary search on index is faster.
  - However, sometimes improvement using a secondary index can be greater, if the data file is sorted on primary key.
    - ratio of performance for primary index is $O(\log b)/O(\log b_i)$.
    - ratio of performance for secondary index is $O(b)/O(\log b_i)$.

    where $b = \#$ data blocks and $b_i = \#$ index blocks.
  - Example:
    - Suppose block size is 1024.
    - Suppose we have a data file of 50,000 records of 100 bytes each.
      $\Rightarrow 1024/100 = 10$ records/block
      $\Rightarrow$ 5000 blocks
      $\Rightarrow$ 2500 blocks on average for linear search
    - Binary search of data file: $\lceil \log_2 5000 \rceil = 13$ blocks.

- $\ast$ Suppose index record is 10 bytes

  $\Rightarrow$ 1024/10 = 102 index records per block
- $\ast$ 5000 data blocks

  $\Rightarrow$ 5000 primary index records

  $\Rightarrow$ 5000/102 = 50 index blocks

  $\Rightarrow$ binary search of index file: $\lceil \log_2 50 \rceil = 6$ blocks

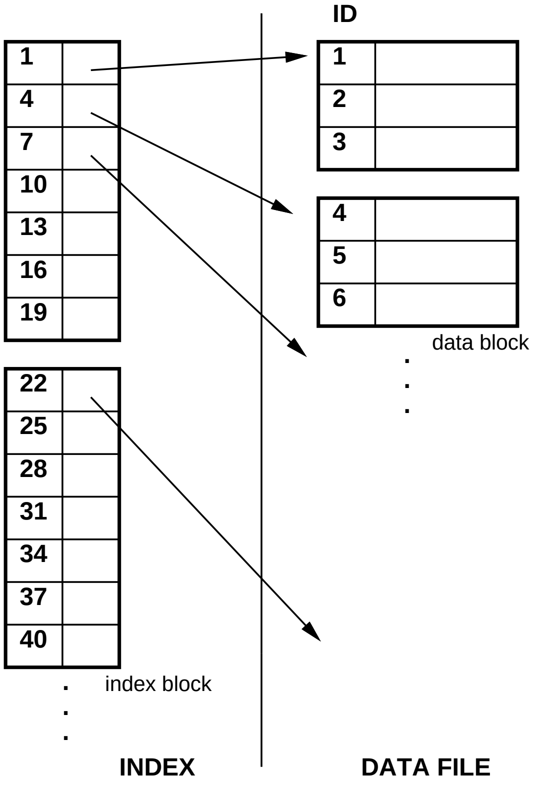
  $\Rightarrow$ search needs 6 + 1 (for data file) block accesses.
- $\ast$ Thus, improvement ratio for primary index is $13/(6+1) \approx 1.8$.
- $\ast$ Now, for a secondary index, we need 50,000 index entries

  $\Rightarrow$ 50000/102 = 491 blocks

  $\Rightarrow$ binary search of index file: $\lceil \log_2 491 \rceil = 8$ blocks.
- $\ast$ Data file requires linear search.
- $\ast$ Thus, improvement ratio for secondary index is $2500/(8+1) \approx$ 277.8.

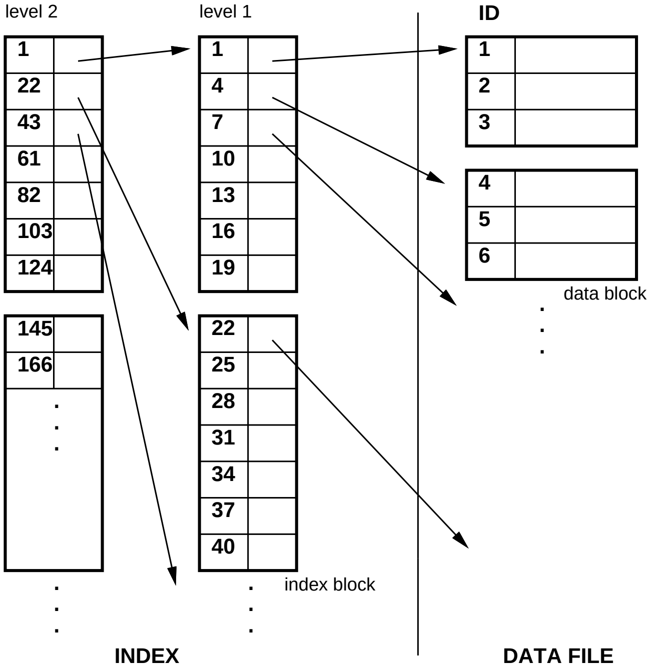
4.16 Multilevel Indices

- Consider an index on STUDENT (ID, NAME, ADDR):



Typically: binary search is used on index file.

Key observation: index file is itself a sorted “data” file.
⇒ can create a primary index on the index:



In this case:

- The second index file (level-2) is an index on the first one (level-1).
- Level-2 contains the anchor values of level-1.
- This is called a *multilevel index*.

- Suppose level-1 has b_i blocks and an index block contains f index records:
  $\Rightarrow$ One level-2 entry for each of the b_i blocks
  $\Rightarrow$ b_i records in level-2
  $\Rightarrow$ $\frac{b_i}{f}$ blocks in level-2
  $\Rightarrow$ level-2 search cost is $\lceil \log_2 \frac{b_i}{f} \rceil$
  $\Rightarrow$ total search cost is: $\lceil \log_2 \frac{b_i}{f} \rceil + 2$
  (1 block in level-1, 1 block from data file).
  $\Rightarrow$ search time is reduced.

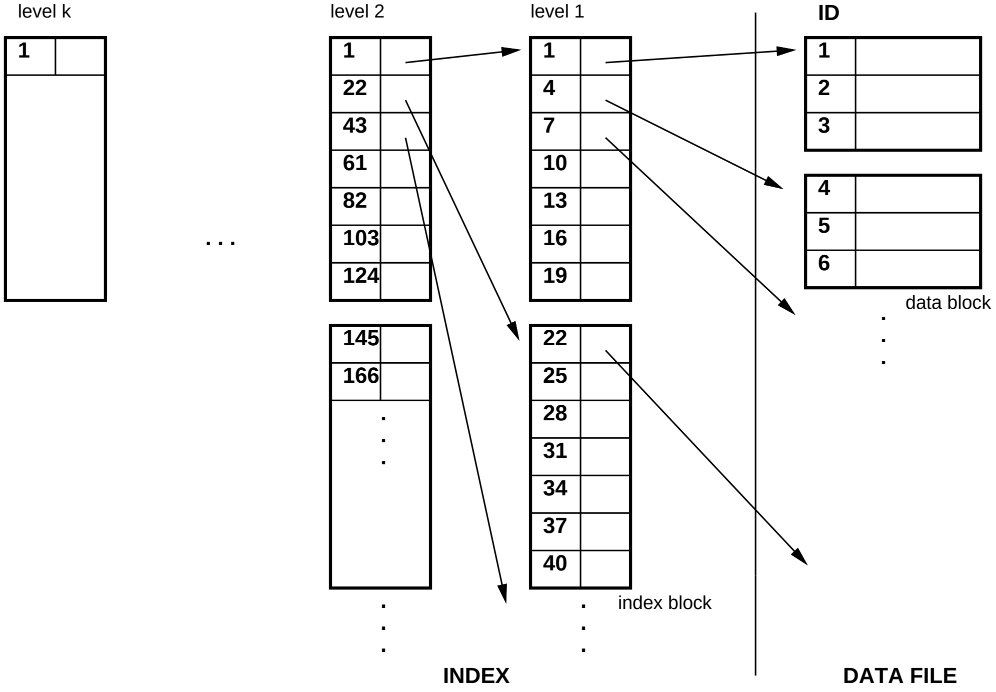
- What about a level-3?
  $\Rightarrow$ $\frac{b_i}{f}$ blocks at level-2
  $\Rightarrow$ $\frac{b_i}{f}$ records at level-3

⇒ $\frac{b_i}{f} \div f$ blocks at level-3

$\vdots$

⇒ $\frac{b_i}{f^{k-1}}$ blocks at level-k.

- When do we stop?
  ⇒ when level-k has only *one* block
  ⇒ $\frac{b_i}{f^{k-1}} = 1$
  ⇒ $k = 1 + \log_f b_i$.



- What is the search cost with a *full* multilevel index?
  ⇒ 1 block per level plus data block
  ⇒ $(k+1) = 2 + \log_f b_i$.

- Is it worth the effort?
  - Cost with one-level index: $1 + \log_2 b_i$.
  - Cost with multi-level index: $2 + \log_f b_i$.

- If $f \gg 2$
  $\Rightarrow 2 + \log_f b_i \ \ll\ 1 + \log_2 b_i$.
- Example: $b_i = 4000$, $f = 100$
  $\Rightarrow 1 + \log_2 b_i = 12$ and $2 + \log_{100} b_i = 3$
  $\Rightarrow$ four times faster.

- How much extra space?
  $\Rightarrow O(b_i)$ additional blocks.

- What about insertions and deletions?
  Insertions and deletions are problematic.

  Typical options for **insert**:

  - Reorganize entire index with each insert:
    $\Rightarrow$ expensive operation but keeps search efficient.
  - Leave some space in each (index and data) block for future inserts
    $\Rightarrow$ potential wastage of space
    $\Rightarrow$ slower search.
  - Place new data and index records in overflow area
    $\Rightarrow$ slower search.

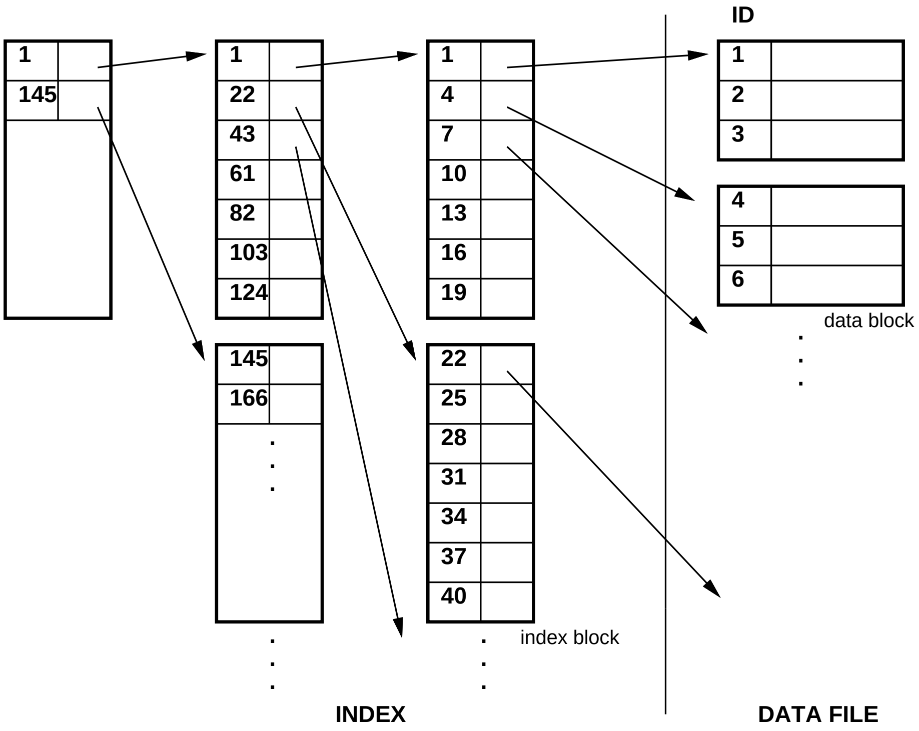
  Delete options are similar to those seen earlier.

- Note:

  - Additional levels can be built on any index file, since level-1 is sorted (even for clustered and secondary indices).
  - A multilevel index is sometimes called a ISAM (Indexed Sequential Access Method) by IBM-types.
  - Multilevel indices were used in practice *before* the arrival of B-trees.
  - Some ISAM products are still available.

4.17 B-Trees: An Introduction

- Recall multilevel indices. For example:



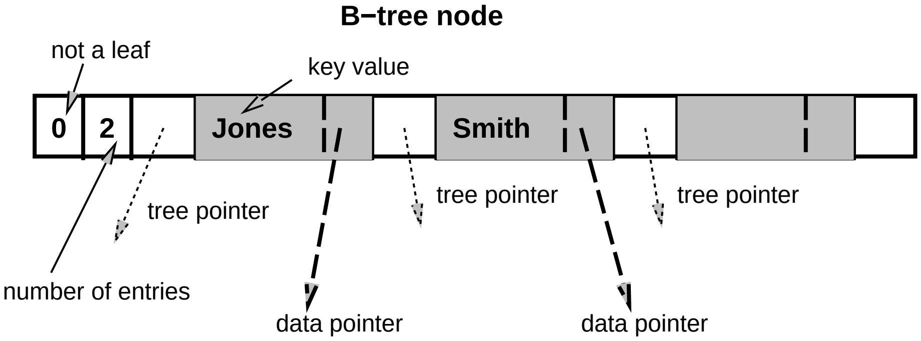
  - Observe that values like 1 and 145 occur in several places in the index
    ⇒ deleting '1' from the index requires several record deletions.
  - If deletions are left in place (no packing), some index blocks could contain many deleted values
    ⇒ even if data file becomes small, index remains large.
  - If re-packing is done for each insertion, $O(b_i)$ blocks may be processed.
  - If overflow blocks are used, then many insertions cause the overflow blocks to be processed frequently

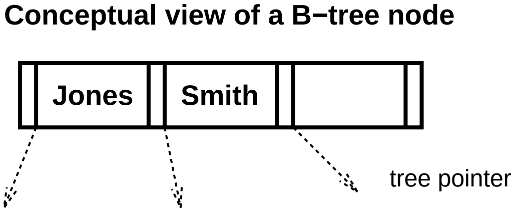
$\Rightarrow$ slower search.

- A B-tree is a type of multilevel index which:
  - is dense;
  - implements insertion and deletion by dynamic reconstruction of the index at the time of each insertion/deletion;
  - limits the cost of dynamic reconstruction to approximately $O(\log_f b_i)$ block accesses;
  - Incurs a search cost of approximately $O(\log_f b_i)$ block accesses;
  - limits the amount of wasted space;
  - does not allow duplicate values in the index.
- A B-tree is a collection of blocks (called B-tree nodes) that contain:
  - search-key (index) values
  - data pointers (block numbers and tuple numbers of data records)
  - tree pointers (block numbers of other B-tree nodes)
  - some information local to each block (such as the number of key values in it)
- Every B-tree node of a B-tree of *degree* m ($m > 1$):
  - (except the root) must have at least $m - 1$ key values, sorted inside the node;
  - cannot have more than $2m - 1$ key values;
  - has as many data pointers as key values;
  - has one more tree pointer than the number of key values;
  - is either a *leaf* or an *internal* node.
- Each B-tree node fits into one disk block and the entire B-tree resides on disk
  $\Rightarrow$ when a B-tree is accessed, relevant nodes are brought into main memory.

Additionally:

- Internal nodes contain tree pointers whereas leaves do not;
- All leaves are at the same depth from the root.





For example, suppose

- block size is 512
- 2 bytes are used for block-specific information
- an index value requires 20 bytes
- a data pointer requires 6 bytes
- a tree pointer requires 4 bytes

Then, we must have

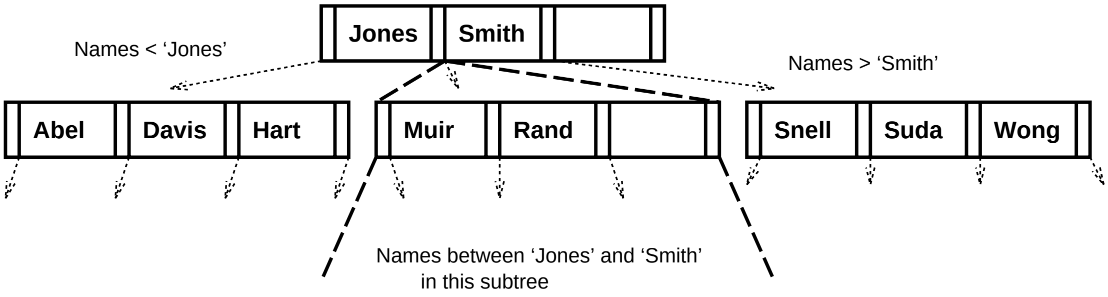
$$2 + 20(2m - 1) + 6(2m - 1) + 4(2m) \quad \leq \quad 512.$$

Or

$$m \leq 8.93.$$

Thus, pick $m = 8$.

- The nodes in a B-tree are arranged in *in-order*:



- NOTE:
  - The root of the B-tree can have fewer than $m - 1$ values.
  - Since $2m-1$ is odd, the maximum number of values is either 3,5,7,... etc.
  - Since $m > 1$, smallest value possible for $2m - 1$ is 3.
  - In a full node, the m-th value is the called *median* or *middle* value.
  - It is possible to generalize the definition to allow for an even number of values. If e values are allowed (e.g. $e = 8$), the middle value is defined to be the $(e + 1)/2$-th value.
- Operations supported by a B-tree:
  - **Insertion**: insert a key-value into the tree.
  - **Deletion**: delete a key-value from the tree.
  - **Search**: search for a particular key-value.
- NOTE:
  - A B-tree is used on top of a data file (usually, a heap file).
  - Often, the combination of the tree and data file is called a B-tree file (or, confusingly, B-tree for short).

    ⇒ the B-tree's **insert** function also inserts into the data file:

```
btree_insert (key, tuple)
```

  - *Search* in a B-tree typically means *equality search.*
  - Range searches are inefficient in a B-tree.

- Remember: the *tree pointer* in a B-tree node is *not* a memory location $\Rightarrow$ it is a block number.

- Worst-case height of a B-Tree.

  It is possible to compute the worst possible height of a B-tree containing n nodes without additional detail.

  In the worst case, all nodes have only $m-1$ values and the root has only 1 value. Thus,

  level 0 (root) has 1 node
  level 1 has 2 nodes
  level 2 has $2m$ nodes
  level 3 has $2m^2$ nodes
  $\vdots$
  level h has $2m^{h-1}$ nodes

  Therefore,

$$\begin{aligned} n &= 1+2+2m+2m^2+\ldots+2m^{h-1} \\ &= 1+2(1+m+m^2+\ldots+m^{h-1}) \\ &= 1+2\frac{m^h-1}{m-1} \end{aligned}$$

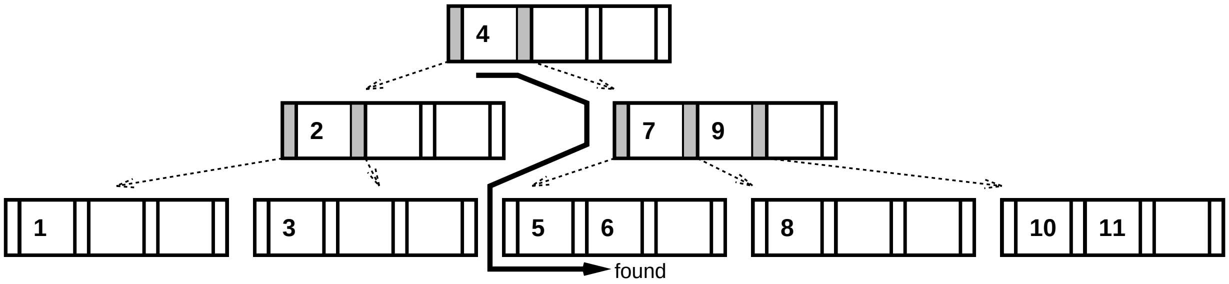
  Or, $h = O(\log_m n)$.

  In fact, $h \leq 1 + log_m n$.

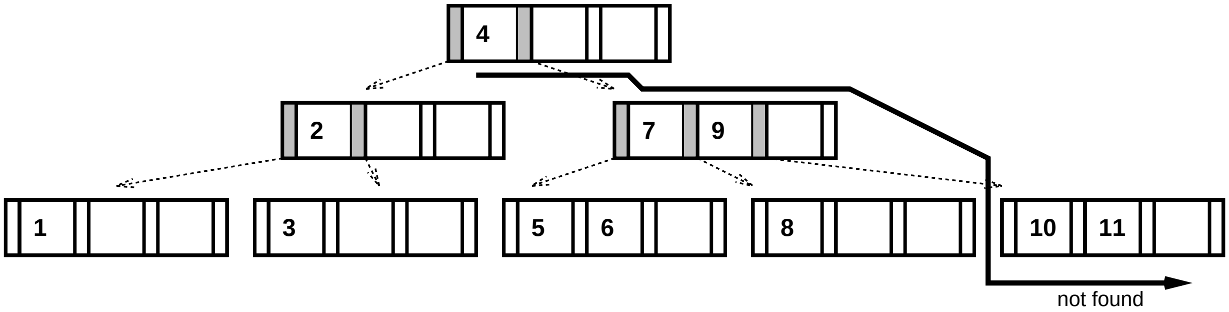
- NOTE:
  - The $m-1$ lower bound translates to "at least 50% full".
  - Any lower bound less than $m-1$ is easy to support
    $\Rightarrow$ a restriction like "at least 70% full" is possible but difficult.

4.18 Search in a B-Tree

- Searching in a B-tree follows in-order traversal.
  - Start at root block.
  - For each block, search sequentially through elements in block until element is either found or the correct child block is identified.
  - If child block pointer is NULL
    ⇒ item not in tree.
  - If item is found
    ⇒ extract data pointer and retrieve tuple from heap file.
  - Otherwise, follow childpointer.
- Example: search for '6' in this tree:



- Example: search for '14':



- What is the complexity of search?
  $\Rightarrow$ as many blocks as height of tree (worst case)
  $\Rightarrow$ $O(\log_m n)$ for search (n blocks of data).

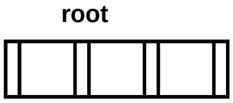
4.19 Insertion in a B-Tree

- We will learn insertion via an example:
  - Consider the case $m = 2$.
  - The key values are integers.
  - Insert the following key values in order: 1,7,8,10,9,3,2,5,4,6,11

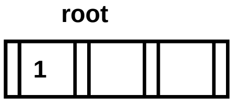
  NOTE:

  - $m = 2 \Rightarrow$ at least $m - 1 = 1$ value per node.
  - $m = 2 \Rightarrow$ at most $2m - 1 = 3$ values per node.
  - Median value is 2nd value.
  - We will not describe insertion of tuple into heap file.

- Initially: Create (empty) root node:

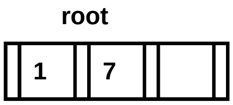


- Insert '1':
  - Space available in root block.

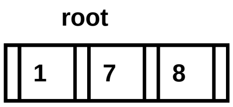


- Insert '7':
  - Fits into root block

- Insert to right of ‘1’ (in sort-order)

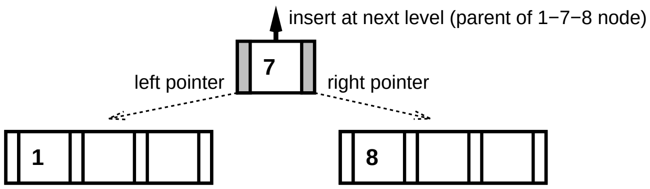


- Insert ‘8’:
  - Fits into root block
  - Insert to right of ‘7’

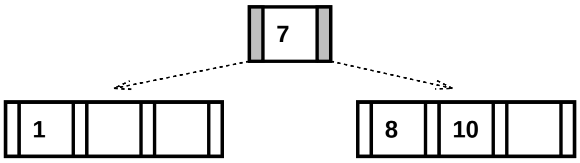


- Insert ‘10’:
  - Root node is full ⇒ must split root.
  - Median element (‘7’) is *bumped up* one level (to new root).
  - Split nodes are children (left and right) of median element.
  - New element (‘10’) is added in appropriate split node.

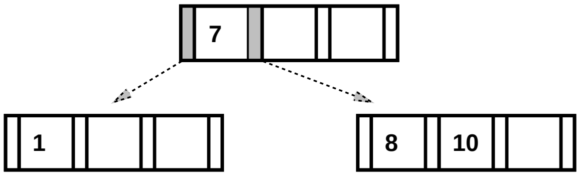
Step 1: Split node:



Step 2: Add new value (‘10’):



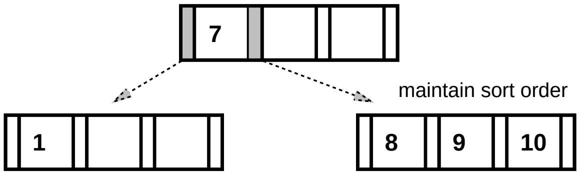
Step 3: Insert median element (with its left and right pointers) at level above.
⇒ in this case, create new root.



This illustrates the general principle behind insertion:

- First, find the correct leaf for insertion.
  ⇒ use in-order to navigate to correct leaf.
- Note: check equality within interior nodes.
  ⇒ if found then insertion is a duplicate.
- If space is available, insert into leaf (maintaining sort order).
- Else, split leaf and 'push up' median element
  ⇒ insert median element into old leaf's parent.
- In pushing up median element, *also move up left and right pointers.*
- Add new item in left or right child (according to sort order).
- If parent is full, split parent etc, recursively.

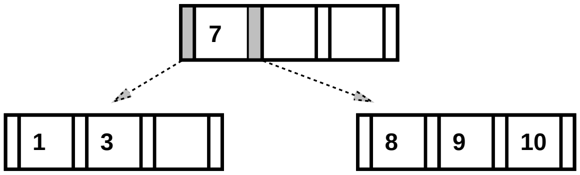
- Insert '9':
  - Search for right leaf ⇒ '8-10' node.
  - Space available ⇒ insert.



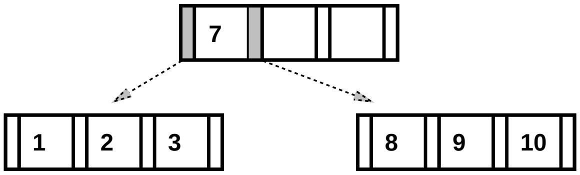
- Insert '3':

– Search for correct leaf ⇒ the ‘1’ node.

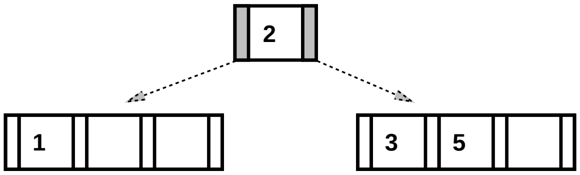
– Space available ⇒ insert.



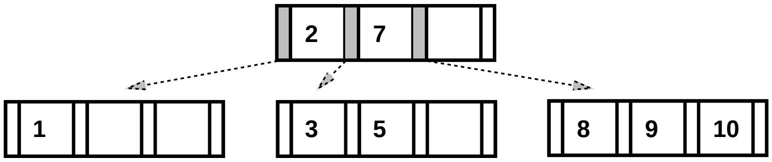
- Insert ‘2’:
  - Search for correct leaf ⇒ the ‘1’ node.
  - Space available ⇒ insert.



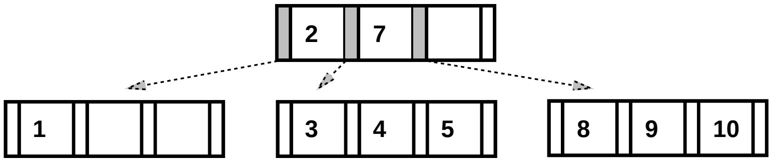
- Insert ‘5’:
  - Search for correct leaf ⇒ the ‘1-2-3’ block.
  - Block is full ⇒ split (median element is ‘2’).
  - Add new element into correct child ⇒ the ‘3’ block:



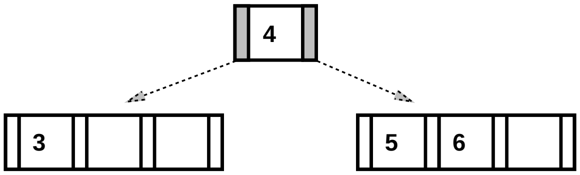
  - Insert ‘2’ into parent (the root, in this case):



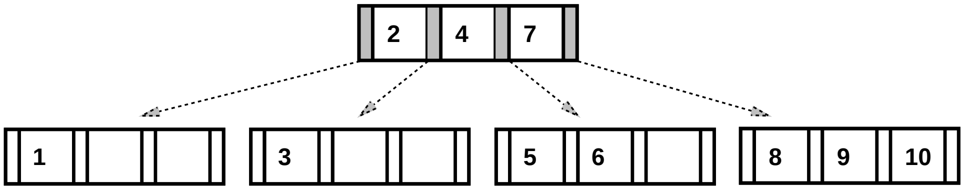
- Insert ‘4’:
  - Find correct leaf ⇒ the ‘3-5’ node.
  - Space available ⇒ insert.



- Insert ‘6’:
  - Find correct leaf ⇒ the ‘3-4-5’ node.
  - Node full ⇒ split (median element is ‘4’)
  - Add new element to correct child (the ‘5’ child, here):

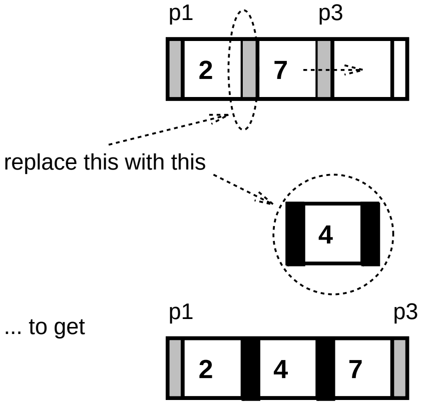


  - Insert ‘4’ into parent:

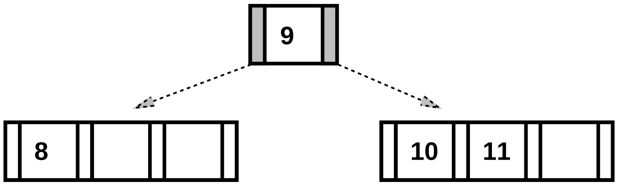


NOTE:

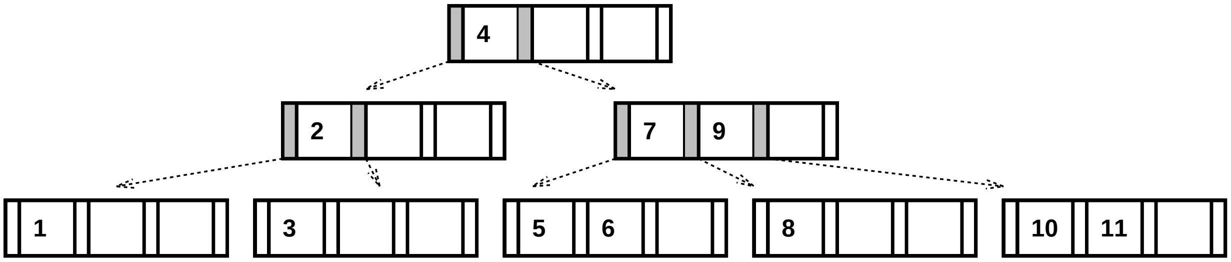
- The element ‘7’ and everything to the right of it (including pointers) are shifted to the right.
- The (tree) pointer between ‘2’ and ‘7’ is *overwritten* by ‘4’ and its two pointers.


p1
p3
2
7
replace this with this
4
... to get
p1
p3
2
4
7

- Insert ‘11’:
  - Search for correct leaf ⇒ ‘8-9-10’ block.
  - Block is full ⇒ split needed (median element is ‘9‘).
  - Add new element in correct child ⇒ the ‘10’ block:



  - Insert ‘9’ into parent node (the ‘2-4-7’ block)
    ⇒ ‘2-4-7’ needs to be split (with median ‘4’)
    ⇒ create new root with ‘4’.
  - Final tree:



- What is the complexity of insertion?
  - Suppose height of tree is h.
  - Need to search for correct leaf (in-order traversal)
    $\Rightarrow$ at most h blocks accessed for search.
  - In the worst case, a split propagates to root
    $\Rightarrow$ h blocks accessed going back up.
  - Thus, $2h$ old blocks read, h old blocks written, h new blocks written.
    $\Rightarrow$ $4h$ disk accesses (worst case)
    $\Rightarrow$ $O(\log_m n)$ for insert.

  NOTE:

  - In practice, some blocks are likely to remain in memory.
  - Typically, m is large
    $\Rightarrow$ height is small.
  - For example, suppose $m = 100$
    $\Rightarrow$ if $h = 4$, $2m^{h-1} = 2,000,000$
    $\Rightarrow$ 2 million blocks is a lot of data!
  - Access times are $O(h)$.

4.20 Implementing a B-Tree

- Implementation issues:
  - How to store data and pointers in blocks?
  - Designing algorithms for insertion and search.
- Recall: B-tree blocks contain
  - key values (at most $2m-1$)
  - data pointers (at most $2m-1$)
  - tree pointers (at most $2m$)

  It is useful to store additional information:
  - A boolean indicating whether the block is a leaf.
  - The number of entries (key-values) in a block.

  Note: If a block has k key values, then it has $k+1$ tree pointers.
- If the degree m is known in advance, we can use a C-like `struct` for a B-tree block with fields, e.g.,

```
struct btree_node {                  // M = degree
  int num_entries;                   // Number of entries
  int leaf;                          // Leaf or interior node?
  char value [KEYSIZE][2*M-1];       // Values
  int blocknum [2*M-1];              // Together, blocknum and
  int tuplenum [2*M-1];              // tuplenum constitute a datapoi
  int treep [2*M];                   // One additional treepointer
};
```

Now, typically, the system reads a block into memory and returns its address. In C, a *cast* may be used to extract fields:

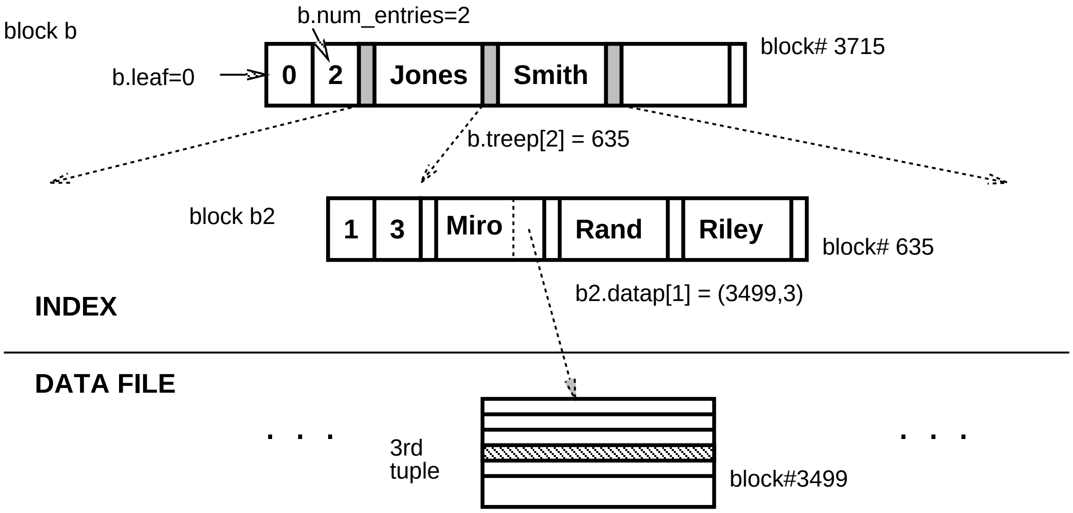
```
b = (btree_node *) Disk_Readblock (blocknum);
if (b->leaf) {
  ... etc
```

Casting will not be shown in the pseudocode that follows.

The following functions will be used to handle I/O in the pseudocode:

- Disk-Readblock (bnum)
  – This function reads block bnum into memory and returns the address.
- Disk-Newblock ()
  – Creates a new block on disk and returns a block number.
  – A separate Disk-Readblock is needed to read the block into memory.
- Disk-Writeblock (bnum) – Writes block bnum to disk.

Example:



- Typical function specifications for creating, searching and inserting:

  Creating a B-tree:

  ```
  btree_create (tuplesize, keysize, keyoffset)
  ```

  - `tuplesize` is the length (in bytes) of the data tuple.
  - `keysize` is the length of the key field.
  - `keyoffset` is the offset (in bytes) of the key field.
  - The function initializes the root block and stores (in private data structures) the tuplesize etc.
  - Most often, a *file identifier* is returned:

    ```
    fd = btree_create (tuplesize, keysize, keyoffset)
    ```

    to be used later in searching or insertion:

    ```
    btree_search (fd, key)
    ```

  Searching a B-tree:

  ```
  tuple = btree_search (key)
  ```

  - `key` is a particular key value (e.g., the character string ‘Smith’ for a NAME field).
  - The function typically returns the entire tuple (if found) or NULL.
    ⇒ the function retrieves the tuple from the data file.

  Inserting a tuple:

  ```
  btree_insert (tuple)
  ```

  - `tuple` is the tuple to be inserted.
  - Often, such functions return the data block (of the heapfile) in which the tuple was inserted.

- Pseudocode for **btree_create**:
  (File identifier has been left out of the pseudocode)

```
Algorithm:    Btree-Create (tuplesize, keysize, keyoffset)

Input: sizes of tuple and key, offset of key within tuple.
Output: root block, written to disk.
    Store size and offset information;
    Create heapfile;
    Compute max_entries (or degree) in B-tree node;
    root_blknum := Disk-Newblock(); // Get a block number
    root := Disk-Readblock (root_blknum); // Read it in
    root→leaf := true;
    root→num_entries := 0;
    Disk-Writeblock (root_blknum);
    return;
```

- Pseudocode for **btree_search**:

```
Algorithm:    Btree-Search (key)

Input: key value to search for.
Output: the corresponding tuple, if found, else null.
    $T$ := Btree-Recursive-Search (root_blknum, key);
    // Here, T is a data pointer into the heapfile
    if $T$ = null return null;
    $b'$ := Disk-Readblock ($T$.blocknum);
    $i$ := $T$.tuplenum;
    return $i$-th tuple in $b'$;
```

Algorithm: Btree-Recursive-Search (bnum, key)

Input: block number bnum and a key.
Output: datapointer if found, **null** otherwise.

```
b := Disk-Readblock (bnum);
i := 1;
while $i \leq n$ and $b$ →value[$i$] < key
   $i := i + 1$;
if $b$ →value[$i$] = key
   Stack-Push (bnum);
   return $b$ →datap[$i$];
endif;
// Otherwise, not found; dig deeper
Stack-Push (bnum); // Save bnum for insert
if $b$ →leaf = true // Can't go down ⇒ not found
   return null
if $i \leq b$ →num_entries // go left
   return Btree-Recursive-Search ($b$ →treep[$i$], key);
else // go right
   return Btree-Recursive-Search ($b$ →treep[$i + 1$], key);
```

- Pseudocode for **btree_insert**:

```
Algorithm: BTREE-INSERT ($x$)

Input: a tuple, $x$.
Output: Insertion of $x$ in data file and key($x$) in B-tree.
$x'$ := BTREE-SEARCH ($x$.key);
if $x' \neq$ null
    print 'Error: key exists';
    return;
endif
$T$ := HEAPFILE-INSERT ($x$);
    // $T$ is a datapointer, (block#, tuple#)
bnum := STACK-POP();
    // After search, stack has block number of correct leaf
BTREE-RECURSIVE-INSERT (bnum, $x$.key, T, 0, 0);
return;
```

Algorithm: BTREE-RECURSIVE-INSERT (bnum, key, T, leftblock, rightblock)

Input: block bnum, a key, datapointer T, left and right treepointers
Output: Insertion of data and datapointer into block bnum.

```
b := DISK-READBLOCK (bnum);
if b →num_entries < max_entries
    insert key, leftblock and rightblock into correct place in b;
    DISK-WRITEBLOCK (bnum);
    return;
endif;
// Otherwise, node needs to be split
median := (max_entries + 1) / 2;
median_key := b →value[median];
median_T := b →datap[median];
bnum2 := DISK-NEWBLOCK(); // New right block
b2 := DISK-READBLOCK (bnum2);
// Including new item, we have max_entries + 1 items;
keep items 1,...,(median-1) in current block, b;
place items (median+1),...,(max_entries+1) in b2;
DISK-WRITEBLOCK (bnum);
DISK-WRITEBLOCK (bnum2);
// Now insert median (and pointers bnum, bnum2) in parent
if bnum ≠ root_blknum
    parent_blknum := STACK-POP();
    BTREE-RECURSIVE-INSERT (parent_blknum, median_key, median_T,
        bnum, bnum2);
else
    CREATE-NEW-ROOT (median_key, median_T, bnum, bnum2);
return;
```

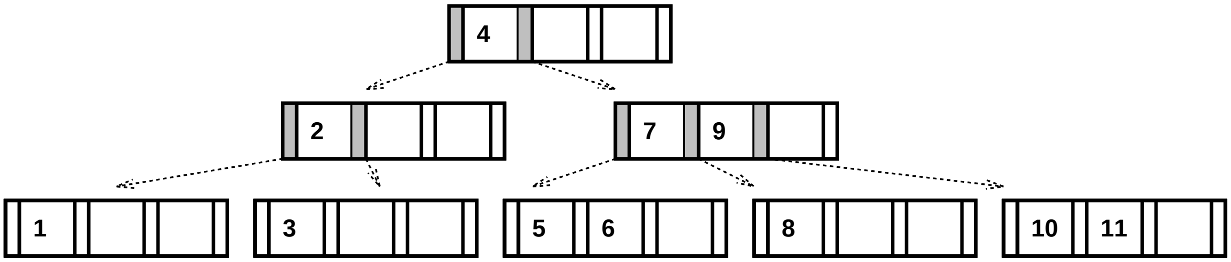
Algorithm: CREATE-NEW-ROOT (key, T, leftblock, rightblock)

Input: a key, a datapointer, left and right block numbers.
Output: a new root block, written to disk.

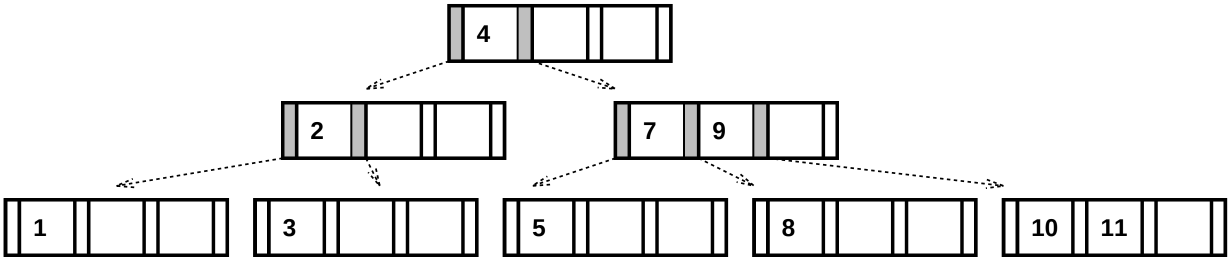
```
root_blknum := DISK-NEWBLOCK();
root := DISK-READBLOCK (root_blknum);
root→num_entries := 1;
root→value[1] := key;
root→datap[1] := $T$;
root→treep[1] := leftblock;
root→treep[2] := rightblock;
DISK-WRITEBLOCK (root_blknum);
return;
```

4.21 Deletion in a B-Tree

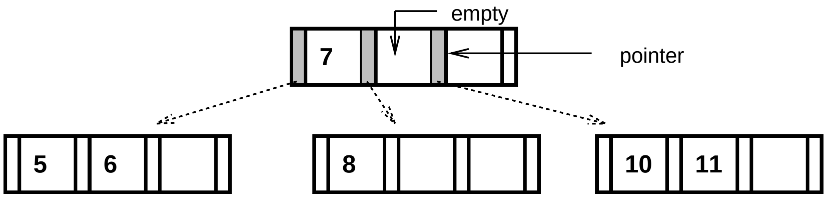
- Deletion is somewhat more complicated than insertion. To see why, consider:



  - Suppose we want to delete ‘6’
    ⇒ easy – simply delete ‘6’ from ‘5-6’ node:



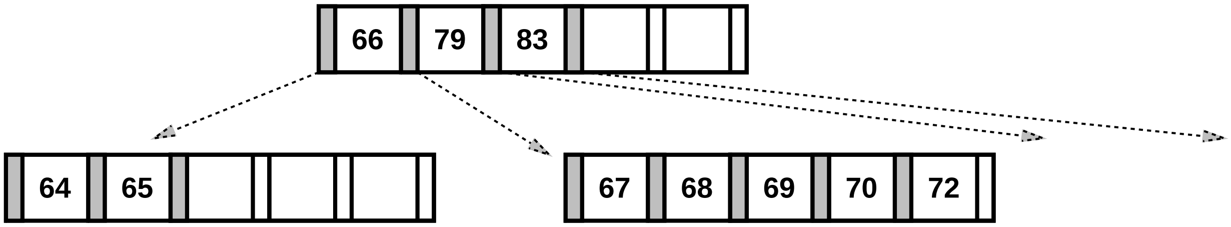
  - Suppose we want to delete ‘9’:
    - ∗ Cannot delete ‘9’ casually.
    - ∗ A search for ‘11’ would cause problems
      ⇒ problem is worse for large m.



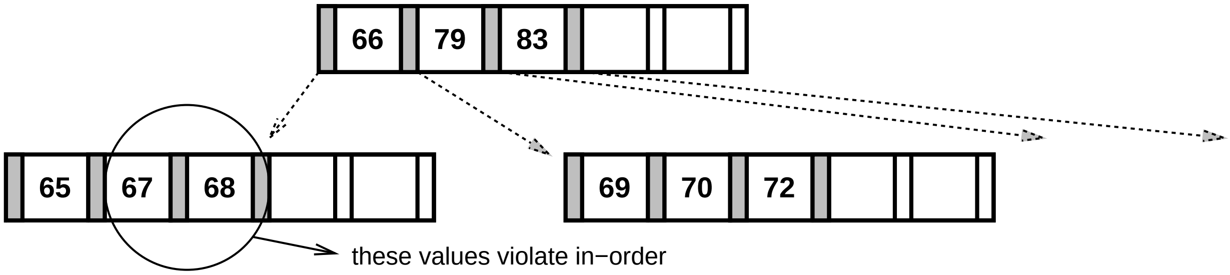
- Suppose we want to delete '2'
  ⇒ need to have at least $m - 1$ elements
  ⇒ node will *underflow*.

- Key ideas used in deletion:
  - *Rotation*: when underflow occurs, try to borrow key values from sibling via rotation.
  - *Merging*: if rotation cannot be done, merge siblings.
  - *Replacement*: when deleting a value from an interior node, replace it with another key value.

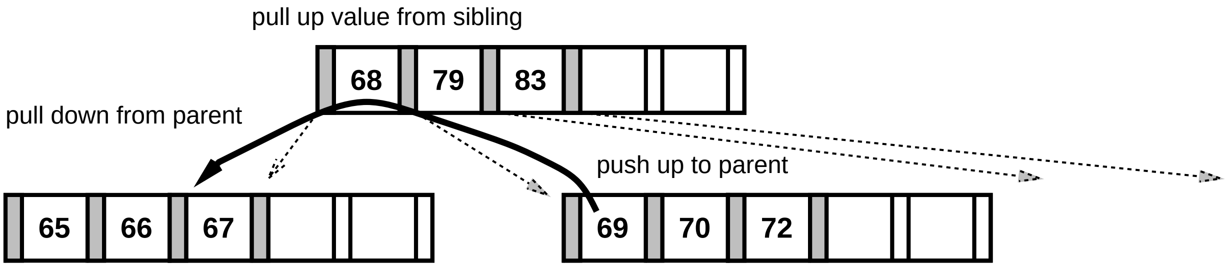
- Rotation:
  - Consider an example where $m = 3$
    ⇒ must have at least $m - 1 = 2$ values per node.



  - Deleting '64' violates the lower limit.
  - But, right sibling has enough values to re-distribute.
  - Values are re-distributed via rotation (including the parent).
  - Note: we cannot ignore parent value in rotation (since that would violate in-order).

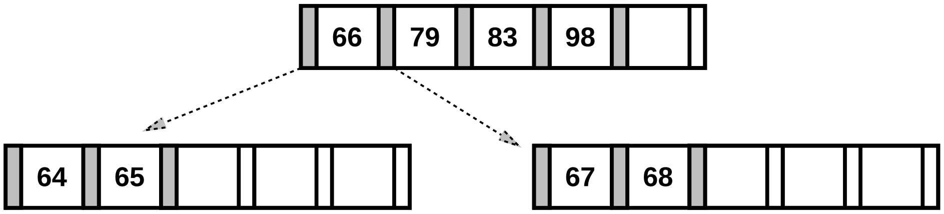


- Instead, parent value must be included in rotation.



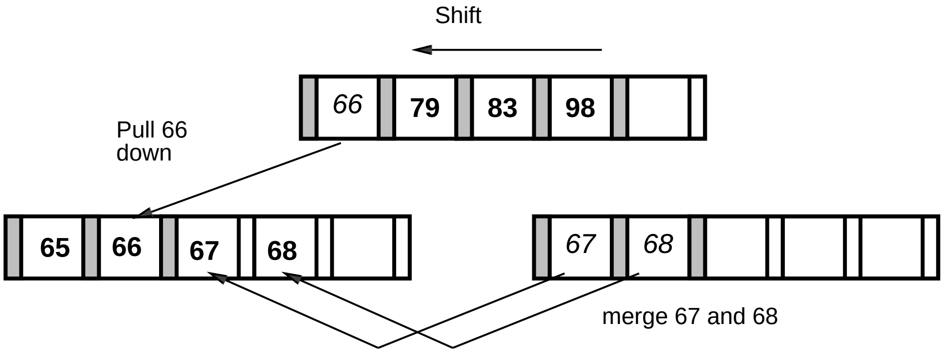
- NOTE: both data pointers and tree pointers need to be moved in a rotation.

- Merging:
  - Sometimes, you can't borrow from a sibling because the sibling doesn't have enough:

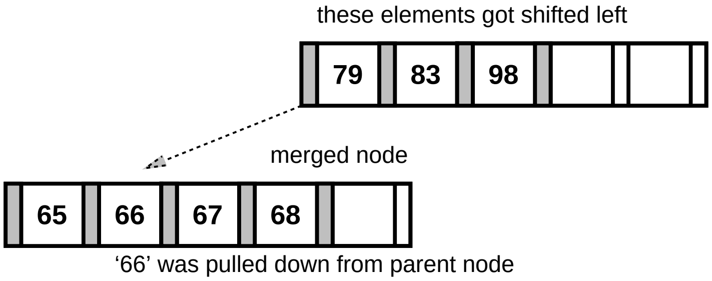


  - Here, rotation from right sibling would cause right sibling to underflow.
  - Key idea: since both siblings are 'borderline', they both have $m-1$ values
    $\Rightarrow$ together they have $2m-2$ values
    $\Rightarrow$ together with parent value they have $2m-1$ values
    $\Rightarrow$ can merge all of these into *a single node.*

– Method: pull ‘66’ down and merge siblings:



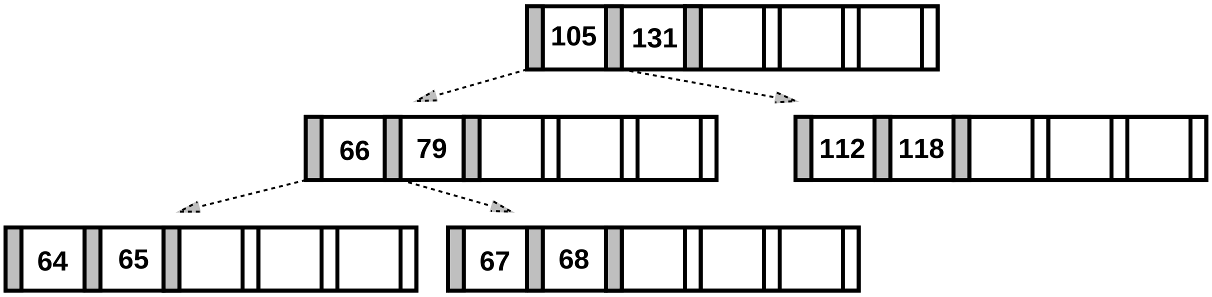
Result:


these elements got shifted left
79
83
98
merged node
65
66
67
68
‘66’ was pulled down from parent node

– But what if pulling down a value from the parent causes underflow at parent’s level?

⇒ balance or merge at parent’s level ... and so on, recursively.

– Example: delete ‘64’ from this tree:

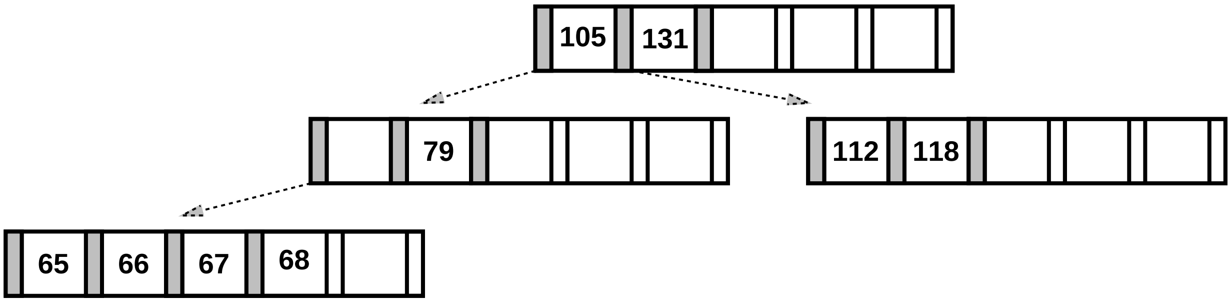


Here,

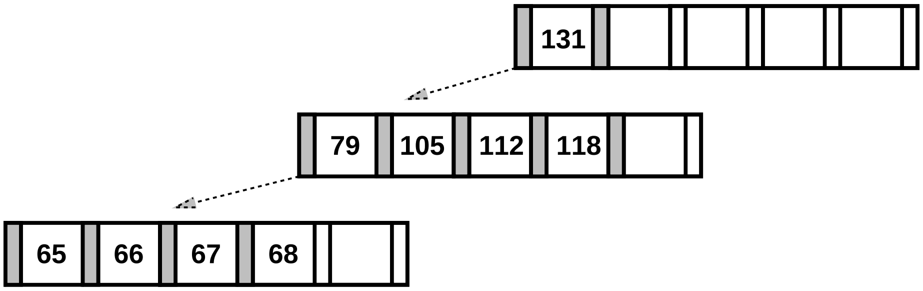
∗ Deleting ‘64’ causes underflow in ‘64-65’ node.

- ∗ Sibling is borderline
  - ⇒ pull down ‘66’ from parent
  - ⇒ causes underflow in parent node
  - ⇒ try to re-distribute among parent and parent’s sibling.
- ∗ Parent’s sibling is borderline
  - ⇒ must merge parent and parent’s sibling.

When we pull down ‘66’, we get:



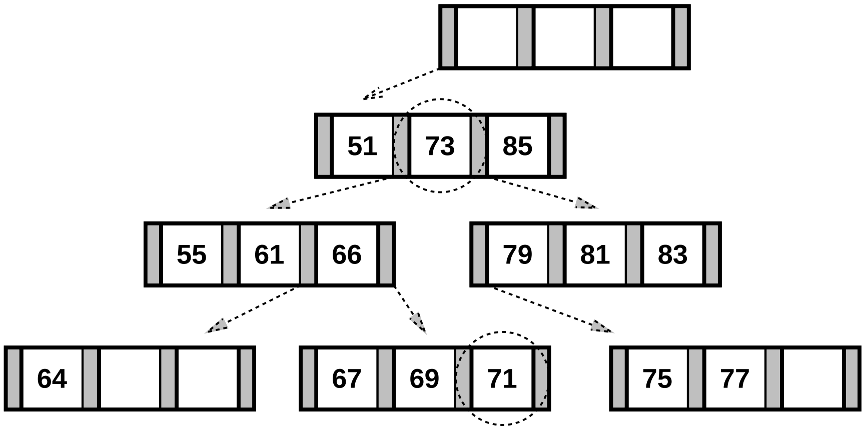
When we pull down ‘105’ we get:



NOTE: if ‘105’ was the only element in the root, we would have one less level.

NOTE: In the above example (and some that follow), the entire tree is not shown for lack of space.

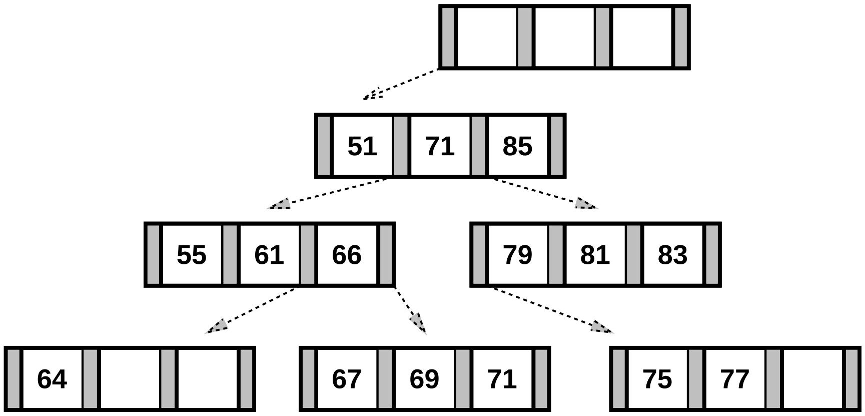
- Replacement:
  - Deletion of an interior node value requires replacing the value.
  - The value is replaced by the in-order *predecessor*.
  - Finding the in-order predecessor requires a search all the way down to the leaf level.
  - Example (with $m = 2$): Delete '73' from this B-tree



'71' is the in−order predecessor of '73'

  - The in-order predecessor of '73' is the largest element in the *left* subtree of '73'.

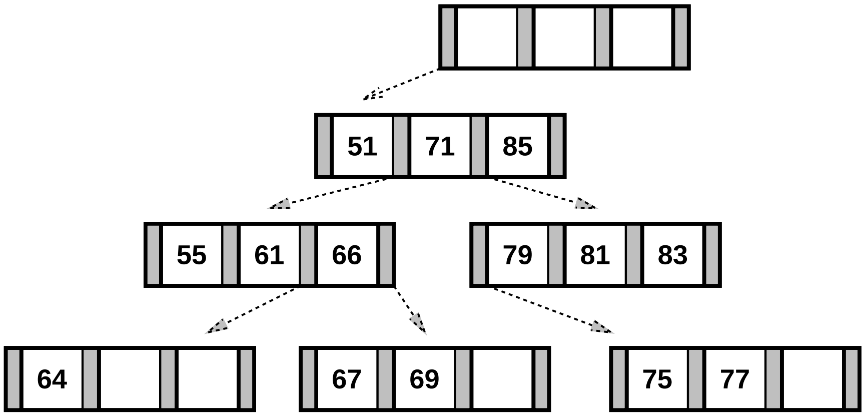
    ⇒ rightmost element in left subtree.
  - Note: an interior value always has a left subtree.
  - Replace '73' with '71'.



original '71' still in place

- Note: after replacement, in-order is maintained.

– Finally, do a regular delete of '71':



– Note: in some cases, the final delete could recursively work its way up to the root. But it does not use *replacement* and so must finally end.

- At last we see why the underflow restriction is $m - 1$ (as opposed to something higher)

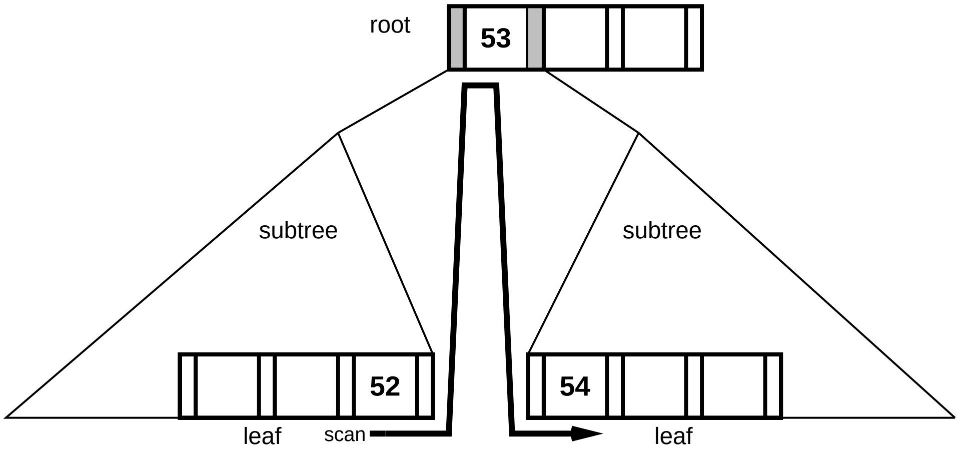
  ⇒ if the restriction were higher (such as 70% full), then how would we merge two nodes that are 70% full?

  (It can be done, but it's harder).

- We will not cover **delete** any further. For additional details, see the literature.

4.22 B+-Trees: Introduction

- Recall the following about a B-Tree:
  Index values occur all over the tree
  ⇒ a **range search** or sorted **scan** can be quite expensive.

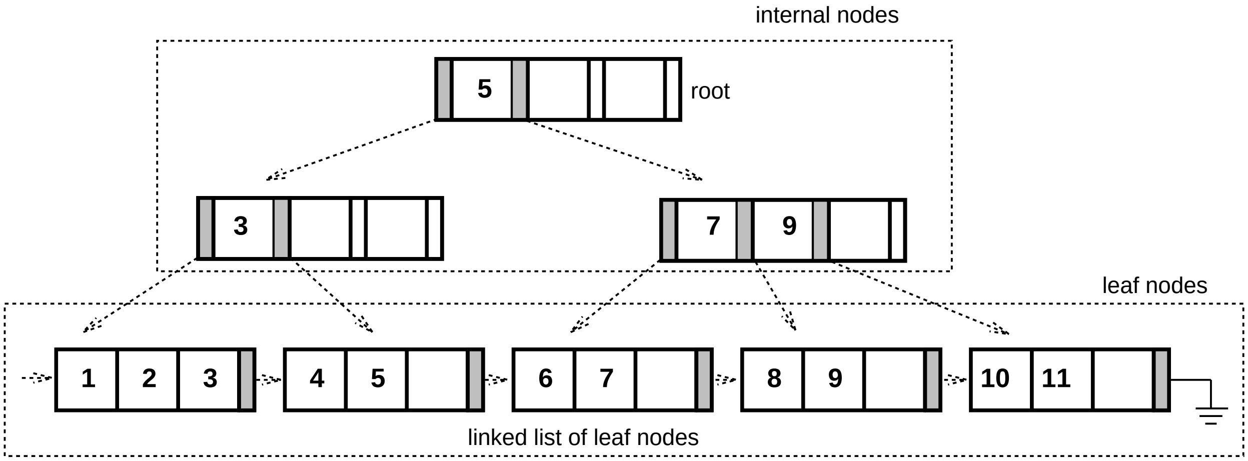


- A B+-tree overcomes this problem (at the cost of allowing duplicate entries).

- Key ideas in a B+-tree:
  - Similar structure to that of a B-tree.
  - Distinguish between *internal* and *leaf* nodes.
  - Leaf nodes contain *all* search keys inserted into tree.
  - Leaf nodes are serially connected by a linked list in *sorted order.*
  - Each *leaf* node:
    - ∗ has no tree pointers
    - ∗ has only search keys and data pointers
    - ∗ has linked list pointer

- Each *internal* node:
  - ∗ has search keys
  - ∗ has tree pointers
  - ∗ has no data pointers
- Search keys in internal nodes are used only for *navigation*.
- Search keys in internal nodes are repeated in the leaves.
- Every search key does not occur as an internal value.
- Which search keys get to be in internal nodes as well?
  ⇒ depends on what's wrought about by insertions/deletions.

For example:



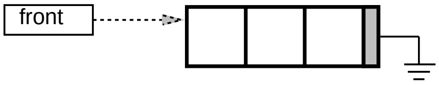
- Constraints on node contents for a B+-tree of degree m:
  - Each internal or leaf node must contain at least $m - 1$ keys.
  - Each internal or leaf node can contain at most $2m - 1$ keys.
  - Each internal node can contain at most $2m$ pointers.
  - Each internal node must contain one more pointer than the number of keys.
  - Each leaf node must contain as many data pointers as there are keys in the node.

NOTE:

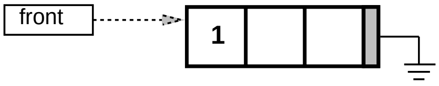
- In a full node, the m-th value is the *median* or *middle* value.
- In practice, tree pointers can be smaller than data pointers
  $\Rightarrow$ internal and leaf nodes can have different *degree*
  $\Rightarrow$ usually internal nodes can pack more keys
  $\Rightarrow$ possibly fewer levels than a corresponding B-tree.
- For simplicity, this presentation will use the same degree for both.
- As with B-trees, the above definitions can be extended to trees with even numbers of keys.

4.23 Insertion in a B+-Tree

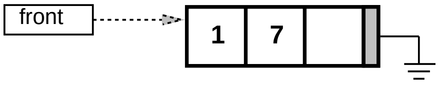
- Key ideas:
  - Similar to a B-tree.
  - Search for correct leaf using *in-order* search.
  - Must insert in appropriate leaf.
  - If leaf is full, split leaf and insert a *copy* of the median element at the next level.
- We will learn insertion via an example:
  - Consider the case $m = 2$.
  - The key values are integers.
  - Insert the following key values in order: 1,7,8,10,9,3,2,5,4,6,11
- Initially: Create (empty) leaf node:



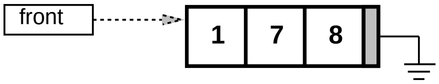
- Insert '1':
  - Space available ⇒ insert.



- Insert '7':
  - Space available ⇒ insert (in sort-order).

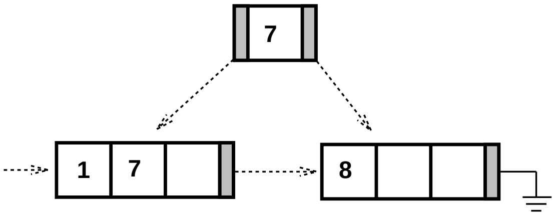


- Insert ‘8’:
  - Space available ⇒ insert (in sort-order).



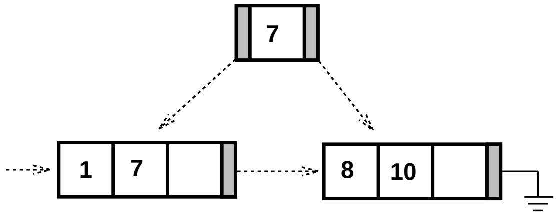
- Insert ‘10’:
  - Node is full ⇒ must split.
  - Median element is ‘7’.
  - Bump up a *copy* of median (‘7’) to next level.
  - Split nodes are children (left and right) of median element.
  - New element (‘10’) is added in appropriate split node.

Step 1: Split node:

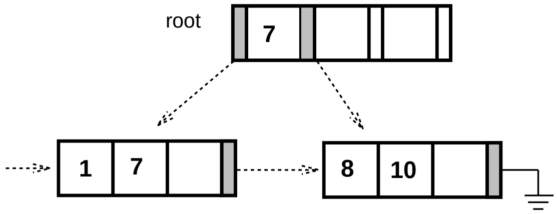


NOTE: ‘7’ remains in left child ⇒ search must use “≤” (to reach ‘7’ in leaf).

Step 2: Add new value (‘10’):



Step 3: Insert median element (with its left and right pointers) at level above.
⇒ in this case, create new root.

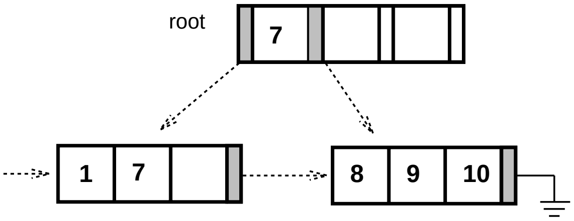


As in a B-tree, the general principle behind insertion is now apparent:

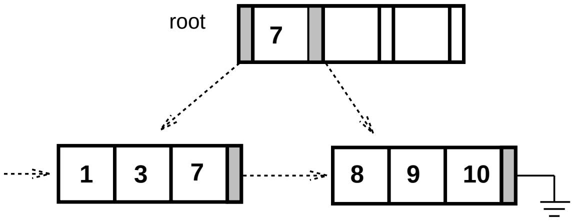
- First, find the correct leaf for insertion.
  ⇒ use in-order to navigate to correct leaf.
- Check if already present in leaf
  ⇒ if so, then insertion is a duplicate ⇒ halt.
- If space is available, insert into leaf (maintaining sort order).
- Else, split leaf and 'push up' a *copy* of the median element
  ⇒ insert median element into old leaf's parent.
- Add new item in left or right child (according to sort order).
- In pushing up median element from a leaf, *create* left and right pointers.
- In pushing up median element from an internal node, also move up left and right pointers.
- If parent is full, split parent,..., etc, recursively.

- Insert '9':

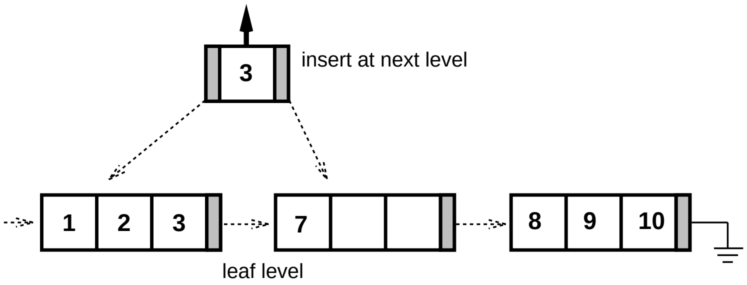
- Search for correct leaf ⇒ the ‘8-10’ node.
- Space available ⇒ insert.



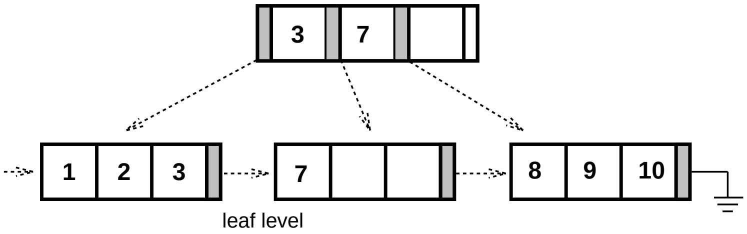
- Insert '3':
  - Search for correct leaf ⇒ the '1-7' node.
  - Space available ⇒ insert.



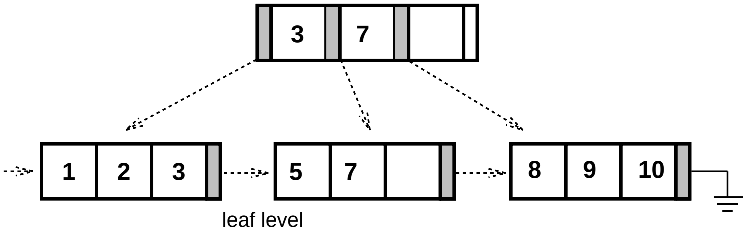
- Insert '2':
  - Search for correct leaf ⇒ '1-3-7' block.
  - Node is full ⇒ split required (median is '3'):



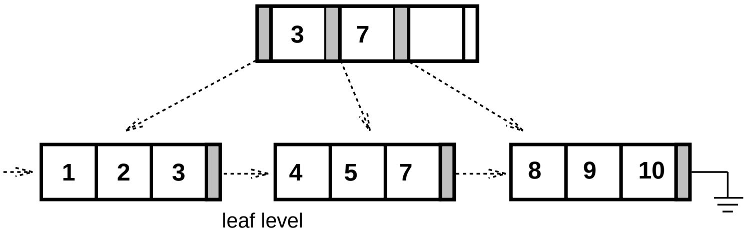
  - Insert '3' into parent (the root, in this case):



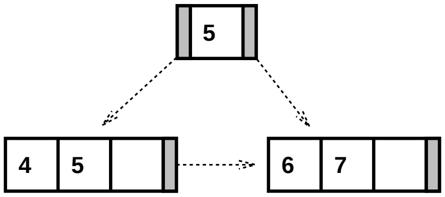
- Insert ‘5’:
  - Search for correct leaf ⇒ the ‘7’ node.
  - Space available ⇒ insert.



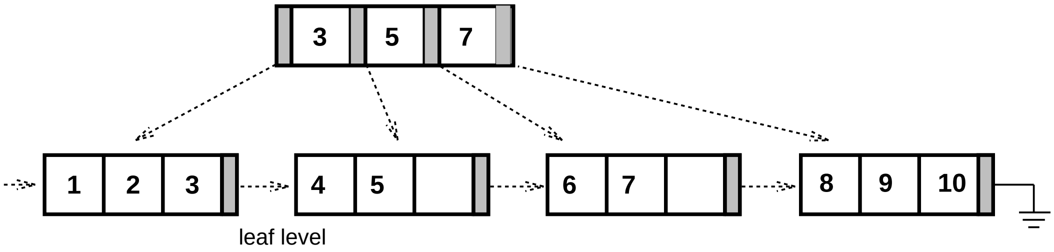
- Insert ‘4’:
  - Search for correct leaf ⇒ the ‘5-7’ node.
  - Space available ⇒ insert.



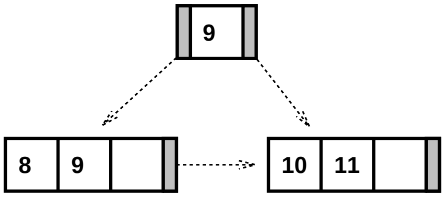
- Insert ‘6’:
    - Search for correct leaf ⇒ the ‘4-5-7’ node.
    - Node is full ⇒ split (median is ‘5’).
    - Add new element to correct child (‘6-7’ block):



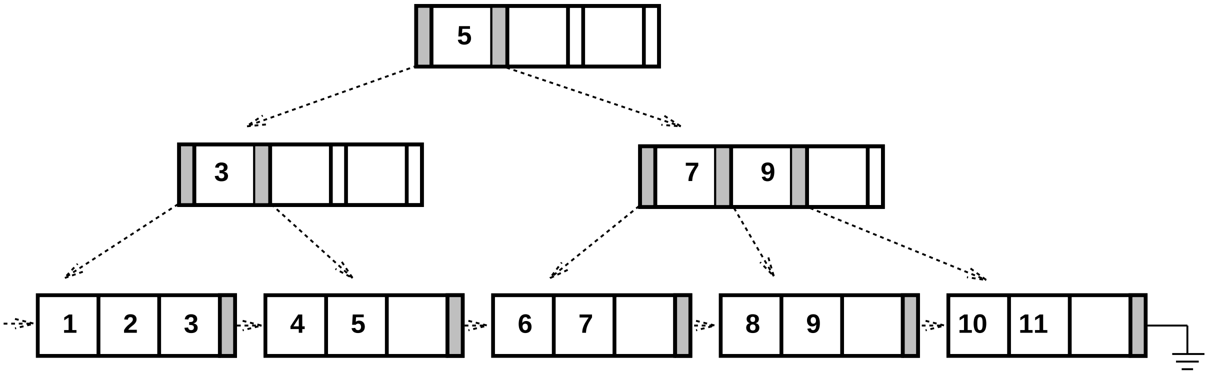
    - Insert ‘5’ into parent:



- Insert ‘11’:
  - Search for correct leaf ⇒ the ‘8-9-10’ block.
  - Block full ⇒ split (median is ‘9’).
  - Add new element in correct child (the ‘10’ block):



  - Insert ‘9’ into parent.
  - Node full ⇒ split (Median is ‘5’).
  - Insert ‘9’ in correct child ⇒ the ‘7’ block.
  - Create new root (‘5’).
  - Final tree:



- NOTE:
  - When a leaf gets split, a *copy* of the median gets pushed up to the next level.
  - The median itself stays in the *left* child (by convention).
    ⇒ searching must use "$\leq$".
  - When an internal node gets split, the median itself gets pushed up.
- *Searching* a B+-tree is almost identical to searching a B-Tree, except:
  - Use "$\leq$" when searching.
  - Must traverse down to leaf to obtain data pointer.
- What was the whole point about a B+-tree?
  ⇒ range search is fast!

  To search for search keys in the range 4-7:
  - Locate the lower limit ('4') via search.
  - Traverse linked list until you reach ('7').
    ⇒ *optimal* after lower limit is found.
- Complexity of B+-tree operations:
  - Identical to B-tree: depends on tree-height.
  - Tree height is $O(\log_m n)$.
  - Search and insertion (and deletion) are $O(\log_m n)$.
  - Range search costs $O(k + \log_m n)$ where k is the number of blocks containing the desired range.
- Pseudocode: analogous to the pseudocode for the B-tree, the following functions are defined:
  - B+tree-Create (tuplesize, keysize, keyoffset).
  - B+Tree-Search (key).
  - B+tree-Recursive-Search (b, key).

- B+tree-Insert (tuple).
- B+tree-Recursive-Insert (b, key, T, leftblock, rightblock).
- B+tree-Create-New-Root (key, T, leftblock, rightblock).

Note that leaf-nodes will have `next`-pointers to link them.

```
Algorithm:   B+tree-Create (tuplesize, keysize, keyoffset)


Input: sizes of tuple and key, offset of key within tuple.
Output: root block, written to disk.
    Store size and offset information;
    Create heapfile;
    Compute max_entries (or degree) in leaf and interior node;
    root_blknum := Disk-Newblock();
    root := Disk-Readblock (root_blknum);
    root→leaf := true;
    root→num_entries := 0;
    Disk-Writeblock (root_blknum);
    return;
```

Algorithm: B+tree-Search (key)

Input: key value to search for.

Output: the corresponding tuple, if found, else null.

```
$T$ := B+tree-Recursive-Search (root_blknum, key);
    // Here, T is a data pointer into the heapfile
if $T$ = null return null;
$b'$ := Disk-Readblock ($T$.blocknum);
$i$ := $T$.tuplenum;
return $i$-th tuple in $b'$;
```

Algorithm: B+TREE-RECURSIVE-SEARCH (bnum, key)

Input: block number bnum and a key.
Output: datapointer if found, **null** otherwise.

```
$b$ := DISK-READBLOCK (bnum);
$i$ := 1;
while $i \leq n$ and $b \rightarrow$value[$i$] $<$ key
    $i$ := $i + 1$;
if $b \rightarrow$leaf = true
    if $b \rightarrow$value[$i$] = key
        STACK-PUSH ($b$);
        return $b \rightarrow$datap[$i$];
    else // Not found at leaf level
        return null;
endif;
// Not a leaf ⇒ must search in correct subtree
STACK-PUSH (bnum); // Save bnum for insert
if $i \leq b \rightarrow$num_entries // go left
    return B+TREE-RECURSIVE-SEARCH ($b \rightarrow$treep[$i$], key);
else // go right
    return B+TREE-RECURSIVE-SEARCH ($b \rightarrow$treep[$i + 1$], key);
```

```
Algorithm:   B+tree-Insert ($x$)


Input: a tuple, $x$.
Output: Insertion of $x$ in data file and key($x$) in B-tree.
$x'$ := B+tree-Search ($x$.key);
if $x' \neq$ null
   print ‘Error: key exists’;
   return;
endif
$T$ := Heapfile-Insert ($x$);
   // $T$ is a datapointer, (block#, tuple#)
bnum := Stack-Pop();
   // After search, stack has block number of correct leaf
B+tree-Recursive-Insert (bnum, $x$.key, T, 0, 0);
return;
```

Algorithm: B+tree-Recursive-Insert (bnum, key, T, leftblock, rightblock)

Input: block bnum, a key, datapointer T, left and right treepointers
Output: Insertion of data and datapointer into block bnum.

```
$b$ := Disk-Readblock (bnum);
**if** $b$ $\rightarrow$leaf = true
    **if** $b$.num_entries < max_entries
        insert key, leftblock and rightblock into correct place in $b$;
        Disk-Writeblock ($b$);
        **return**;
    **endif**;
    // Else, we must split block
    bnum2 := Disk-Newblock(); // Get a new block number
    $b2$ := Disk-Readblock (bnum2) // Read it into memory
    newblk.next := $b$ $\rightarrow$next; // Adjust linked list
    $b$ $\rightarrow$next := newblk;
    median := (max_entries) / 2;
    median_key := $b$ $\rightarrow$value[median];
    Keep items 1,...,median in current block, $b$;
    Put items (median+1),...,max_entries in new block bnum2;
    $b2$ $\rightarrow$num_entries := $b$ $\rightarrow$num_entries - median;
    $b$ $\rightarrow$num_entries := median;
    **if** key $\leq$ median_key // Key goes into left block
        Put key in $b$ in appropriate place and increment num_entries;
    **else** // Key goes into the right block
        Put key in $b2$ in appropriate place and increment num_entries;
    Disk-Writeblock (bnum);
    Disk-Writeblock (bnum2);
    // Next, insert the median value at the next level
    **if** bnum $\neq$ root_blknum
        parent_blknum := Stack-Pop();
        B+tree-Recursive-Insert (parent_blknum, median_key, null,
                bnum, bnum2)
    **else**
        B+tree-Create-New-Root (median_key, bnum, bnum2);
    **return**;
**endif** // of "if $b$ $\rightarrow$leaf=true"
continued ...
```

Algorithm: B+tree-Recursive-Insert ... continued

```
// If b is not a leaf, we're inserting into an interior node
if $b$ →num_entries $<$ max_interior_entries
    Insert key in appropriate place and increment entries;
    Disk-Writeblock (bnum);
    return;
endif;
// Otherwise, we need to split this interior node
bnum2 := Disk-Newblock(); // Get a new block number
$b2$ := Disk-Readblock (bnum2); // Read it in
median := (max_entries) / 2;
median_key := $b$ →value[median];
Keep items 1,...,(median-1) in current block, $b$;
Put items (median+1),...,max_entries in $b2$;
Set num_entries for each block appropriately;
if key $\leq$ median_key // Put key in $b$
    Put key in $b$ in appropriate place and increment num_entries;
else
    Put key in $b2$ in appropriate place and increment num_entries;
Disk-Writeblock (bnum);
Disk-Writeblock (bnum2);
// Next, insert the median value at the next level
if bnum $\neq$ root_blknum
    parent_blknum := Stack-Pop();
    B+tree-Recursive-Insert (parent_blknum, median_key, null,
            bnum, bnum2)
else
    B+tree-Create-New-Root (median_key, bnum, bnum2);
return;
```

Note: the above presentation assumes that there is enough memory to hold both the block b and the new block newblk. The code can be modified to obviate this assumption.

Algorithm: B+TREE-CREATE-NEW-ROOT (key, leftblock, rightblock)

Input: a key, a datapointer, left and right block numbers.
Output: a new root block, written to disk.

```
root_blknum := DISK-NEWBLOCK();
root := DISK-READBLOCK (root_blknum);
root→num_entries := 1;
root→value[1] := key;
root→treep[1] := leftblock;
root→treep[2] := rightblock;
DISK-WRITEBLOCK (root_blknum);
return;
```

4.24 Deletion in a B+-Tree

- Since some values appear twice (leaf and internal nodes), is deletion in a B+-tree complex? Yes and no.
- Three types of deletion:

  Lazy deletion:

  - Delete only at leaf level.
  - Mark corresponding internal node value as deleted (if one exists).
  - Do not remove internal value
    ⇒ it can still be used for navigation.
  - Periodically rebuild tree and remove deleted internal values.

  Very lazy deletion:

  - Mark both internal and leaf values as deleted.
  - Periodically rebuild tree and remove both deleted internal and leaf values.

  Full deletion:

  - Remove both leaf and internal copies.

  Lazy deletion methods:

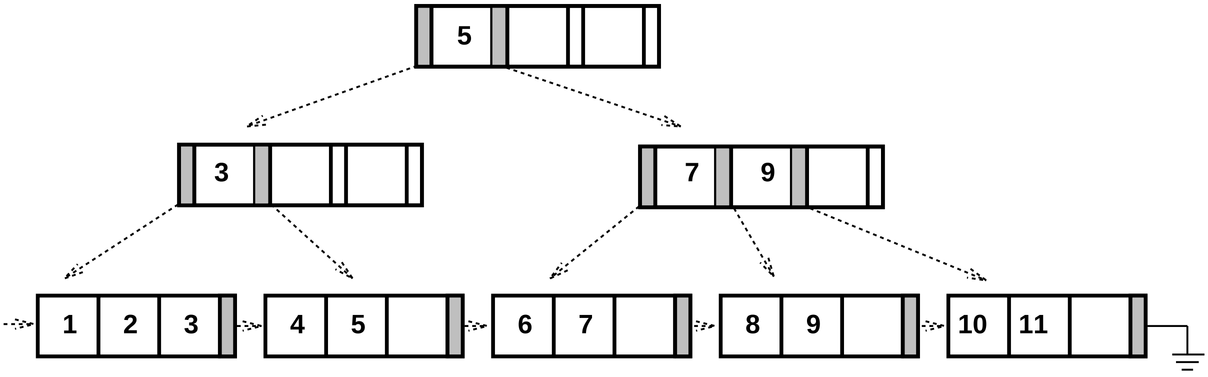
  - Easy to implement.
  - Tree can become large.
  - Have to be careful about re-insertions of deleted values (otherwise duplicates might exist in internal nodes).
  - Need to keep track for periodic re-building.

  Full deletion is harder to implement, but is space efficient. Each delete takes longer.

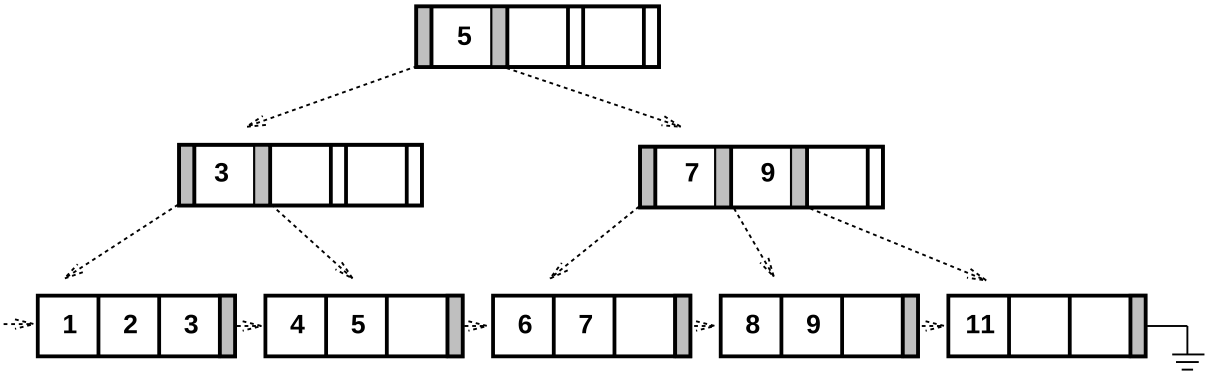
4.25 Lazy Deletion in a B+-Tree

- Consider this example (degree $m = 2$):

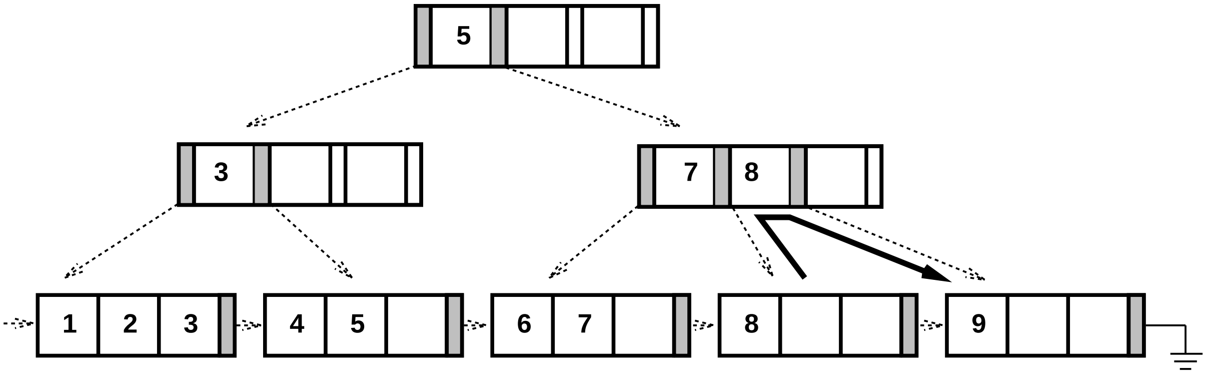
  Delete 10, 11, 5, 4, 2, 3, 8 from:



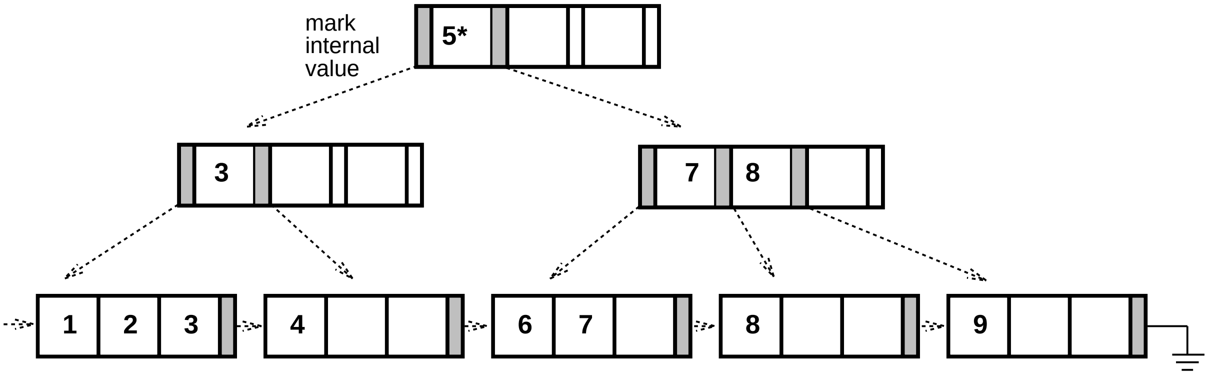
- Delete ‘10’:
  - Search for ‘10’.
  - ‘10’ is not found in an internal node ⇒ no marking needed.
  - Direct delete does not cause underflow ⇒ delete.



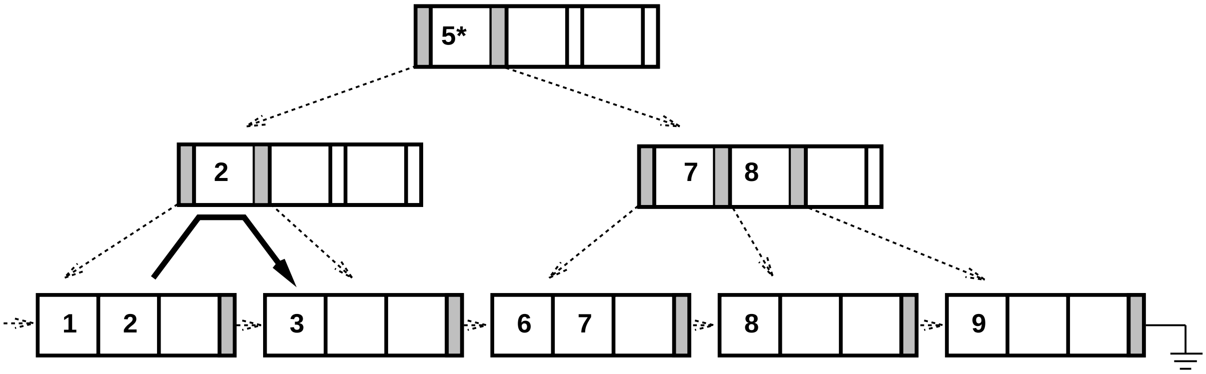
- Delete ‘11’:
  - Search for ‘11’.
  - ‘11’ does not occur as an internal value.
  - Deletion from leaf causes underflow ⇒ try to borrow from sibling.
  - To borrow, rotate from sibling
    ⇒ when moving ‘9’ over, copy of ‘8’ is placed into parent:



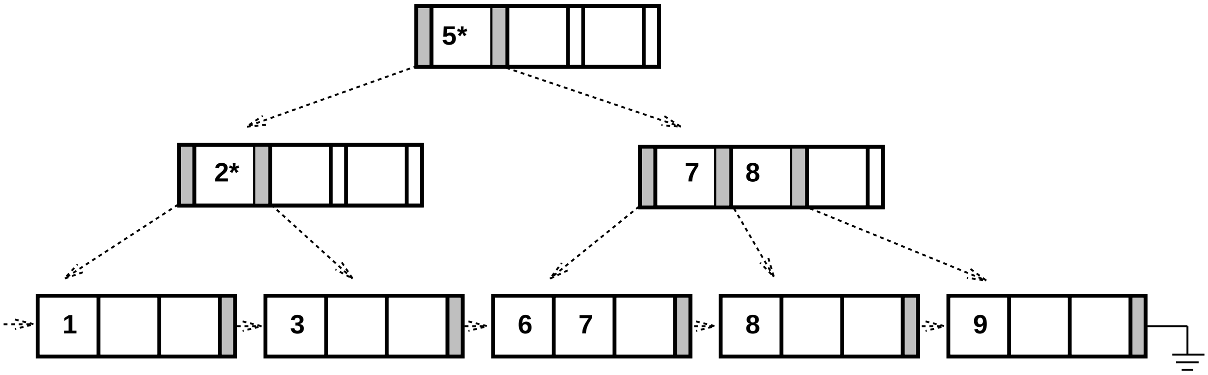
- Delete ‘5’:
  - Search for ‘5’.
  - ‘5’ occurs as internal value ⇒ mark internal value.
  - Deleting ‘5’ from leaf does not cause underflow.



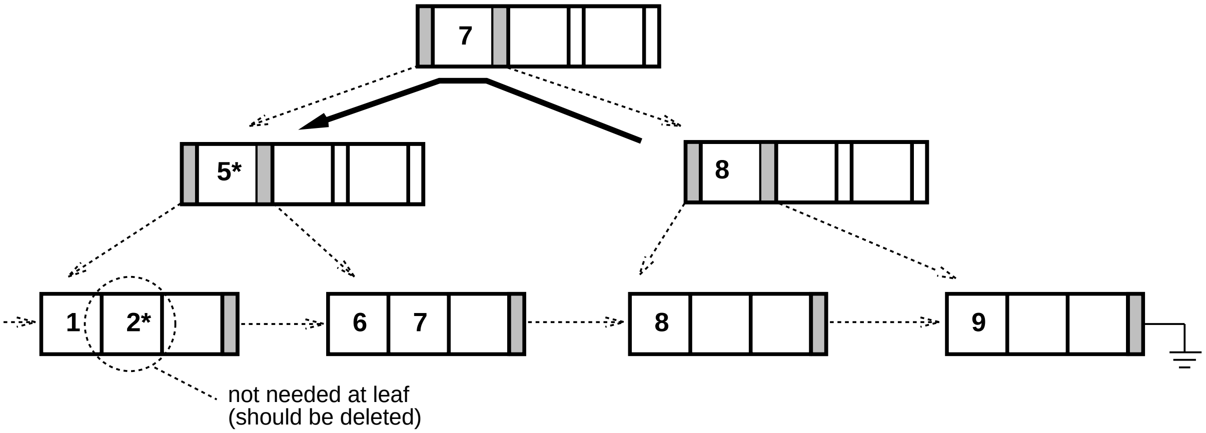
- Delete ‘4’:
  - Search for ‘4’.
  - ‘4’ does not occur as an internal value.
  - Deletion from leaf causes underflow ⇒ borrow from sibling.
  - Rotate from sibling:



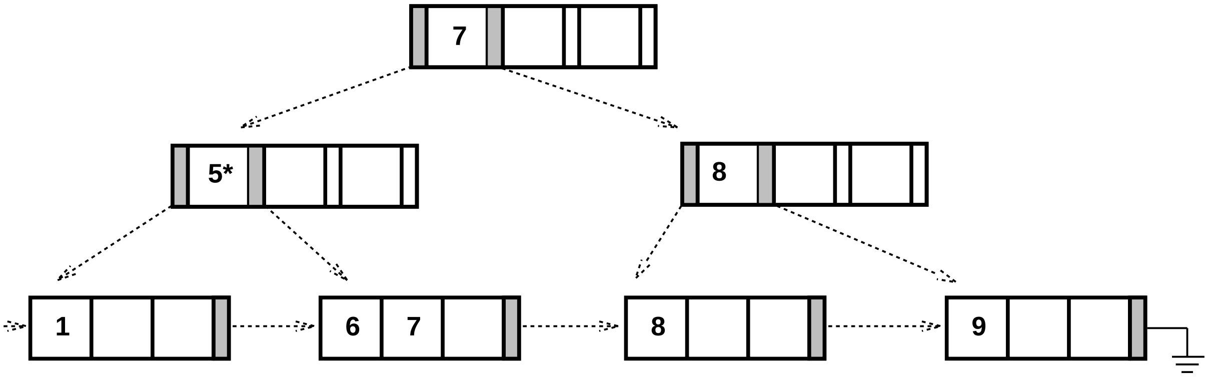
- Delete ‘2’:
  - Search for ‘2’.
  - ‘2’ occurs as an internal value ⇒ mark ‘2’.
  - Deletion from leaf does not cause overflow.



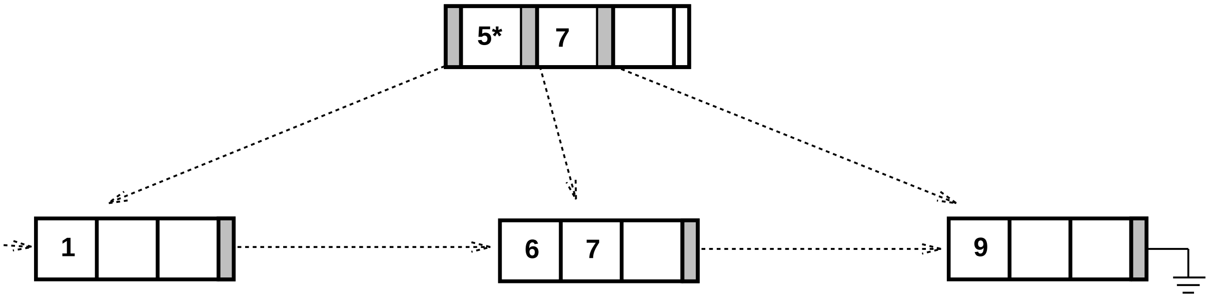
- Delete ‘3’:
  - Search for ‘3’.
  - ‘3’ does not occur as internal value.
  - Deletion from leaf causes underflow ⇒ borrow from sibling
  - Borrow from sibling not possible
    ⇒ Merge needed.
  - Pull-down from parent causes underflow at parent
    ⇒ borrow from parent’s sibling:



  - Marked values are not needed at leaf level (they have no data pointers)
    ⇒ they should be deleted.



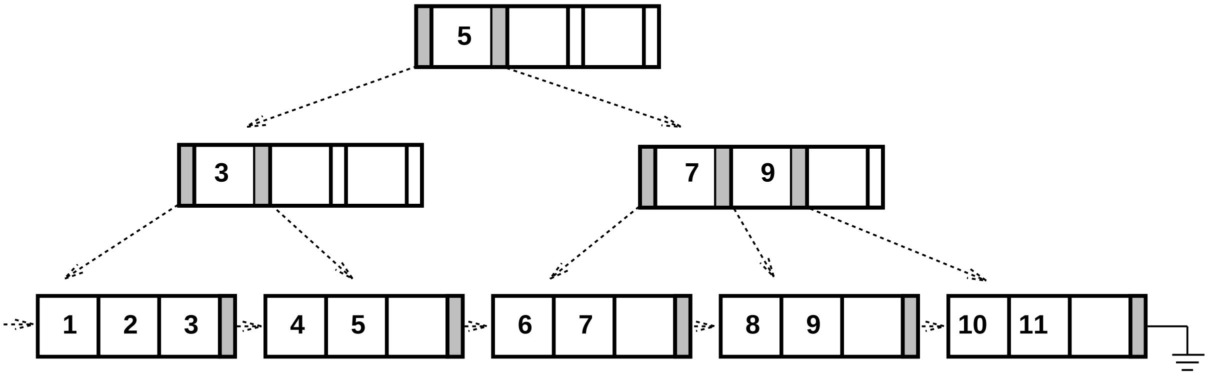
- Delete ‘8’:
  - Search for ‘8’.
  - ‘8’ occurs as an internal value ⇒ mark ‘8’.
  - Deletion from leaf causes underflow ⇒ try borrowing.
  - Borrow causes sibling underflow ⇒ pull down ‘8*’ from parent.
  - Underflow at parent’s level ⇒ try borrowing at parent’s level.
  - Borrow not possible at parent’s level ⇒ pull down ‘7’.
  - Final tree:



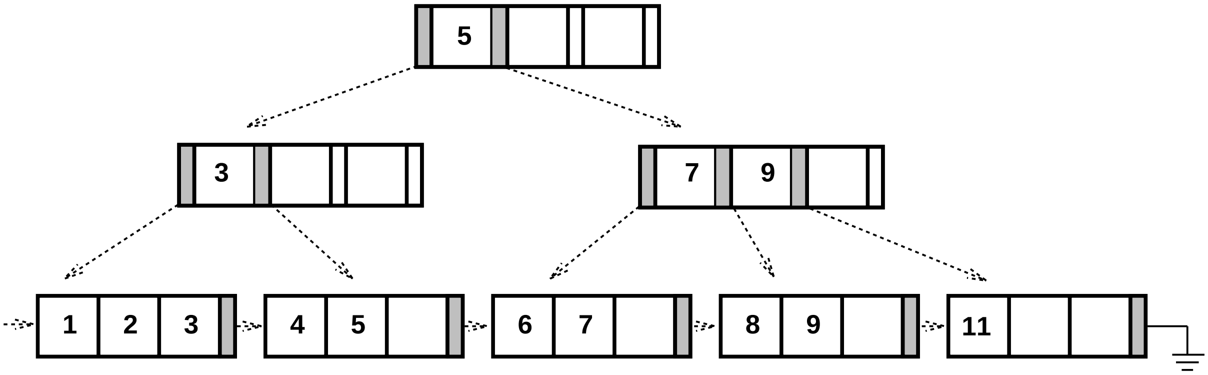
4.26 Full Deletion in a B+-Tree

- Consider this example (degree $m = 2$):

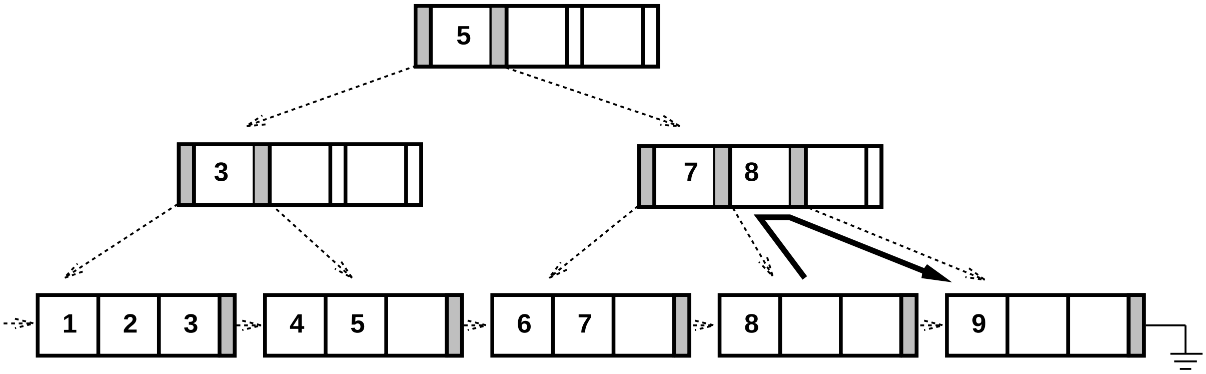
  Delete 10, 11, 5, 4, 2, 3, 8 from:



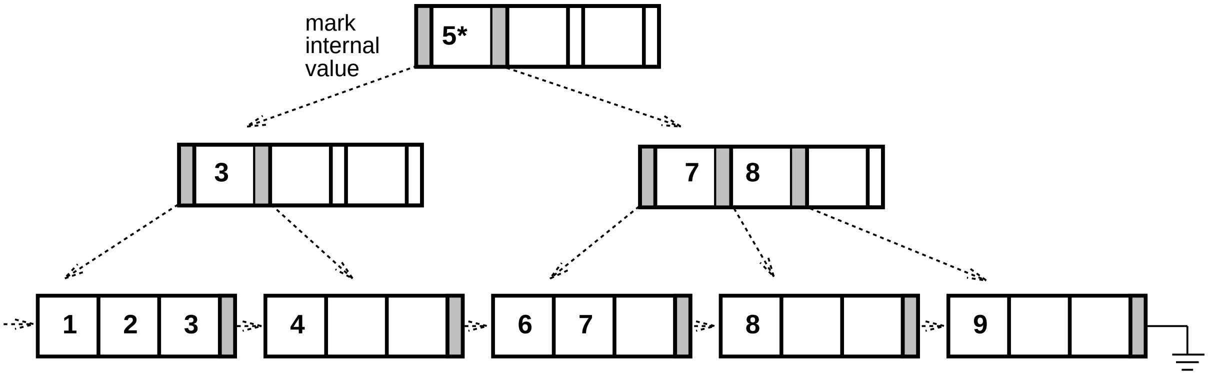
- Delete '10':
  - Search for '10'.
  - '10' does not occur as an internal value.
  - Deletion from leaf does not cause underflow.



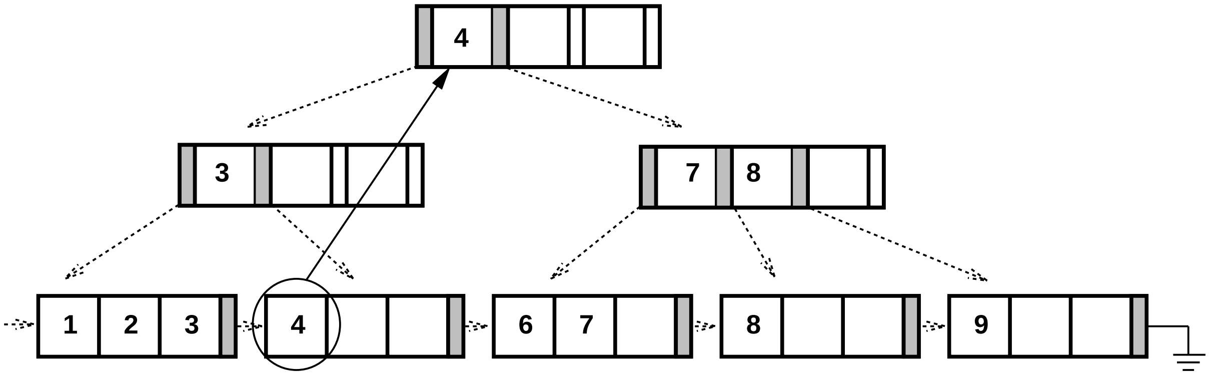
- Delete ‘11’:
  - Search for ‘11’.
  - ‘11’ does not occur as an internal value.
  - Deletion from leaf causes underflow ⇒ try to borrow from sibling.
  - To borrow, rotate from sibling
    ⇒ when moving ‘9’ over, copy of ‘8’ is placed into parent:



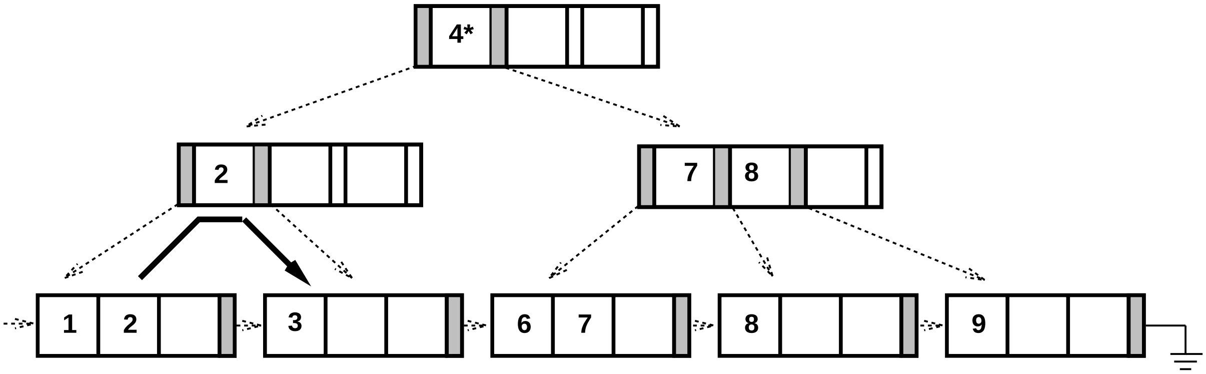
- Delete ‘5’:
  - Search for ‘5’.
  - ‘5’ occurs as internal value ⇒ mark internal value.
  - Deleting ‘5’ from leaf does not cause underflow.



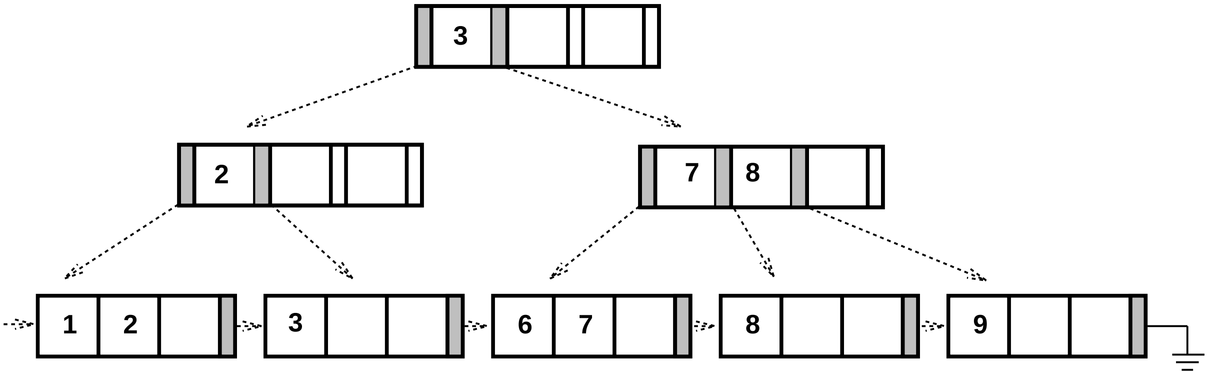
  - Replace internal value with copy of predecessor (rightmost leaf value in left subtree).



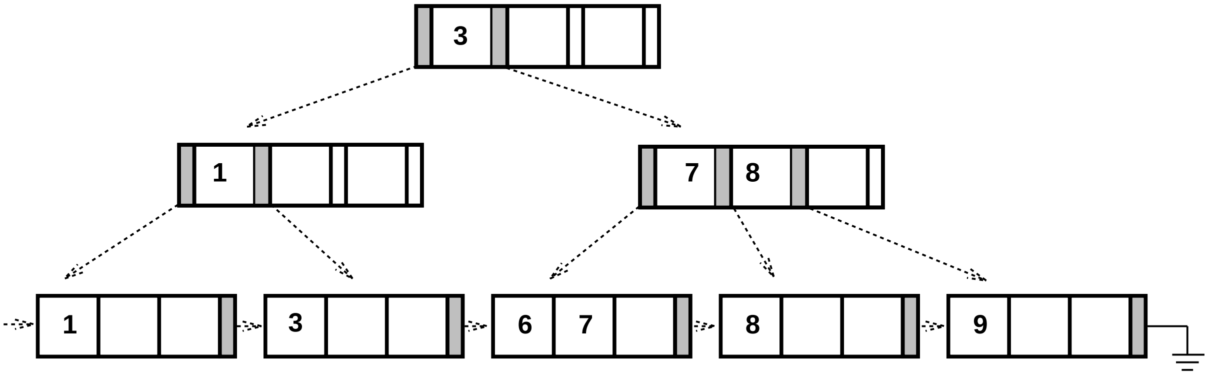
- Delete ‘4’:
  - Search for ‘4’.
  - ‘4’ occurs as an internal value ⇒ mark it.
  - Delete ‘4’ from leaf ⇒ need to rotate from sibling:



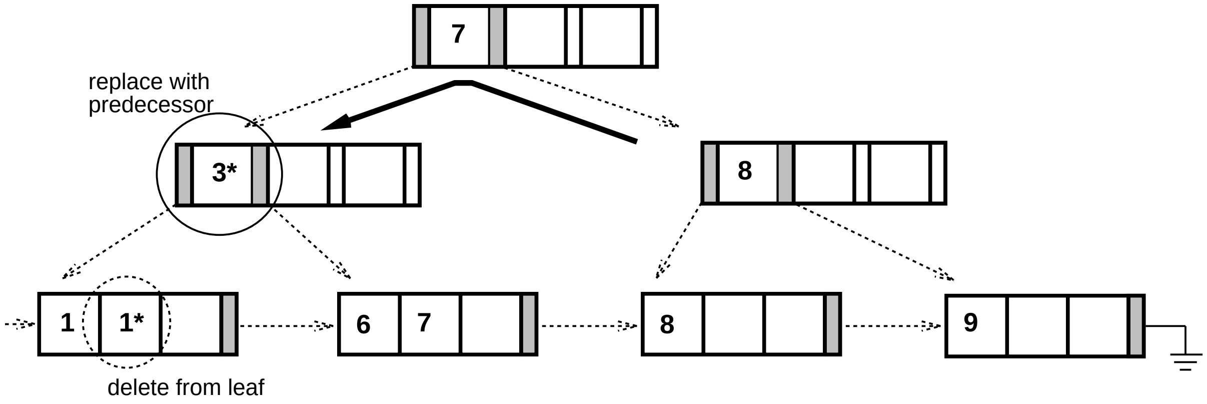
  - Replace internal value by predecessor:



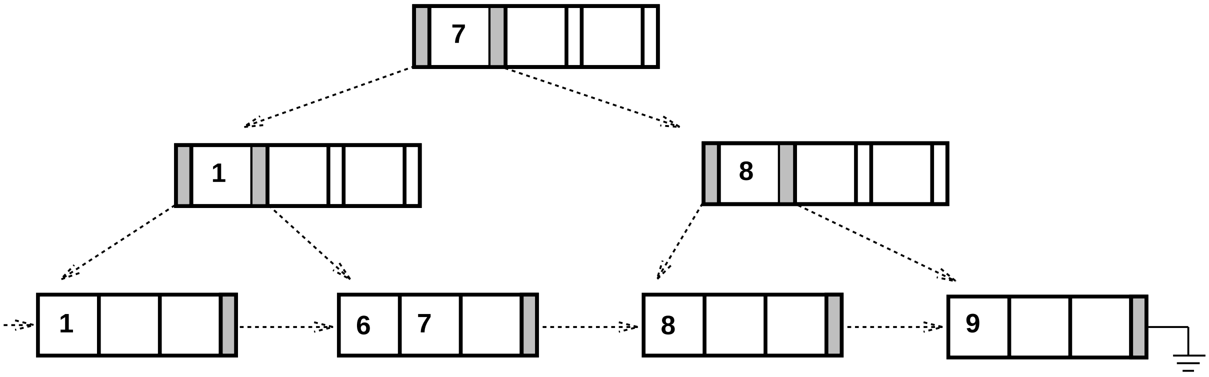
- Delete ‘2’:
  - Search for ‘2’.
  - ‘2’ occurs as an internal value ⇒ mark it.
  - Deletion from leaf does not cause underflow.
  - After leaf deletion, replace internal value by predecessor (‘1’):



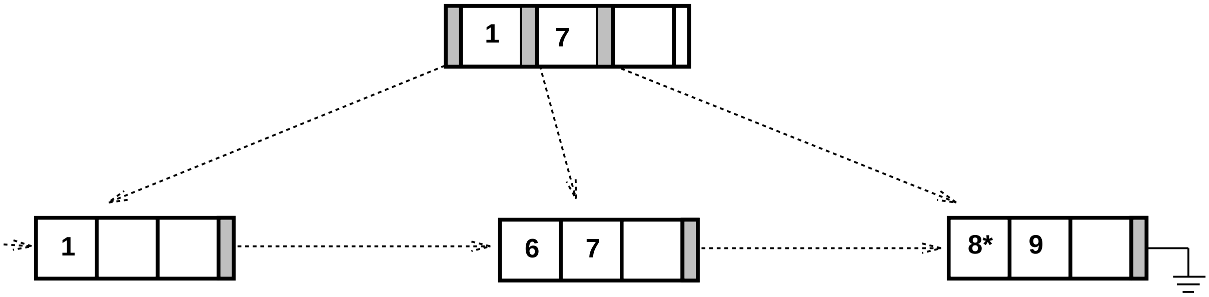
- Delete ‘3’:
  - Search for ‘3’.
  - ‘3’ occurs as an internal value ⇒ mark it.
  - Deletion from leaf causes underflow ⇒ try borrowing.
  - Borrowing causes underflow in sibling ⇒ pull down ‘1’.
  - Pulling down ‘1’ causes parental underflow ⇒ borrow at parent’s level.



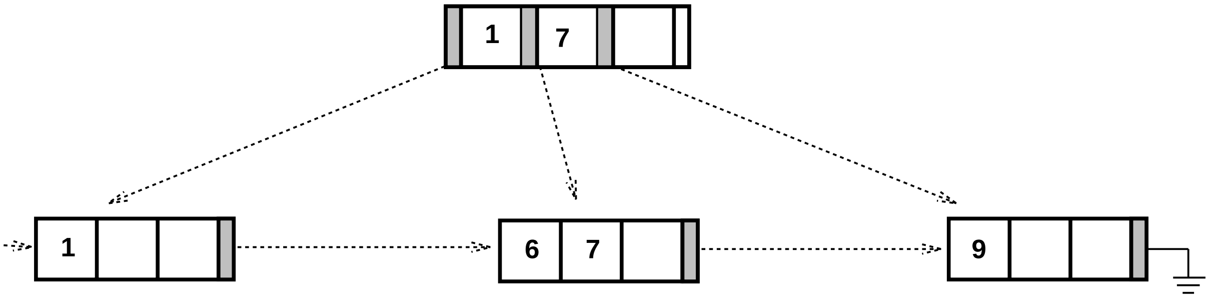
  - Internal value of ‘1’ does not belong in leaf ⇒ delete it.
  - Replace internal value of ‘3’ with predecessor (‘1’):



- Delete ‘8’:
  - Search for ‘8’.
  - ‘8’ occurs as an internal node ⇒ mark it.
  - Deletion of ‘8’ causes underflow ⇒ try to borrow from sibling
  - Borrowing from sibling causes underflow ⇒ pull down parent.
  - Borrowing at parent’s level causes underflow ⇒ pull down ‘7’.



  - Remove deleted internal value from leaf:



4.27 B-Trees and B+-Trees: A Summary

- B-trees and B+-trees:
  - are versatile indexing methods, usually used instead of plain single or multilevel indices;
  - support search in $O(\log_m n)$ time where

    $$\begin{array}{lcl} m & = & \text{degree of node} \\ n & = & \text{number of index blocks} \end{array}$$

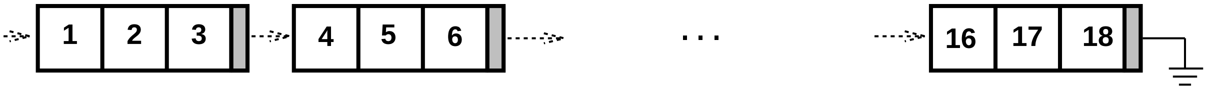
  - support dynamic insertion and deletion in time $O(\log_m n)$;
  - are in-order trees with multiple key values in each tree node;
- B+-trees are usually favored over B-trees in practice.
  ⇒ range search is faster in a B+-tree.
- Disadvantages?
  - Worst-case, each node will be only 50% full
    ⇒ twice the blocks needed for an ISAM index.
  - However, it can be shown that under uniform accesses, block occupancy is more than 65%.
  - B*-tree: a variation of a B+-tree that forces a high occupancy (67%).
- Some remaining practical issues:
  - *Duplicate values*:
    - Sometimes, we want to allow duplicate values.
    - In a B+-tree, this can be done by storing duplicate values in additional leaf nodes
      ⇒ during deletion, we must be careful to remove all duplicates.

- *Key compression*:
  - Sometimes, keys can be very long
    ⇒ few records in each index node
    ⇒ small m
    ⇒ not very efficient.
  - In many cases, it makes sense to use only a *substring* in the key (e.g., a small prefix)
    ⇒ compression of key increases m
    ⇒ more efficient.
  - Need to be careful with duplicate substrings from different keys.
- *Bulk loading*:
  - Often, a B+-tree is created on top of an existing heapfile.
  - Naive approach to building B+-tree: scan heapfile and insert items successively.
  - Successive insertion can cause many nodes to have low occupancy.
  - Better to create initial B+-tree using a *bulk loading* algorithm.
  - Some bulk loading algorithms build the tree bottom-up.

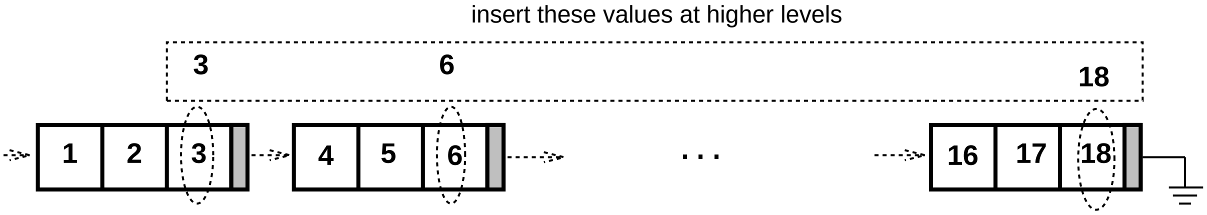
- Example of bulk-loading ($m = 2$):

  Create a B+-tree from the keys 1,2,3,...,18.

- First, create the leaf-level nodes in order:

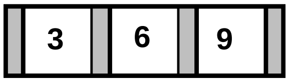


- Next, create copies of last values in each block: leaf block: 3,...,18.

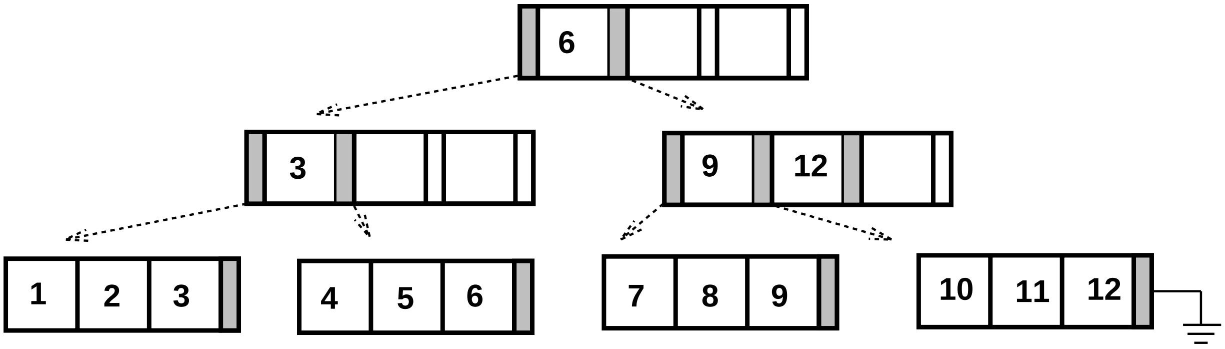


  These values are to be copied into higher levels.

- Values 3,6,9 form first block at next level



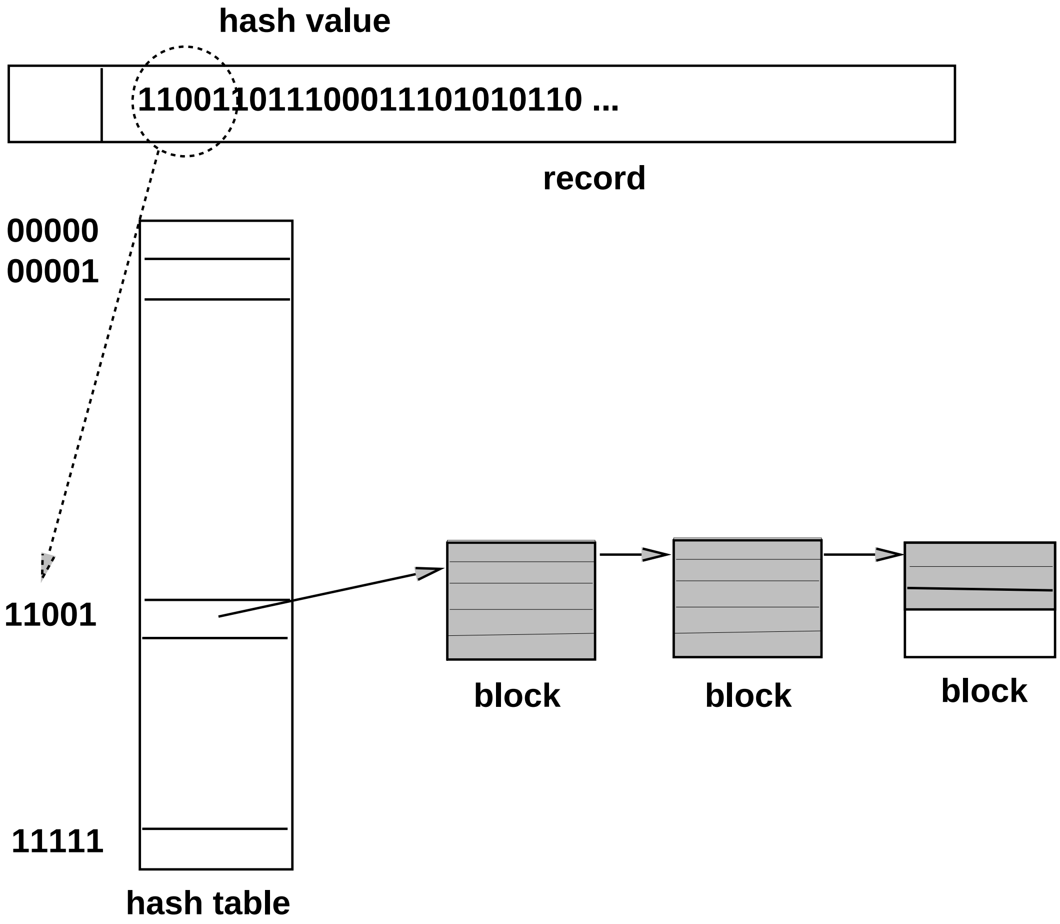
- Inserting 12 causes a split:



- Other items are added similarly.

4.28 Hash Indices: Introduction

- Recall key ideas in hashing:



  - A hashing function is applied to each record
    ⇒ a mapping function from the bits in the record to integers
    e.g, pick the first 5 bits in the record
    ⇒ maps records to the integers $0, 1, 2, \ldots, 31$.
  - Each integer is associated with a *bucket*: the integer is the *bucket number*.

  - Initially, a block is assigned to each bucket
    ⇒ as records get added to the bucket they are placed in the block.
  - Later, when the first block assigned to a bucket gets filled up, additional blocks are used.
  - The hash table itself is a (small) collection of blocks.
  - Usually: try to pick hashing function so that records spread evenly across buckets.
  - Some buckets could have long overflow chains.

- Hashing can be used for indexing in two ways:

  Separate index records:

  - For each data record, create an index record.
  - Index record: key and data pointer.
  - Place data records in data file (heap file).
  - Place index records in hashed file.
  - Index records are smaller
    ⇒ overflow chains may be short.
  - Note: hash table itself occupies blocks.

  No index records:

  - Data records are themselves placed in a hashed file.
  - Since data records are longer
    ⇒ overflow chains may be long, if table is unbalanced.

- Two major problems with simple hashing:

  Non-uniformity:

  - Distribution of data across buckets is not uniform.
  - Possible explanation: poor choice of hash function.
  - Also: data could be skewed.

  Dynamic Growth:

  - Data could grow arbitrarily.
  - Fixed number of buckets could overestimate or underestimate actual number of buckets needed.
  - Problem with underestimation: long chains.
  - Problem with overestimation: hash table unnecessarily big.
  - To change number of buckets in fixed scheme
    ⇒ all data must be re-hashed (complete re-organization).

- To solve *non-uniformity* problem:

  Select hashing function carefully:

  - Try to involve all contributing parts of the key, e.g.,
    - Key: 40-char (240-bit) string
    - Desired hash value: 16-bits (2^{16} buckets).
    - Add all 2-byte (16-bit) pairs in key.
  - Use special properties of numbers, e.g.,
    - Suppose x is an irrational number.
    - Let $f(x)$ = fractional part of x.
    - Then the collection of numbers $f(x), f(2x), f(3x), ...$ is approximately uniformly distributed.
    - Thus, pick an irrational x, multiply hash value k by x and use (portion of) fractional part.

- To solve *dynamic growth* problem:
  Use a dynamic hashing method, e.g.,

  1. *Extendible hashing.*
  2. *Linear hashing.*

  In both methods:

  – Index and data file grow as insertions are made.

  – Index and data file shrink with successive deletions.

  – Number of bits in hash value changes with data size
  $\Rightarrow$ initially, use very few bits; later, use more bits.

  – Lengths of bucket chains are controlled.

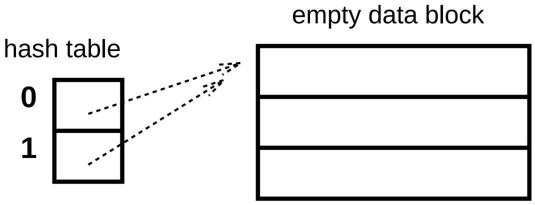
4.29 Extendible Hashing

- Key ideas in extendible hashing:
  - Initially, use a single hash bit
    ⇒ hash value is 0 or 1.
  - No overflow blocks are used.
  - When a block overflows, use additional bits to resolve overflow
  - Sometimes, the additional bit enlarges the hash table
    ⇒ the hash table is doubled.
  - Sometimes, only new blocks are created.
- Example:
  - Consider a key field consisting of an integer.
  - Use 6 bits for integer.
  - Most significant bit is first hash value.
  - Assume 3 data records fit into block.
  - No index records (data tuples are placed in hashtable blocks).

  Insert the following: 16, 49, 51, 21, 23, 54, 8, 44, 27, 11, 40, 25, 4, 35, 31, 9, 39, 28.
  In binary, these values are:

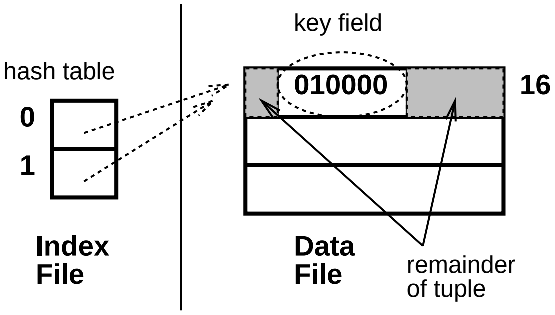
  010000 (16), 110001 (49), 110011 (51), 010101 (21), 010111 (23),
  110110 (54), 001000 (8), 101100 (44), 011011 (27), 001011 (11),
  101000 (40), 011001 (25), 000100 (4), 100011 (35), 011111 (31),
  001001 (9), 100111 (39), 011100 (28)

- Initially:
  - Only one hash bit.
  - One data block.
  - Both pointers point to same block.

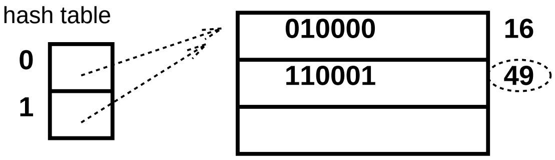


- Insert 010000 (16)
  - First bit 0.
    ⇒ Find 0 block.
  - Space available
    ⇒ Insert into block

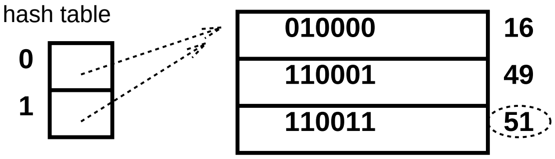
  NOTE:
  - Data blocks contain (full) data records.
  - Index blocks contain hash table entries.



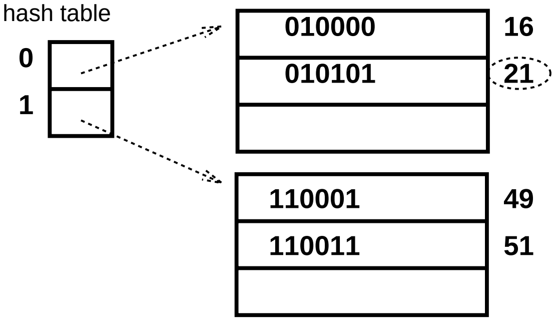
- Insert 110001 (49)
  - First bit is 1.
    ⇒ Find 1-block.
  - Space available.
    ⇒ Insert into block.



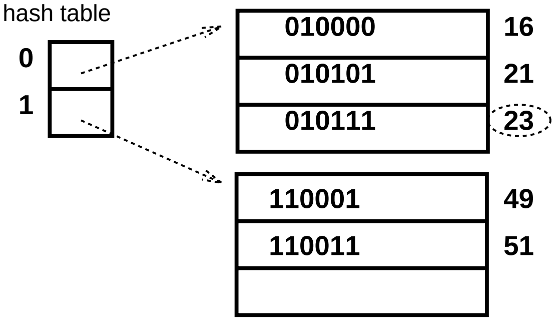
- Insert 110011 (51)
  - First bit is 1.
    ⇒ Find 1-block.
  - Space available.
    ⇒ Insert into block.



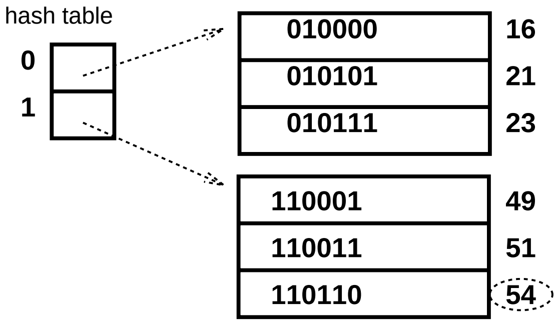
- Insert 010101 (21)
  - First bit is 0.
    ⇒ Find 0-block.
  - Block full.
    ⇒ Split block.
  - Distribute items.



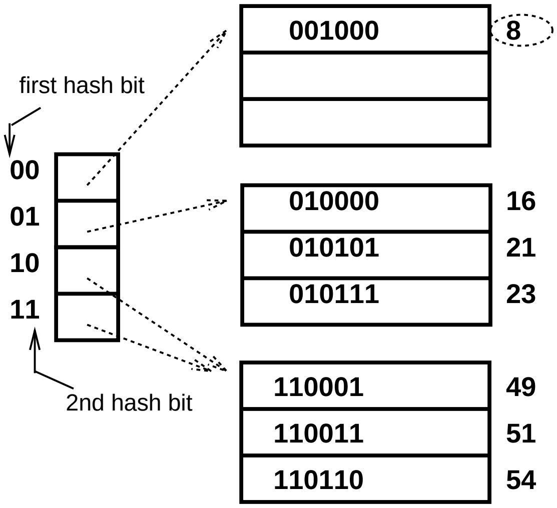
- Insert 010111 (23)
  - First bit is 0.
    ⇒ Find 0-block.
  - Space available.
    ⇒ Insert item.



- Insert 110110 (54)
  - First bit is 1.
    ⇒ Find 1-block.
  - Space available.
    ⇒ Insert item.

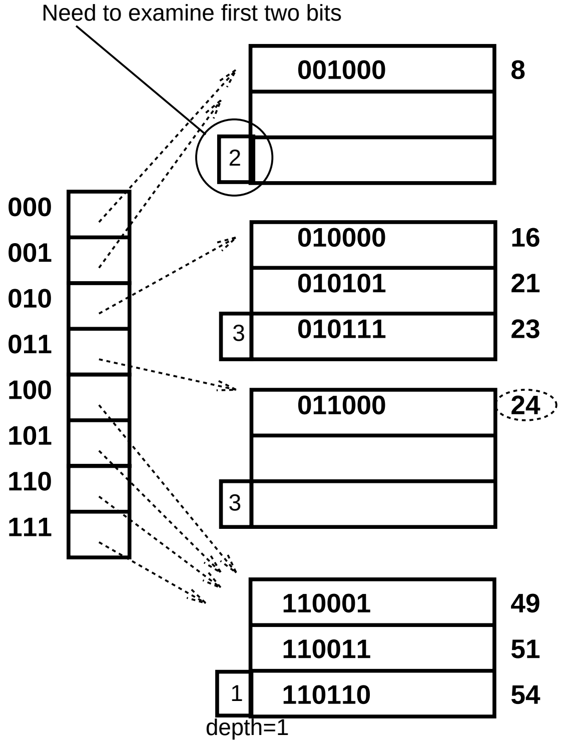


- Insert 001000 (8)
  - First bit is 0.
    ⇒ Find 0-block.
  - Block full.
  - Block is not shared.
    ⇒ Increase hash bits.
  - Double hash table.
  - Split block.
  - Distribute items.



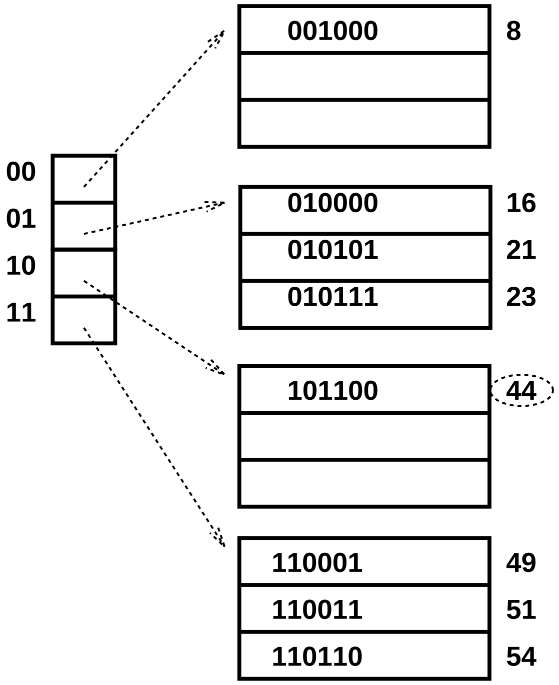
- NOTE:
  - Suppose we had inserted 011000 (24) instead of 8.
    ⇒ 2 bits are not enough to distinguish
    ⇒ further splitting is required:

    Insert 011000 (24)

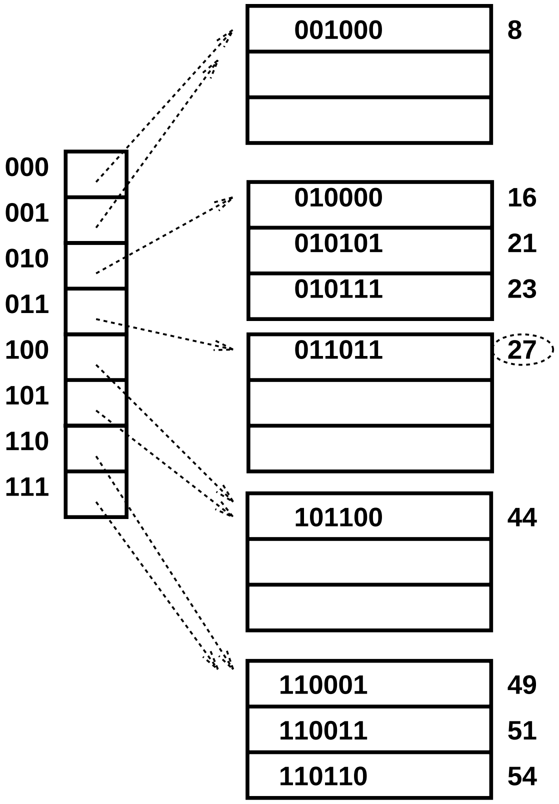


  - To identify the hash-bit level, a *depth* identifier is usually associated with each block.
  - The depth-id is used to decide if hash table needs doubling.
  - If depth-id is less than current hash-length, we only need to split the block (Table does not need doubling).
  - Later examples will not include block-depth in illustrations.

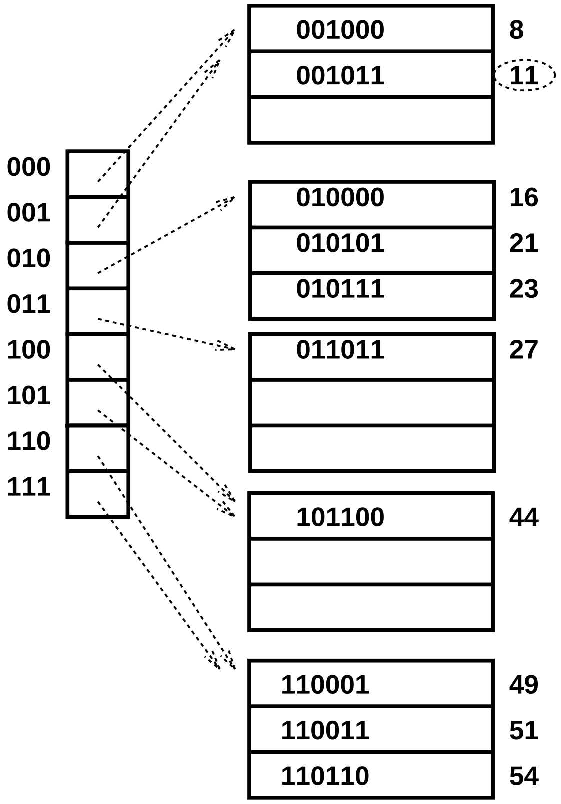
- (Previous example, continued):
  Insert 101100 (44)
  - Hash bits are 10.
    ⇒ Find 10-block.
  - Block full.
  - Block depth is 1.
    ⇒ Split block.
  - Distribute items.



- Insert 011011 (27)
  - Hash bits are 01.
    ⇒ Find 01-block.
  - Block full.
  - Block depth is 2.
    ⇒ Double hash table.
    ⇒ # hash bits = 3.
  - Split block.
  - Distribute items.

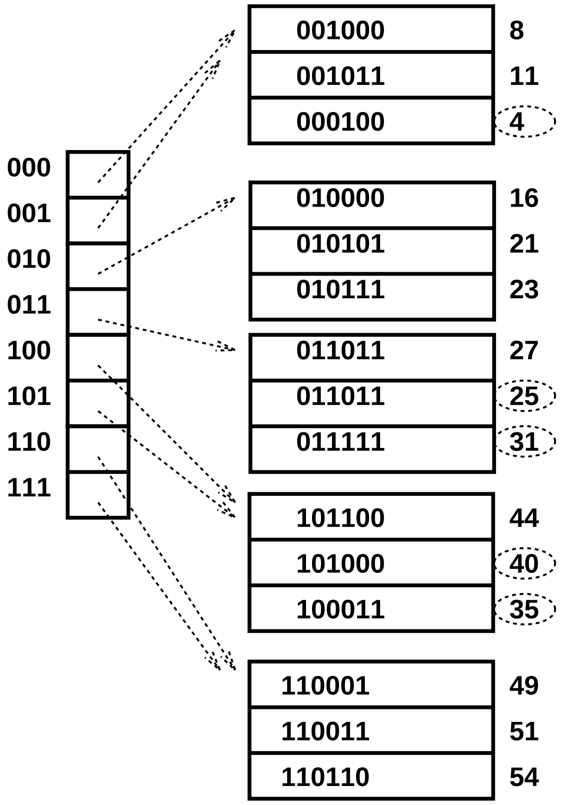


- Insert 001011 (11)
  - Hash bits are 001.
    ⇒ Find 001-block.
  - Space available.
    ⇒ Insert item.

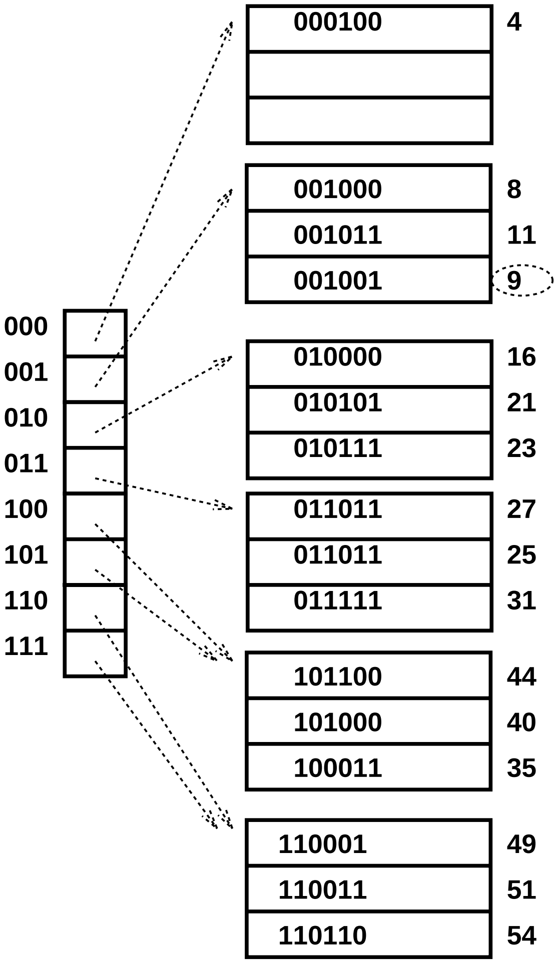


- Similarly, (it so happens) there is space for the next 5 inserts: 101000 (40), 011001 (25), 000100 (4), 100011 (35), 011111 (31).

Insert
101000 (40),
011001 (25),
000100 (4),
100011 (35),
011111 (31).



- Insert 001001 (9)
  - Hash bits are 001.
    ⇒ Retrive 001-block.
  - Block is full.
  - Block depth is 2.
    ⇒ split block.



000
001
010
011
100
101
110
111

000100 4

001000 8
001011 11
001001 9

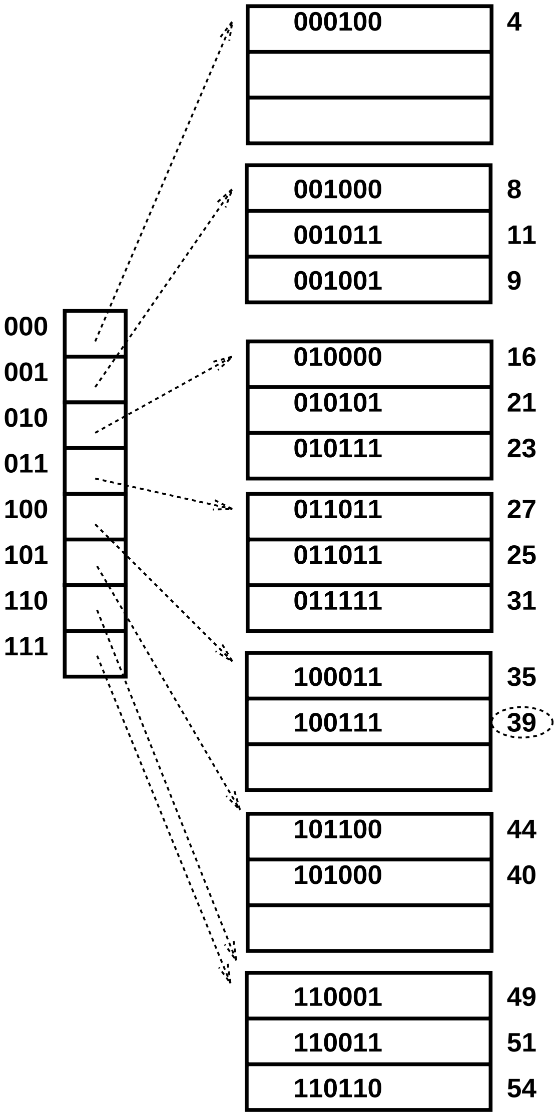
010000 16
010101 21
010111 23

011011 27
011011 25
011111 31

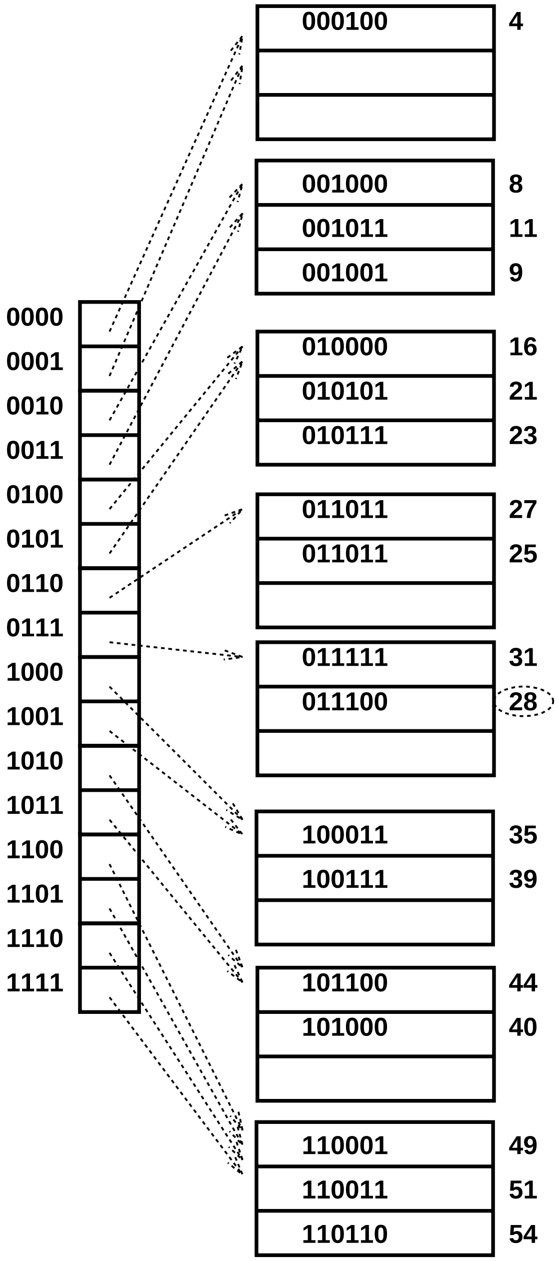
101100 44
101000 40
100011 35

110001 49
110011 51
110110 54

- Insert 100111 (39)
  - Hash bits are 100.
    ⇒ Retrive 100-block.
  - Block is full.
  - Block depth is 2.
    ⇒ split block.

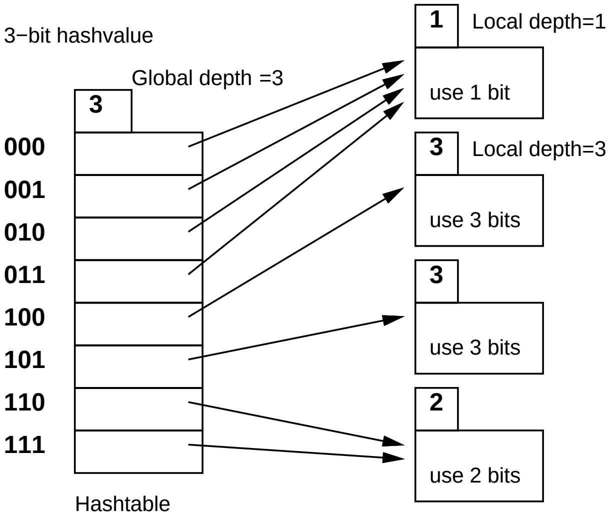


- Insert 011100 (28)
  - Hash bits are 011.
    ⇒ Retrive 011-block.
  - Block is full.
  - Block depth is 3.
    ⇒ double hash table.

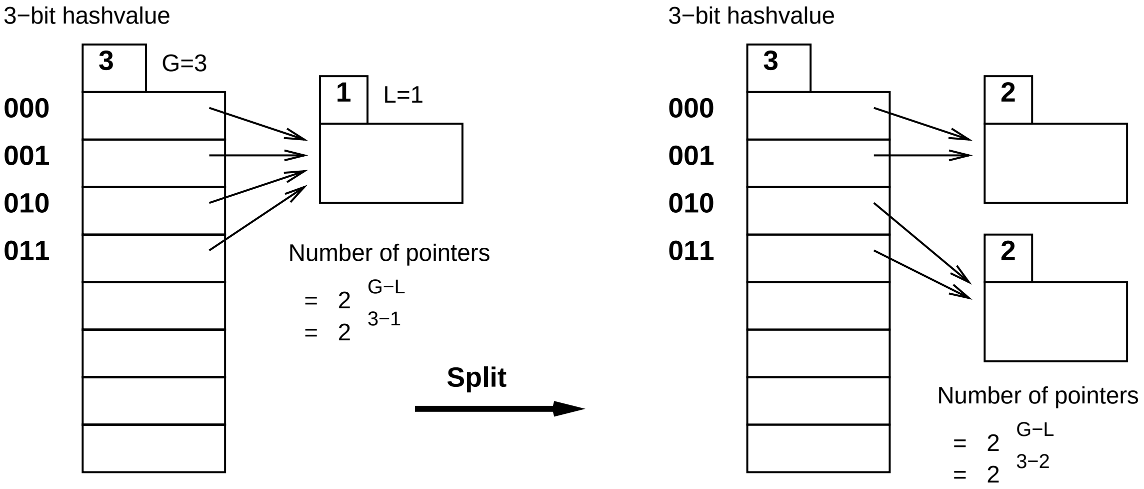


- NOTE:
  - If directory (hash table) doubling occurs too often
    ⇒ slow expansion by allowing more overflow blocks in each bucket.
  - When expansion occurs, only directory data is moved.
- Problems with extendible hashing:
  - A directory (hash table) is needed.
  - When directory is doubled
    ⇒ index data is moved between blocks
    ⇒ major overhead for large tables.
- Implementation of extendible hashing: we will use an interface identical to the B-tree interface:
  - Exhash-Create (tuplesize, keysize, keyoffset)
    – Creates a hashfile and initializes the directory.
  - Exhash-Insert (tuple)
    – Inserts a tuple into the hashfile.
    – Note: hashfile will contain whole tuples as opposed to only keys.
  - Exhash-Search (key)
    – Returns a tuple of key is found, NULL otherwise.

  In addition, functions will be needed for splitting a block and doubling the directory.
- Before presenting the pseudocode, it will help to review some ideas and also look at some finer details:
  - Recall the concept of *depth*:



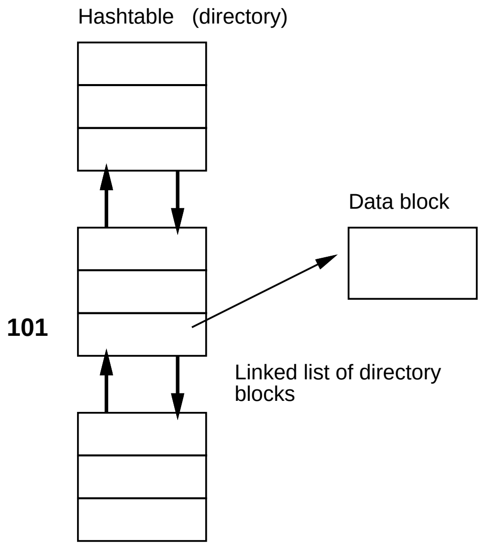
- $*$ How many directory entries point to a data block?
  Answer: $2^{(\text{Global\_depth - Local\_depth})}$.
- $*$ Or, 2^{G-L}, if we use G for global depth and L for a block's local depth.
- $*$ Another way of understanding local depth: the number of hash-bits you need to examine to distinguish entries in the block from entries in other blocks.
- $*$ When a block is split, its (local) depth increases:



- We will use *hashtable* and *directory* interchangeably.
- Two ways of implementing a directory:

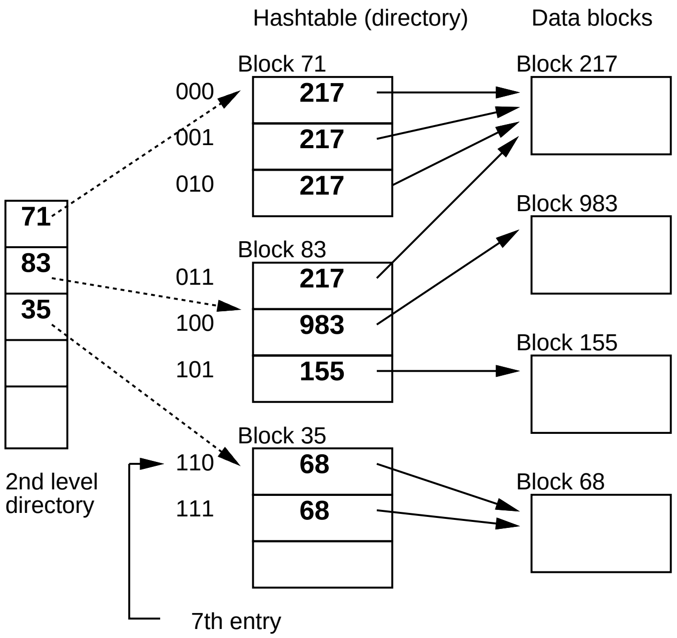
(Recall: directory is itself stored on disk)

1. *Linked list of blocks*:



- ∗ Given hashvalue, compute which block contains entry, then get directory block.
- ∗ After directory block is obtained, find appropriate entry and get data block.
- ∗ Advantages: Simple, directory is self-contained.
- ∗ Disadvantages: Long list traversal may be needed for single data searches.

2. 2-level directory:

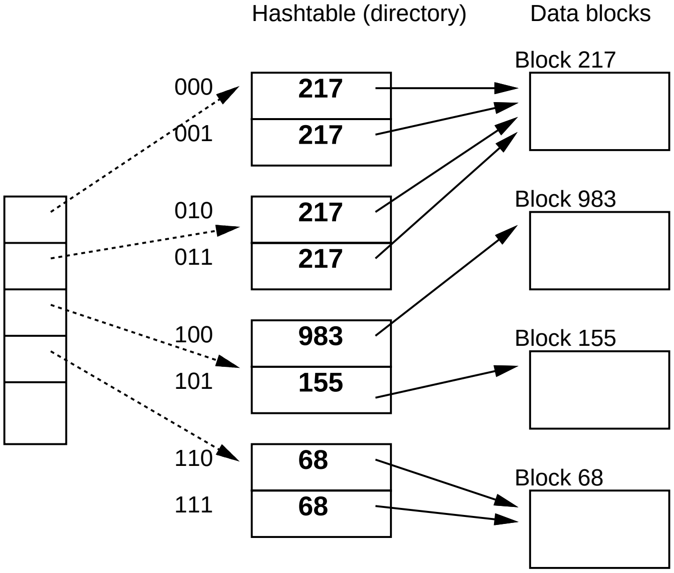


- ∗ 1st-level contains pointers to data blocks.
- ∗ 2nd-level contains pointers to 1st-level blocks.
- ∗ Example: consider hashvalue = 110:
  - · hashvalue 110 ⇒ 6+1 = 7th entry.
  - · Since there are 3 entries per dir-block
    ⇒ 7th entry is in $\lceil \frac{7}{3} \rceil$ = 3rd dir-block.
  - · Now look up 3rd entry in 2nd-level directory
    ⇒ block # 35.
  - · Fetch block # 35 into memory.
  - · Compute offset (of hashvalue 110) within this block
    ⇒ $1 + 6 \bmod 3 =$ 1st entry in block
    ⇒ pointer to block 68.
  - · Finally, fetch data block 68.
- ∗ Advantages: Computing offsets is faster than list-traversal.
- ∗ If number of entries is a power-of-2, offsets can be computed using bit-shifts (very fast).
- ∗ Disadvantages: Extra level needed.

- Consider disadvantage:
  - Example: 5 million tuples, blocksize=1K, 4 bytes/entry
  - 5 million tuples and $\approx 10^3$ blocksize
    $\Rightarrow \approx 5 \times 10^6/10^3 = 5000$ data blocks
  - Dir entry needs 4 bytes
    $\Rightarrow 10^3/4 = 250$ entries per block
    $\Rightarrow 5000/250 = 20$ directory blocks
    $\Rightarrow$ only 20 2nd-level entries
    $\Rightarrow$ 1 block is sufficient!
    (Thus, 2nd-level blocks can be kept in main memory).

– We will use the 2-level approach and also make a simplifying assumption: the number of entries in a dir-block is a power-of-2.
e.g., block size = 96, entry size = 4 bytes
$\Rightarrow$ 24 possible entries
$\Rightarrow$ but use only 16 = largest power-of-2 less than 24.

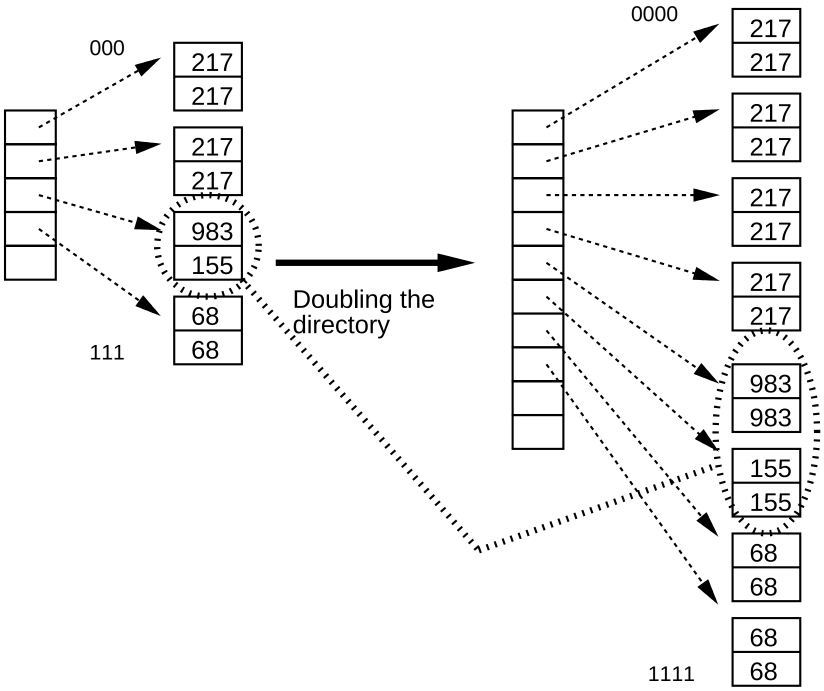
– For example (2 entries per block):



Reason for assumption: doubling the directory becomes easier to describe

⇒ one additional block needed for each dir-block.

- Doubling the directory:
  - Consider splitting block 155 above
    ⇒ directory must be doubled.
  - Two steps in doubling:
    1. Directly double the directory blocks.
    2. Modify 2nd-level appropriately.



- Redistributing entries in a split block:
  - Entries are redistributed by considering one additional bit.
- Splitting and doubling are independent of each other
  ⇒ sometimes a split does not need doubling the directory

- Finally, we present the pseudocode.

```
Algorithm:    EXHASH-CREATE (tuplesize, keysize, keyoffset)

Input: sizes of tuple and key, offset of key within tuple.
Output: 1st and 2nd level dir blocks, data block written to disk.
    Store size and offset information;
    Compute max_data_entries; // Max in a data block
    Compute max_dir_entries; // Max in a dir block
    // Create first data block, for hashbits 0 and 1
    blknum := DISK-NEWBLOCK();
    b := DISK-READBLOCK (blknum);
    b →num_entries := 0;
    b →depth := 1; // Local depth set to 1 (one bit)
    DISK-WRITEBLOCK (blknum);
    // Now create first directory block
    dir_blknum := DISK-NEWBLOCK();
    d := DISK-READBLOCK (dir_blknum);
    d →num_entries := 2; // Initially, pointers to 0,1 blocks
    d →blocklist[1] := blknum;
    d →blocklist[2] := blknum; // Both point to data block
    DISK-WRITEBLOCK (dir_blknum);
    // Now create 2nd level block
    SecL_blknum := DISK-NEWBLOCK();
    s := DISK-READBLOCK (SecL_blknum);
    s →num_entries := 1;
    s →entry[1] := dir_blknum;
    DISK-WRITEBLOCK (SecL_blknum);
    // Set Global depth
    G := 1;
    return;
```

Algorithm: EXHASH-SEARCH (key)

Input: key value to search for.

Output: corresponding tuple if found, **null** otherwise.

```
// Compute hashvalue based on $G$ bits.
$h$ := HASH-VALUE (key, $G$);
dir_blknum := Compute which block $h$ lies in (use 2nd level);
offset := Compute offset within the dir block;
$d$ := DISK-READBLOCK (dir_blknum);
blknum := $d$ →blocklist[offset] // block number of datablock
$b$ := DISK-READBLOCK (blknum);
for $i$ := 1 to $b$ →num_entries
   Compare key with $i$-th tuple in block $b$;
   if found return tuple;
endfor;
STACK-PUSH (blknum); // To be retrieved elsewhere
return null; // Not found
```

Algorithm: Exhash-Insert (x)

Input: tuple x to be inserted.
Output: insertion of tuple in hashfile.

```
// First check to see if key already exists in hashfile
if Exhash-Search ($x$.key) $\neq$ null;
    print 'Error:  key exists';
    return;
endif;
// Otherwise, retrieve data block from stack (where search ended)
blknum := Stack-Pop ();
$b$ := Disk-Readblock (blknum);
if $b$ →num_entries < max_data_entries
    $d$ →num_entries := $d$ →num_entries+1 //Space available
    $d$ →data[$d$ →num_entries] := $x$;
    Disk-Writeblock (blknum);
    return;
endif;
// Otherwise a split is needed
if $G \neq b$ →depth // No doubling needed now
    Recursive-Split-Insert ($x$);
else // Doubling needed since all bits are being used
    Double-Directory ();
    Recursive-Split-Insert ($x$);
endif;
return;
```

Algorithm: DOUBLE-DIRECTORY ()

Input: none.
Output: a new directory, doubled in size.

```
for $i$ := 1 to num_dir_blocks
    dnum := Get block number of $i$-th dir block;
    $d$ := DISK-READBLOCK (dnum);
    dnum2 := DISK-NEWBLOCK (); // For each doubling
    $d2$ := DISK-READBLOCK (dnum2);
    Double bottom half of $d$ into $d2$;
    Double upper half of $d$ into $d$;
    DISK-WRITEBLOCK (dnum);
    DISK-WRITEBLOCK (dnum2);
endfor;
$G$ := $G + 1$; // Increment global depth
return;
```

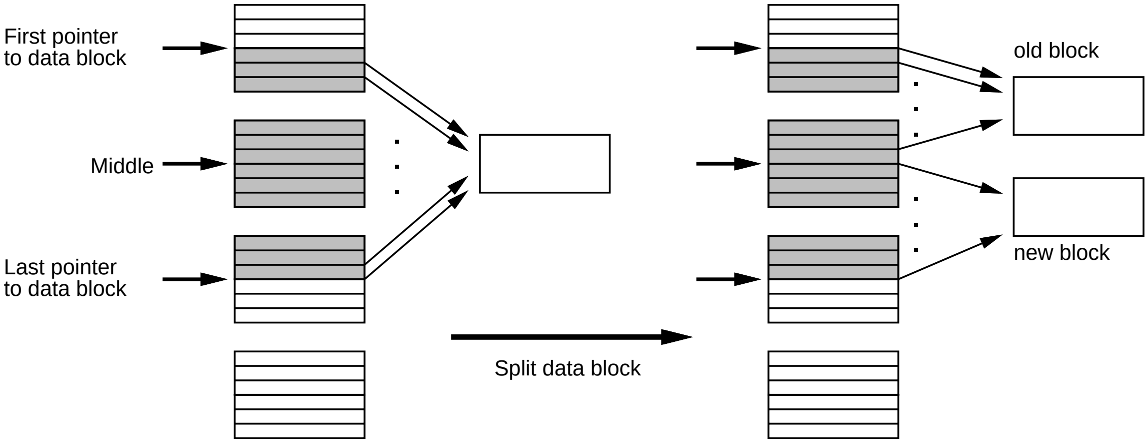
The Split-and-insert algorithm is the most complicated. Before presenting the pseudocode, we will present a useful function:

Algorithm: SET-DIRECTORY-ENTRY (hashval, data_blknum)

Input: a hashvalue and a blocknumber of a data block.
Output: The dir entry corresponding to the hashvalue is set to data_blknum.

```
// First, find out which dir block and offset
dir_blknum := Compute which block hashval lies in;
offset := Compute offset within the dir block;
// Now set the value
d := DISK-READBLOCK (dir_blknum);
d →blocklist[offset] := data_blknum;
DISK-WRITEBLOCK (dir_blknum);
return;
```

Note that when a split occurs, a bunch of identical pointers need to be split up into two groups:



We need to compute the First and Last dir entries that will be affected, as well as the Middle entry (the first entry that will point to the new block).

⇒ we need block numbers and offsets for these special dir entries.

Finally, the code for Recursive-Split-Insert:

Algorithm: Recursive-Split-Insert (x)

Input: A tuple x.
Output: Insertion of tuple into hashfile.

```
// First, recompute hashvalue, since $G$ might have changed
$h$ := Hash-Value ($x$.key, $G$);
// Next, compute data block and hashvalues Middle, Last
blknum := Find data block corresponding to $h$;
$b$ := Disk-Readblock (blknum);
First := $b \rightarrow$depth $* \; 2^{(G - b \rightarrow \text{depth})}$;
Last := First + $2^{(G - b \rightarrow \text{depth})}$;
Middle := First + $2^{(G - b \rightarrow \text{depth}) - 1}$;
// Get a new block and reset half the dir entries
bnum2 := Disk-Newblock ();
// Make all entries from Middle to Last point to new block
for $h2$ := Middle to Last
   Set-Directory-Entry ($h2$, bnum2);
$b$ := Disk-Readblock (blknum);
$b2$ := Disk-Readblock (bnum2);
Distribute the tuples among the two data blocks;
Set num_entries appropriately in each data block;
// Now insert the new tuple in the right block
if $h$ < Middle // Try to place in old block
   if $b \rightarrow$num_entries < max_data_entries
      Insert $x$ in $b$, increment $b \rightarrow$num_entries;
   else if $b \rightarrow$depth $\neq G$ // Regular split
      Recursive-Split-Insert ($x$);
   else // Directory doubling needed
      Double-Directory ();
      Recursive-Split-Insert ($x$);
   endif;
continued...
```

Algorithm: Recursive-Split-Insert ... continued

```
else // Try to place in new block
    if $b2$ →num_entries < max_data_entries
        Insert $x$ in $b$, increment $b$ →num_entries;
    else if $b2$ →depth $\neq G$ // Regular split
        Recursive-Split-Insert ($x$);
    else // Directory doubling needed
        Double-Directory ();
        Recursive-Split-Insert ($x$);
    endif;
endif;
return;
```

4.30 Linear Hashing

- Key ideas in *linear hashing*:
  - Avoid using a directory altogether.
    ⇒ faster and avoids copying overhead during expansion.
  - Tradeoffs:
    - Needs contiguous allocation of blocks, e.g., block numbers 550 to 3010.
    - Long overflow chains are possible in pathological cases.
  - Let h_i denote the hashing function that uses bits 0,...,$i-1$.
    ⇒ e.g., h_4 uses 5 bits.
  - At any given time, two successive functions are in use, e.g., h_4 and h_5.
  - Start with two empty data blocks (0 and 1).
  - Start with h_0 and h_1 as a pair of hash functions.
  - Need two block pointers: *Current* and *Last*.
  - Initially, *Current* = 0, *Last* = 1.
  - When a block overflows:
    - We *do not* necessarily split *that* block.
    - Instead, we split the *Current* block.
    - *Current* is incremented.
    - This way, blocks get split in turn, even if a split is triggered by another block overflowing.
    - Use an overflow block for the block that overflows. (Does not have to be contiguous).

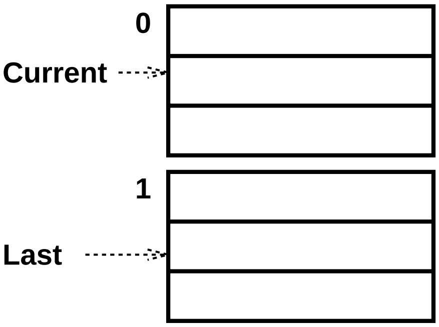
- When *Current* = *Last* and a split occurs:
  - ∗ Shift hash functions up by one (from h_i, h_{i+1} to h_{i+1}, h_{i+2}).
  - ∗ *Last* := largest h_i value.
- When searching, use h_{i+1} for blocks numbered before *Current*. Use h_i otherwise.

- Example:
  - We will assume the contiguous blocks start at 0. (For the general case, simply add offset).
  - Consider example used in extendible hashing with one difference: first hash-bit is *least significant bit*.
  - 6-bit integers.
  - Assume 3 data records per block.
  - No index records.

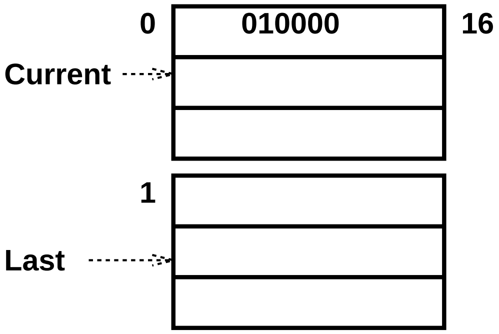
Insert the following: 16, 49, 51, 21, 23, 54, 8, 44, 27, 11, 40, 25, 4, 35, 31, 9, 39, 28.
In binary, these values are:

> 010000 (16), 110001 (49), 110011 (51), 010101 (21), 010111 (23),
> 110110 (54), 001000 (8), 101100 (44), 011011 (27), 001011 (11),
> 101000 (40), 011001 (25), 000100 (4), 100011 (35), 011111 (31),
> 001001 (9), 100111 (39), 011100 (28)

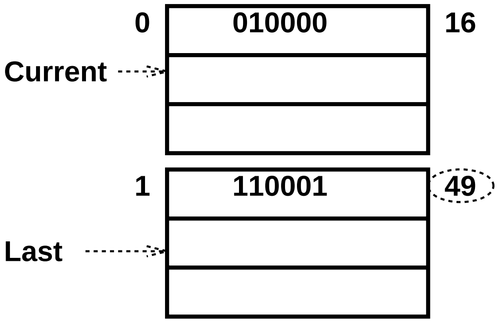
- Initially:
  - Two empty blocks, with block numbers 0 and 1.
  - Use functions h_0 and h_1.
  - *Current* := 0.
  - *Last* := 1.



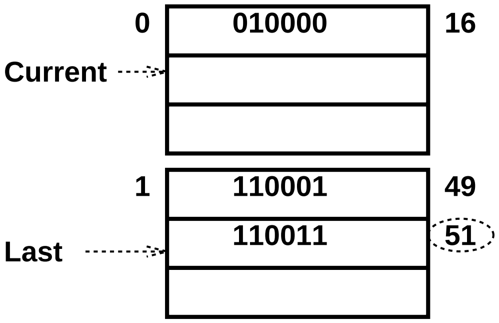
- Insert 010000 (16):
  - Apply h_0 to 010000.
  - $h_0(010000) = 0$ (lowest order bit).
    $\Rightarrow$ Retrieve block 0.
  - Block $0 \geq$ *Current*
    $\Rightarrow$ no need to use h_1.
  - Space available.
    $\Rightarrow$ Insert item.



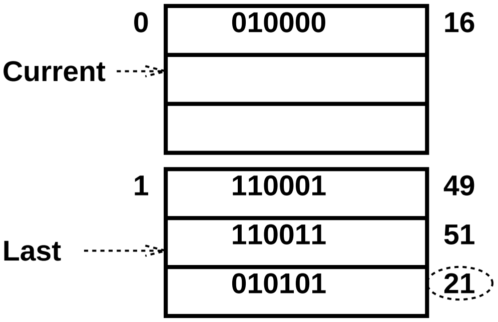
- Insert 110001 (49):
  - Apply h_0 to 110001
  - $h_0(110001) = 1$ (lowest order bit).
    $\Rightarrow$ Retrieve block 1.
  - Block $1 \geq$ *Current*
    $\Rightarrow$ no need to use h_1.
  - Space available.
    $\Rightarrow$ Insert item.



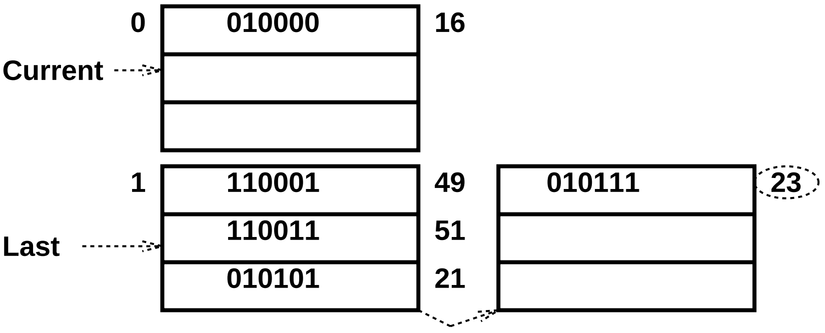
- Insert 110011 (51):
  - Apply h_0 to 110011:
  - $h_0(110011) = 1$.
    $\Rightarrow$ Retrieve block 1.
  - Block $1 \geq$ *Current*
    $\Rightarrow$ no need to use h_1.
  - Space available.
    $\Rightarrow$ Insert item.



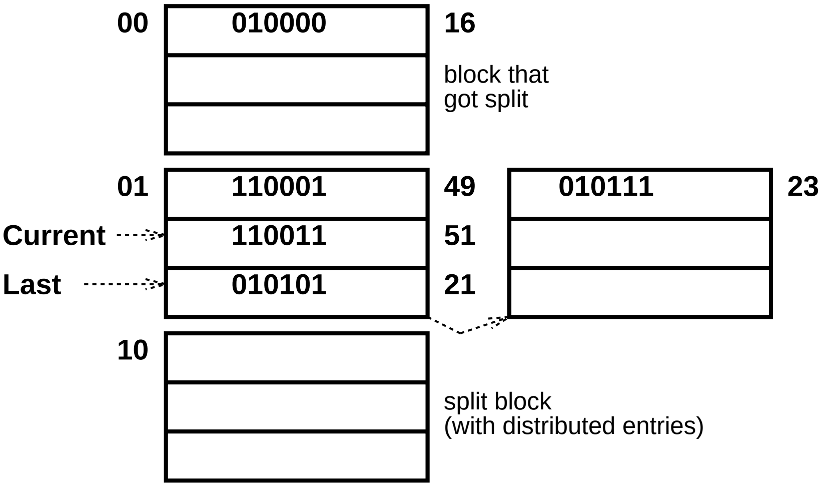
- Insert 010101 (21):
  - Apply h_0 to 010101.
  - $h_0(010101) = 1$.
    ⇒ Retrieve block 1.
  - Block 1 $\geq$ *Current*
    ⇒ no need to use h_1.
  - Space available.
    ⇒ Insert item.



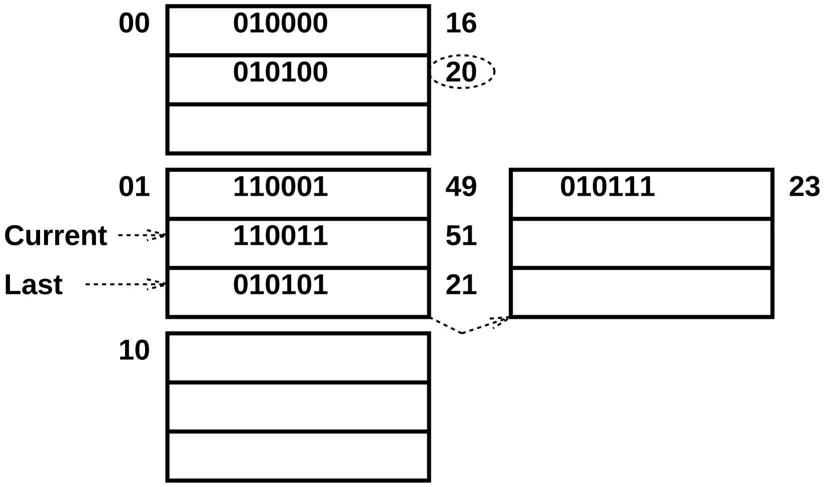
- Insert 010111 (23):
  - Apply h_0 to 010111.
  - $h_0(010111) = 1$.
    ⇒ Retrieve block 1.
  - Block 1 $\geq$ *Current*
    ⇒ no need to use h_1.
  - Block full
    ⇒ Create overflow block.



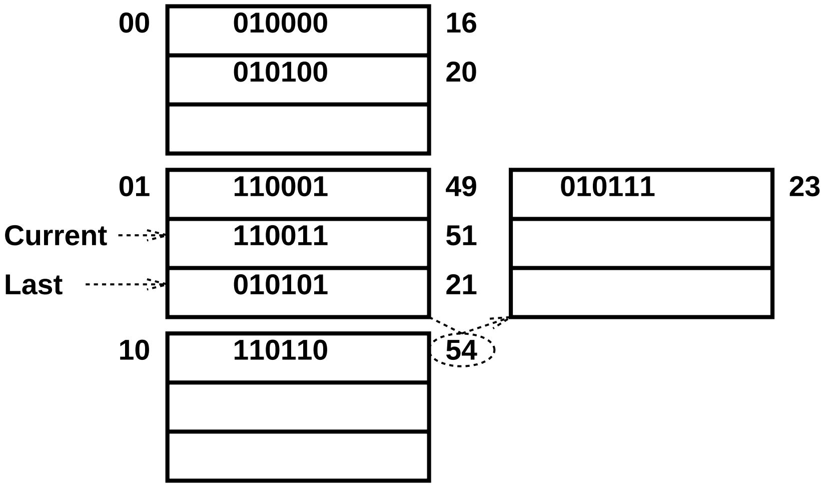
  - Split *Current* block.
  - Increment *Current*.



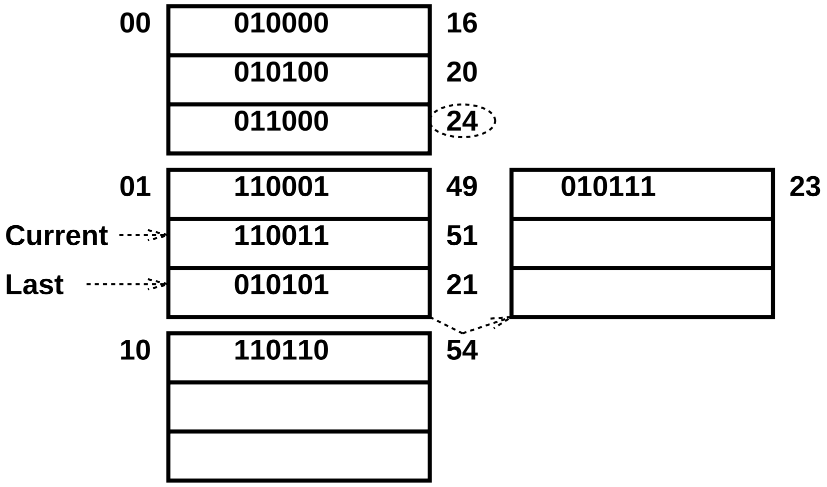
- Insert 010100 (20):
  - Apply h_0 to 010100.
  - $h_0(010100) = 0$.
  - Block 0 < *Current*
    ⇒ must use h_1.
  - $h_1(010100) = 00$.
    ⇒ Retrieve block 00.
  - Space available.
    ⇒ Insert item.



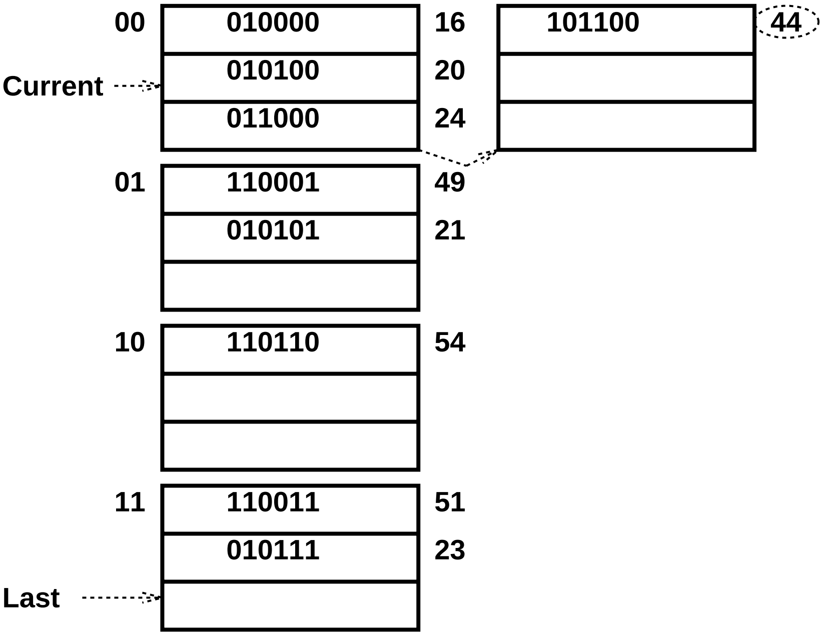
- Insert 110110 (54):
  - Apply h_0 to 110110.
  - $h_0(110110) = 0$.
  - Block 0 < *Current*
    ⇒ must use h_1.
  - $h_1(110110) = 10$.
    ⇒ Retrieve block 10.
  - Space available.
    ⇒ Insert item.



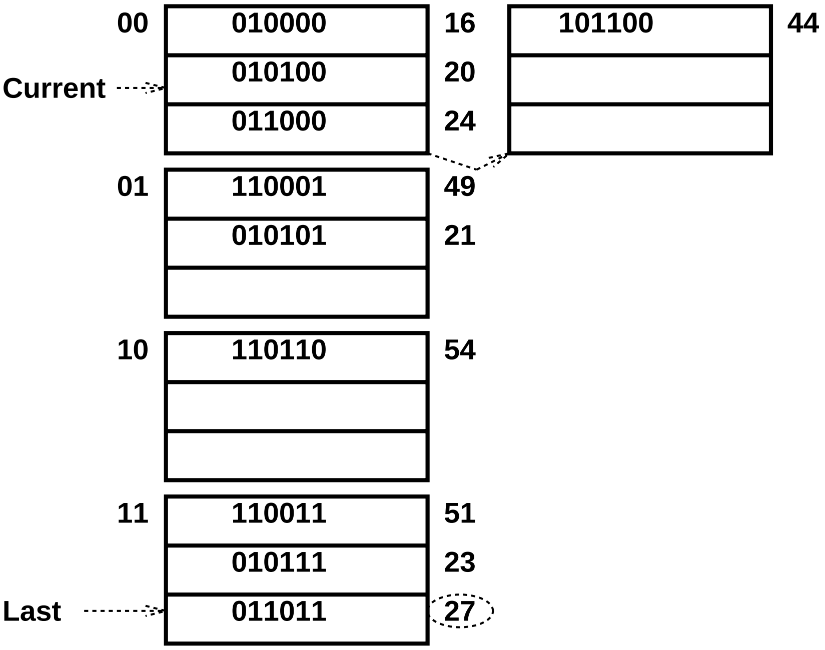
- Insert 011000 (24):
  - Apply h_0 to 011000.
  - $h_0(011000) = 0$.
  - Block $0 < \mathit{Current}$
    $\Rightarrow$ must use h_1.
  - $h_1(011000) = 00$.
    $\Rightarrow$ Retrieve block 00.
  - Space available.
    $\Rightarrow$ Insert item.



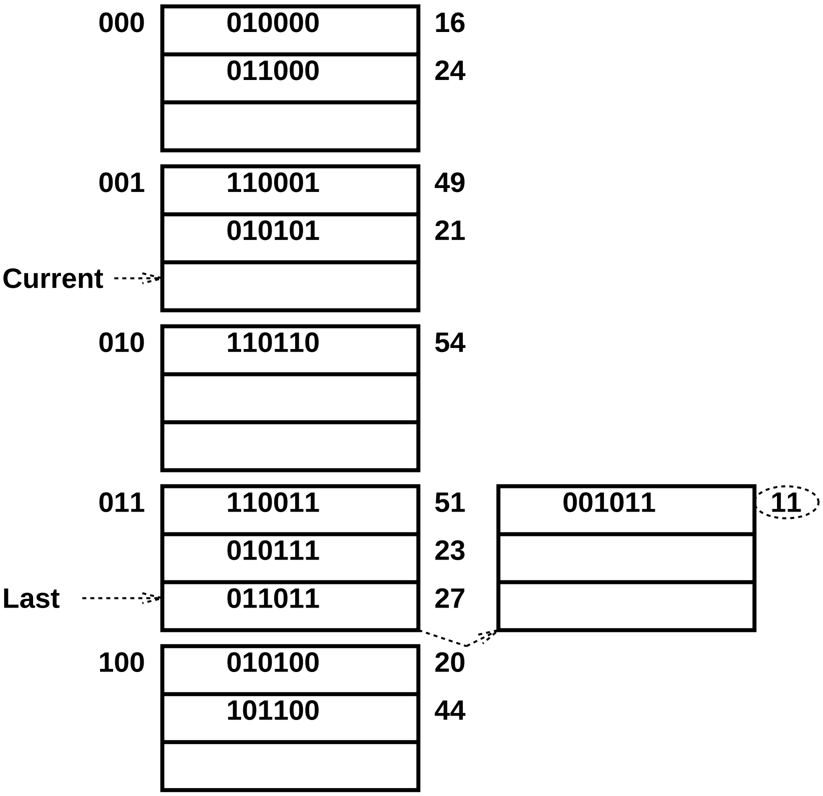
- Insert 101100 (44):
  - Apply h_0 to 101100.
  - $h_0(101100) = 0$.
  - Block $0 <$ *Current*
    $\Rightarrow$ must use h_1.
  - $h_1(101100) = 00$.
    $\Rightarrow$ Retrieve block 00.
  - Block full $\Rightarrow$
    1. Create overflow block (for block 00).
    2. Split *Current* block (block 01).
    3. Distribute items among split blocks (using h_1).
  - *Current* = *Last* $\Rightarrow$
    1. *Last* := largest h_1 value.
       $\Rightarrow$ *Last* := 11.
    2. *Current* := 00 (first block).
    3. Use functions h_1, h_2 now.



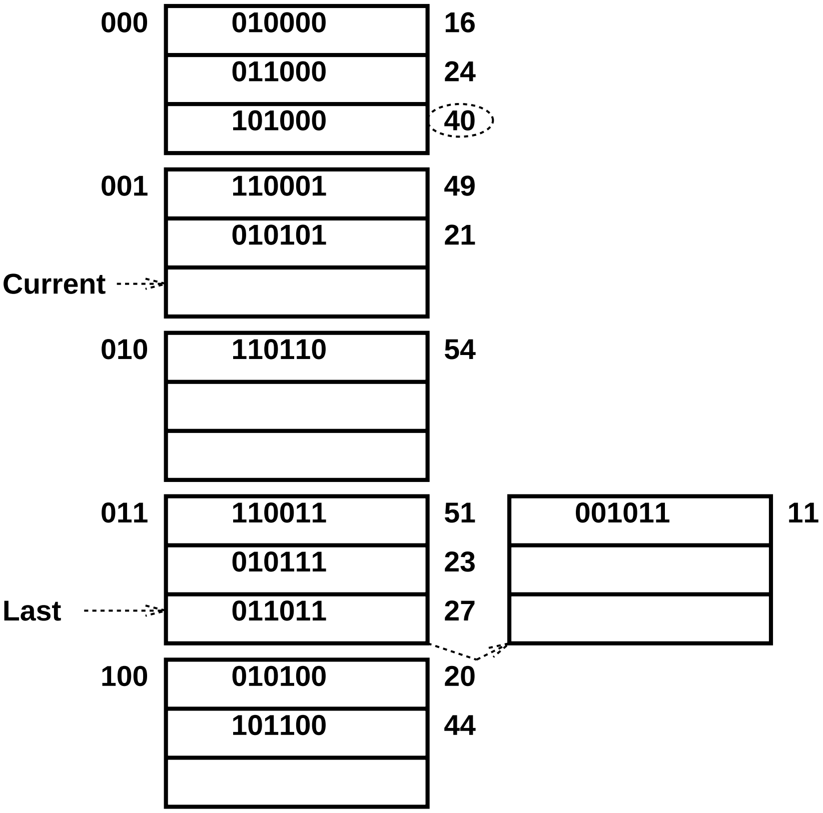
- Insert 011011 (27):
    - Functions to use: h_1, h_2.
    - Apply h_1 to 011011.
    - $h_1(011011) = 11$.
    - Block 11 $\geq$ *Current*
      $\Rightarrow$ no need to use h_2.
      $\Rightarrow$ Retrieve block 11.
    - Space available.
      $\Rightarrow$ Insert item.



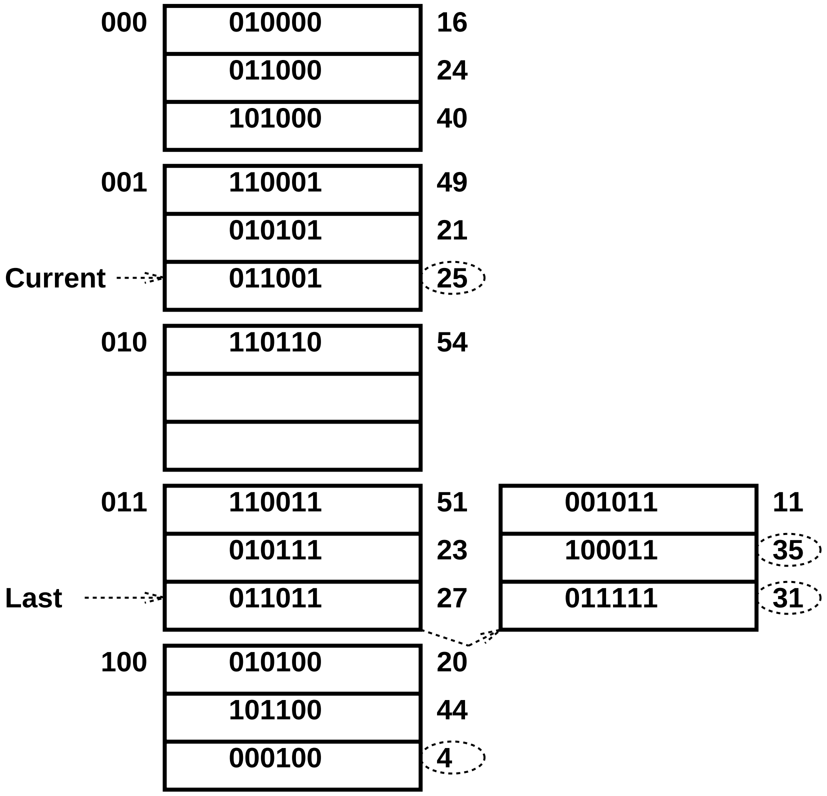
- Insert 001011 (11):
  - Apply h_1 to 001011.
  - $h_1(001011) = 11$.
  - Block 11 $\geq$ *Current*
    ⇒ no need to use h_2.
    ⇒ Retrieve block 11.
  - Block full ⇒
    1. Create overflow block (for block 11).
    2. Split *Current* block (block 00).
    3. Distribute items among split blocks (using h_2).
  - Increment *Current*.



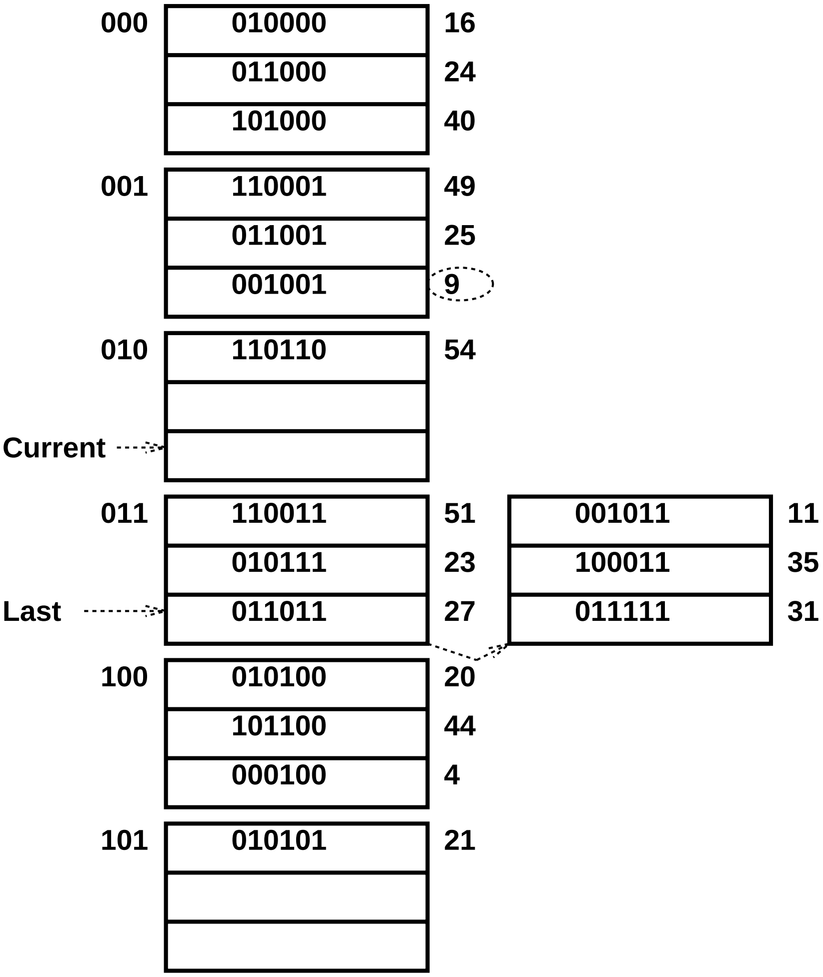
- Insert 101000 (40):
  - Apply h_1 to 101000.
  - $h_1(101000) = 00$.
  - Block 00 $<$ *Current* $\Rightarrow$ must use h_2.
  - $h_2(101000) = 000$. $\Rightarrow$ Retrieve block 000.
  - Space available. $\Rightarrow$ Insert item.



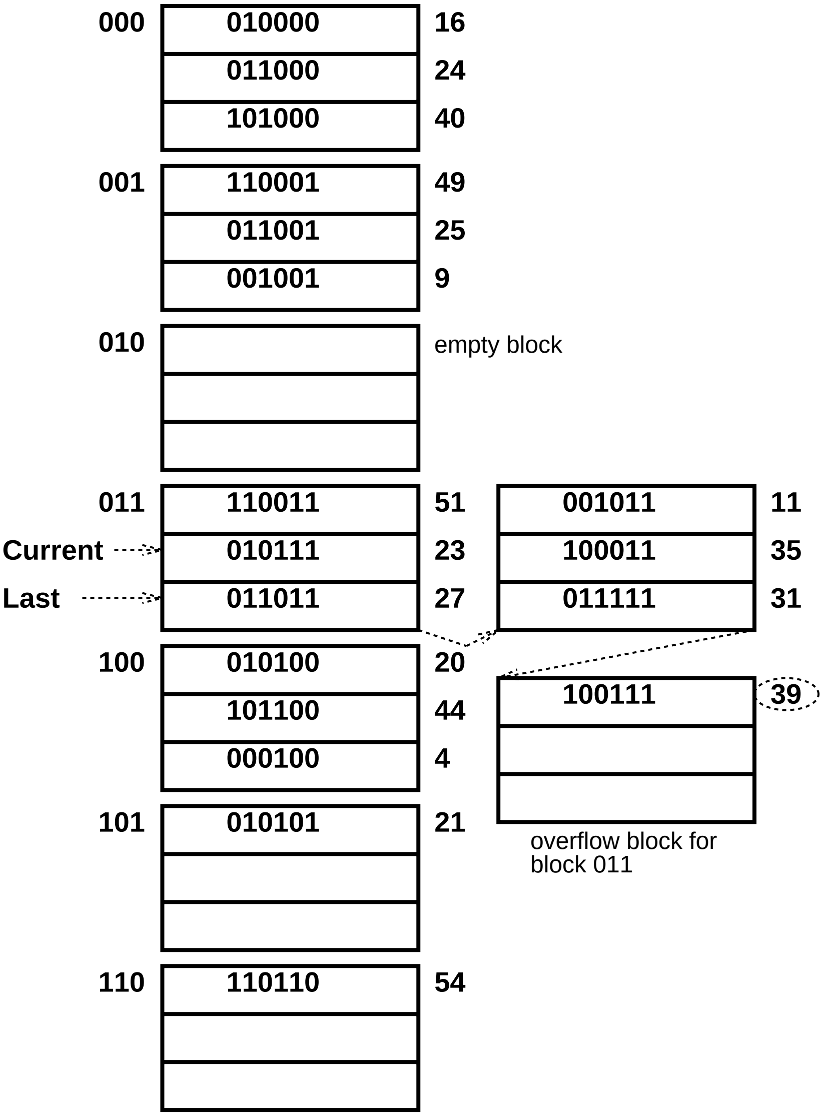
- Similarly (it so happens), space is available for the next 4 items:
  Insert
  011001 (25),
  000100 (4),
  100011 (35),
  011111 (31)



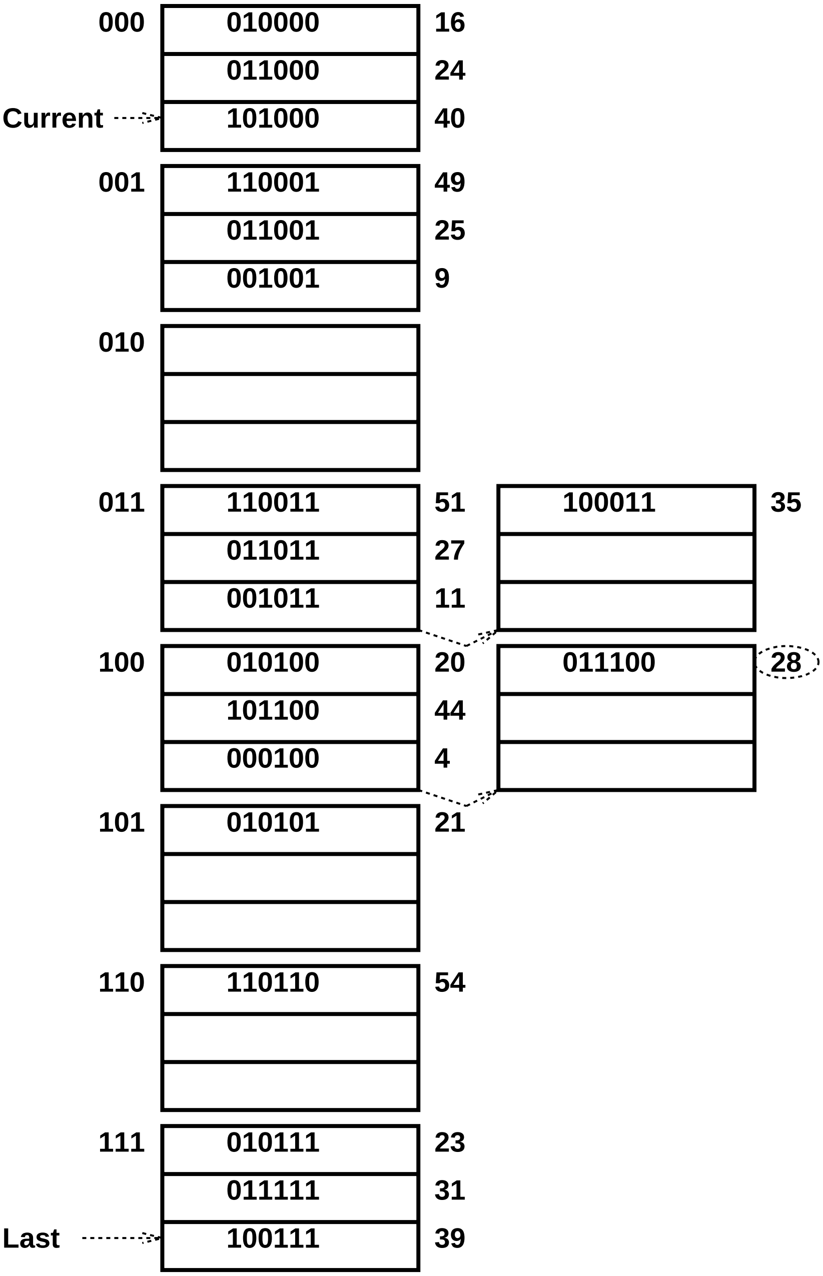
- Insert 001001 (9):
  - Apply h_1 to 001001.
  - $h_1(001001) = 01$.
  - Block 01 $\geq$ *Current*
    $\Rightarrow$ don't use h_2.
    $\Rightarrow$ Retrieve block 01 (block 001).
  - Block full $\Rightarrow$
    1. Create overflow block (for block 001).
    2. Split *Current* block (block 001).
    3. Distribute items among split blocks (using h_2).
  - Note: overflow and split can be done together.
  - Increment *Current*.



- Insert 100111 (39):
  - Apply h_1 to 100111.
  - $h_1(100111) = 11$.
  - Block 11 $\geq$ *Current*
    $\Rightarrow$ don't use h_2.
    $\Rightarrow$ Retrieve block 11 (block 011).
  - Block full $\Rightarrow$
    1. Create overflow block (for block 011).
    2. Split *Current* block (block 010).
    3. Distribute items among split blocks (using h_2).
  - Note: block 010 is empty after distribution.
  - Increment *Current*.



- Insert 011100 (28):
  - Apply h_1 to 011100.
  - $h_1(011100) = 00$.
  - Block 00 < *Current*
    ⇒ must use h_2.
  - $h_2(011100) = 100$.
    ⇒ Retrieve block 100.
  - Block full ⇒
    1. Create overflow block (for block 100).
    2. Split *Current* block (block 011).
    3. Distribute items among split blocks (using h_2).
  - *Current* = *Last* ⇒
    1. *Last* := largest h_2 value.
       ⇒ *Last* := 111.
    2. *Current* := 000 (first block).
    3. Use functions h_2, h_3 now.



- Implementation of linear hashing: we will use an interface identical to the B-tree interface:
  - Linhash-Create (tuplesize, keysize, keyoffset)

  - Creates a hashfile and initializes the directory.

- Linhash-Insert (tuple)
  - Inserts a tuple into the hashfile.
  - Note: hashfile will contain whole tuples as opposed to only keys.

- Linhash-Search (key)
  - Returns a tuple of key is found, NULL otherwise.

In addition, functions will be needed for overflow and splitting.

We will assume that Disk-Allocate-Contiguous-Blocks allocates a large group contiguous blocks on disk and that successive calls to Disk-Get-Contiguous-Block return consecutive block numbers.

Each block b will contain the following local-to-the-block fields:

- b →num_entries: the number of tuples currently in the block.
- b →next: a block number of the next block in the chain.
  (b →next $= 0$ if no chain exists.)

Algorithm: Linhash-Create (tuplesize, keysize, keyoffset)

Input: size of tuple and key, offset of key within tuple.
Output: First and Last data blocks, written to disk.

```
Store size and offset information;
Compute max_entries;
Disk-Allocate-Contiguous-Blocks ();
First := Disk-Get-Contiguous-Block ();
Last := Disk-Get-Contiguous-Block ();
Current := 0;
$b$ := Disk-Readblock (First);
$b$ →num_entries := 0;
$b$ →next := 0;
Disk-Writeblock (First);
$b$ := Disk-Readblock (Last);
$b$ →num_entries := 0;
$b$ →next := 0;
Disk-Writeblock (Last);
$I$ := 0; // Start with $h_I = h_0$.
return;
```

```
Algorithm: Linhash-Search (key)

Input: search key.
Output: tuple, if found.
hashval := $h_I$ (key);
if hashval < Current // Use second hash function
    hashval := $h_{I+1}$ (key);
bnum := First + hashval;
$b$ := Disk-Readblock (bnum);
for $i$ := 1 to $b$ →num_entries
    Compare key with $i$-th tuple in block $b$;
    if found
        return tuple;
endfor;
if $b$ →next ≠ 0 // search rest of chain, if any
    return Linhash-Recursive-Search (key, $b$ →next);
return null;
```

Algorithm: LINHASH-RECURSIVE-SEARCH (key, blocknum)

Input: search key, start block number.
Output: tuple if found.

```
$b$ := DISK-READBLOCK (blocknum);
for $i$ := 1 to $b \rightarrow$num_entries
    Compare key with $i$-th tuple in block $b$;
    if found
        return tuple;
endfor;
if $b \rightarrow$next $\neq 0$ // search rest of chain, if any
    return LINHASH-RECURSIVE-SEARCH (key, $b \rightarrow$next);
return null;
```

Algorithm: Linhash-Insert (tuple)

Input: tuple to be inserted.
Output: insertion of the tuple into the hash file.

```
if hashval < Current
    hashval := $h_{I+1}$ (key);
    bnum := First + hashval;
$b$ := Disk-Readblock (bnum);
if $b$ →num_entries < max_entries
    $b$ →num_entries := $b$ →num_entries + 1;
    Insert tuple into block $b$;
    Disk-Writeblock (bnum);
else if $b$ →next = 0 // Need to create a chain
    Linhash-Create-Overflow (bnum, tuple);
    Linhash-Split-Current (); // Must split
else // Chain exists; split only if overflow occurs.
    Linhash-Recursive-Overflow ($b$ →next, tuple);
endif;
return;
```

Algorithm: Linhash-Create-Overflow (blknum, tuple)

Input: block number to chain from, tuple to be inserted.
Output: insertion of the tuple into the hash file.

```
b := Disk-Readblock (blknum);
newblk := b →next := Disk-Newblock ();
Disk-Writeblock (blknum);
b := Disk-Readblock (newblk);
b →num_entries := 1;
b →next := 0;
Insert tuple into b;
Disk-Writeblock (newblk);
```

Algorithm: Linhash-Recursive-Overflow (blknum, tuple)

Input: blknum where tuple is to be inserted.
Output: insertion of the tuple into the hash file.

```
$b$ := Disk-Readblock (blknum);
if $b \rightarrow$num_entries = max_entries
    if $b \rightarrow$next $\neq 0$
        Linhash-Recursive-Overflow ($b \rightarrow$next, tuple);
    else
        Linhash-Create-Overflow (blknum, tuple);
        Linhash-Split-Current ();
    endif;
else
    $b \rightarrow$num_entries := $b \rightarrow$num_entries + 1;
    Insert tuple into block $b$;
    Disk-Writeblock (blknum);
endif;
return;
```

Algorithm: Linhash-Split-Current ()

Input: none.
Output: Split the block pointed by Current.

```
blk := First + Current;
$b$ := Disk-Readblock (blk);
newblk := Disk-Get-Contiguous-Block ();
$b2$ := Disk-Readblock (newblk);
Distribute entries of $b$ (including chain) among $b$ and $b2$;
Write all these blocks to disk;
Current := Current + 1;
if Current = Last
    $I$ := $I + 1$; // Now use $h_{I+1}$ and $h_{I+2}$;
    Current := 0;
    Last := $2^{I+1} - 1$;
endif;
return;
```

4.31 Hash Indices: Summary

- Key ideas common to hashing methods:
  - Hashing is used for *equality* search on a key.
  - Apply hashing function to a key to obtain bucket #.
  - Store data record in bucket.
  - A bucket may be a single block (hopefully) or a chained list of blocks.
  - Dynamic methods
    - reduce occurence of long chains;
    - allow regulated growth of hash index;
    - use varying numbers of hash-bits;
  - Poor choice of hashing function can result in skew (non-uniform distribution across buckets).
- Deletions:
  - Deletions are approximately the “reverse” of insertions.
  - Deletions result in empty blocks
⇒ return empty blocks.
  - In extendible hashing: directory occasionally collapses.
  - In linear hashing: use lower pair of hash functions when appropriate.
  - In practice: hashing is often used for temporary files created during joins.
⇒ deletion not used often.

- Extendible vs. linear hashing:

  Extendible hashing:

  - Advantages:
    - Always splits overflows
      ⇒ at most one block accessed during search.
  - Disadvantages:
    - Uses a directory.
    - Directory space may be wasted if directory doubles too often.
      ⇒ e.g., pathological case: continual insertions into bucket 0.

  Linear hashing:

  - Advantages:
    - Does not use directory.
    - Splitting is decoupled from insert bucket.
  - Disadvantages:
    - Needs contiguous allocation of blocks.
    - Long chains are still possible in pathological cases.

- NOTE:
  - Neither method works well with poor choice of hash function.
  - If contiguous allocation not possible
    ⇒ can implement a directory version of linear hashing.
  - Although examples started with using a single bit, most often, initial hash values start with more bits (e.g., 8 bits).
  - Hash files are slow for scans, range search.

4.32 External Sorting

- Popular sorting techniques:
  - Insertion sort.
  - Heapsort.
  - Mergesort.
  - Quicksort.

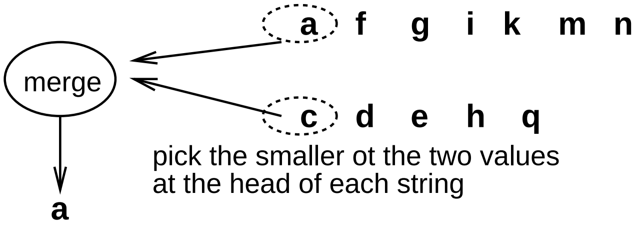
  These are all *internal* sorting methods
  ⇒ data is in memory.

- In an *external* sorting problem:
  - Data is too large to fit into main memory.
  - Data must be sorted in pieces.
  - Data is usually a heapfile of records.
  - Desired end result: a sorted file (sorted on some key).

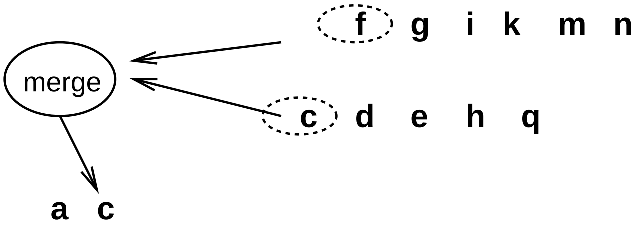
- Key ideas in *external sorting*:
  - Individual blocks can be easily sorted
    ⇒ read them in and use an internal sorting method.
  - Create *runs*:
    - A *run* is a group of sorted blocks in sort order (a sorted piece of a file).
    - Runs can be created by merging data from several blocks in memory.
  - Merge shorter runs into longer runs.
  - Finally, merge runs into final sorted result.

4.33 Binary Merge-Sort

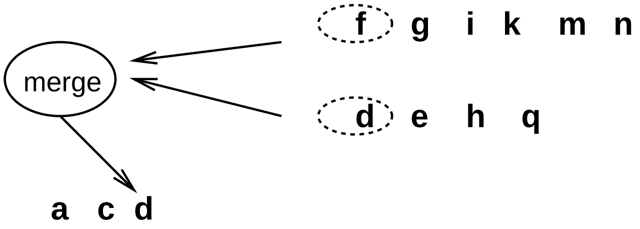
- The basic idea in merging can be conceptually explained as follows: e.g., Merge the two strings 'afgikmn' and 'cdehq' in alphabetical order
  Step 1


a f g i k m n
merge
c d e h q
pick the smaller ot the two values
at the head of each string
a

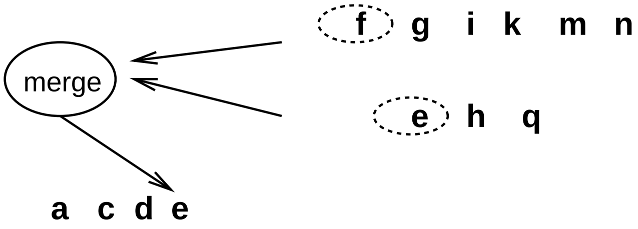
Step 2



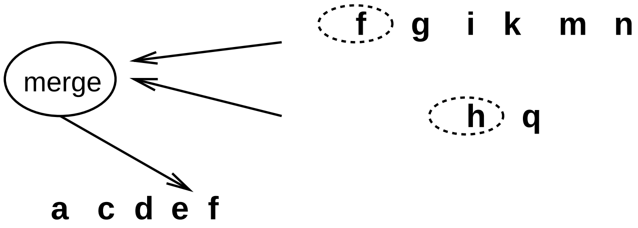
Step 3



Step 4

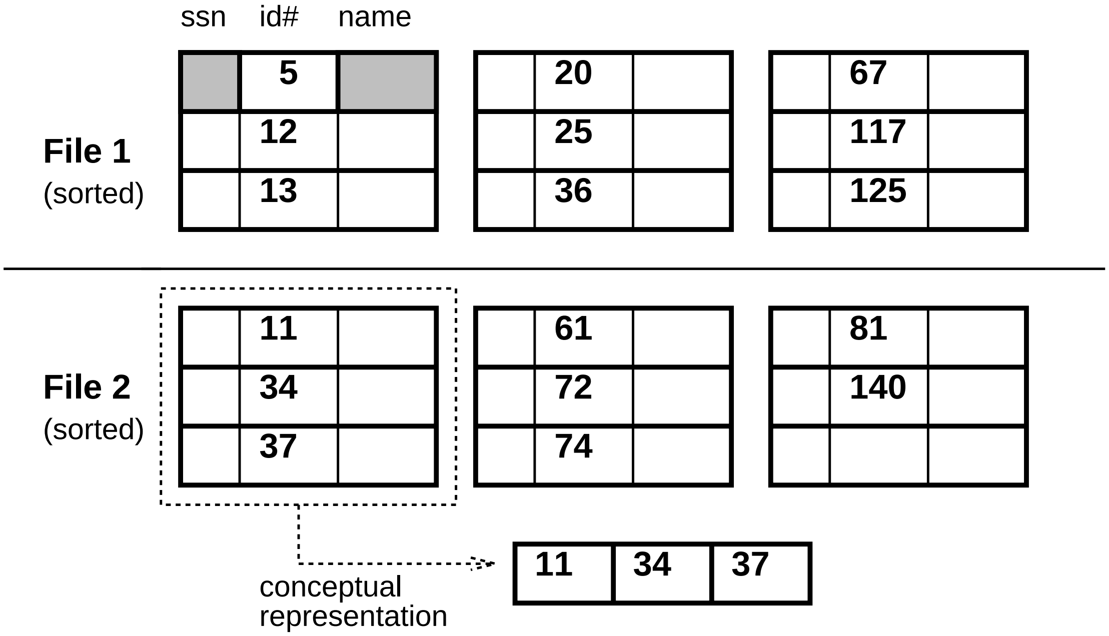


Step 5

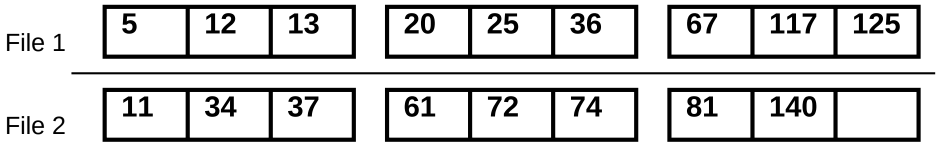


... and so on.

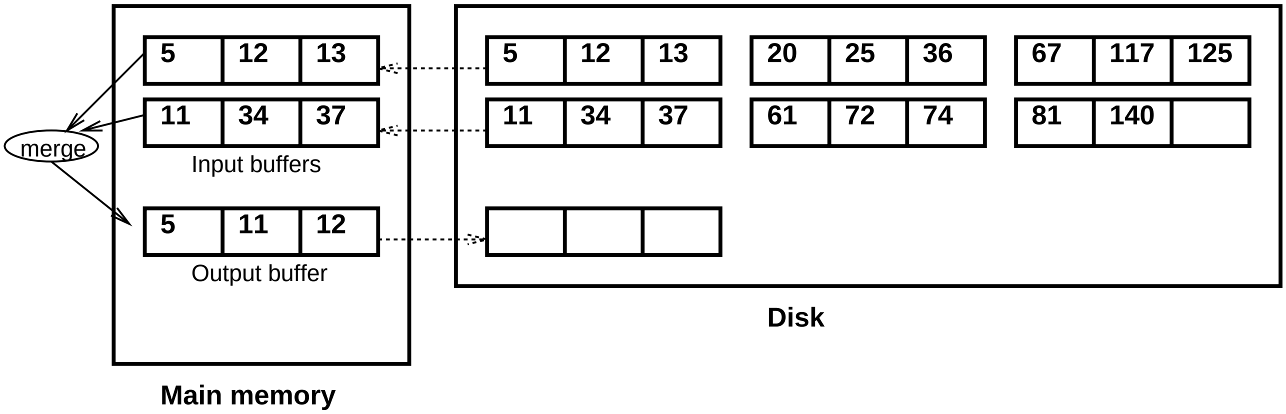
- Now suppose we have two sorted files (runs) that need to be merged: Suppose record structure is (SSN, ID#, NAME) and we are sorting by ID#:



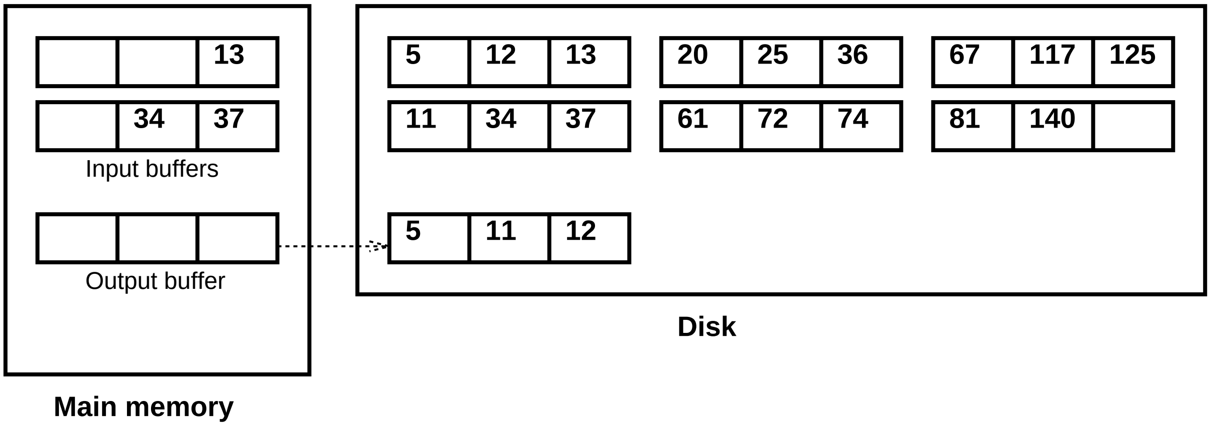
Conceptually,



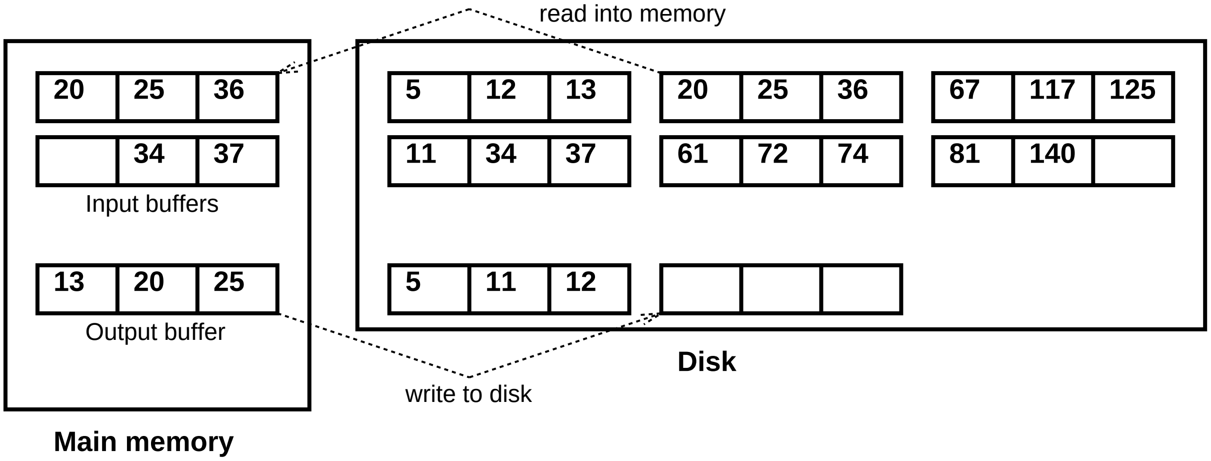
To merge these runs, we bring in two blocks at a time into memory:



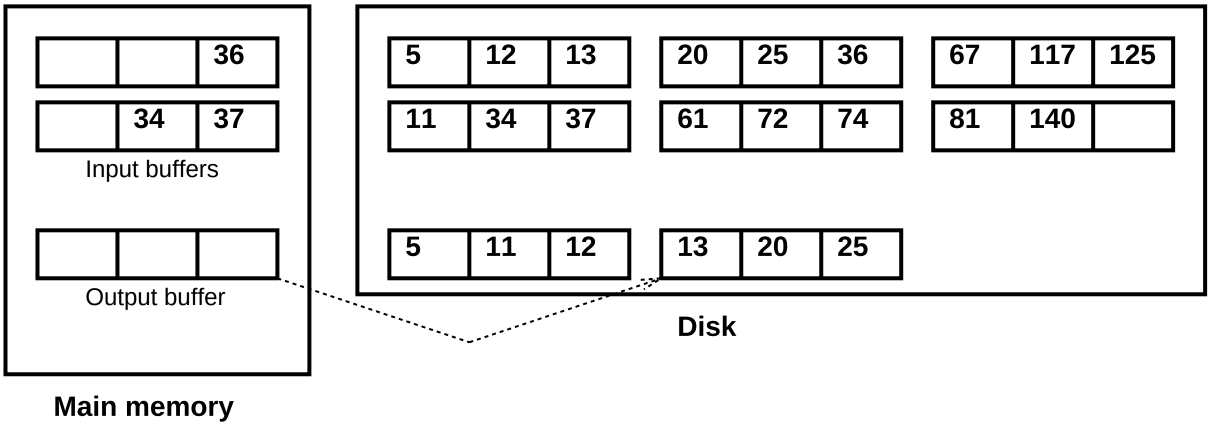
Write output block:



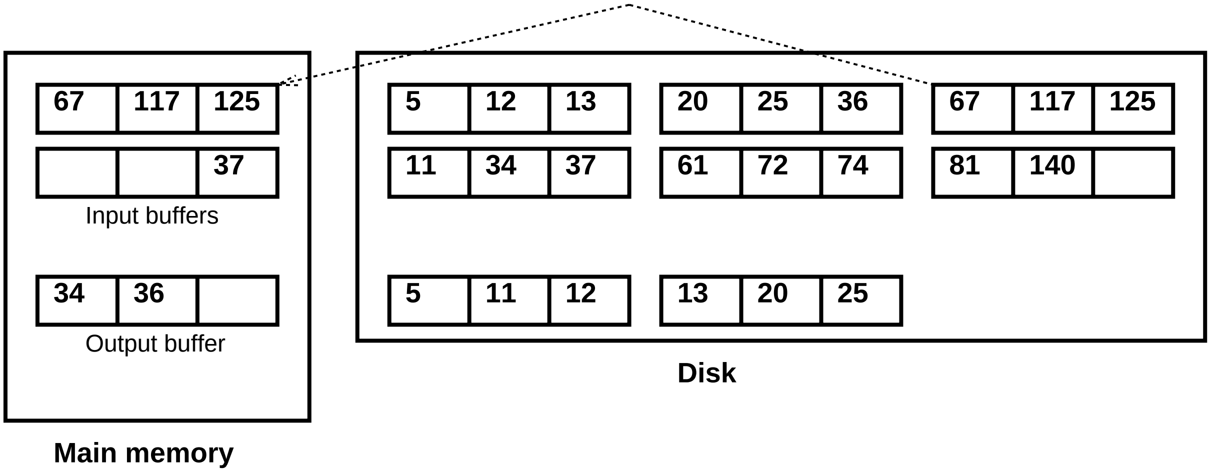
After ‘13’ is in the output buffer, we must read next block from File 1:



Write next output block:

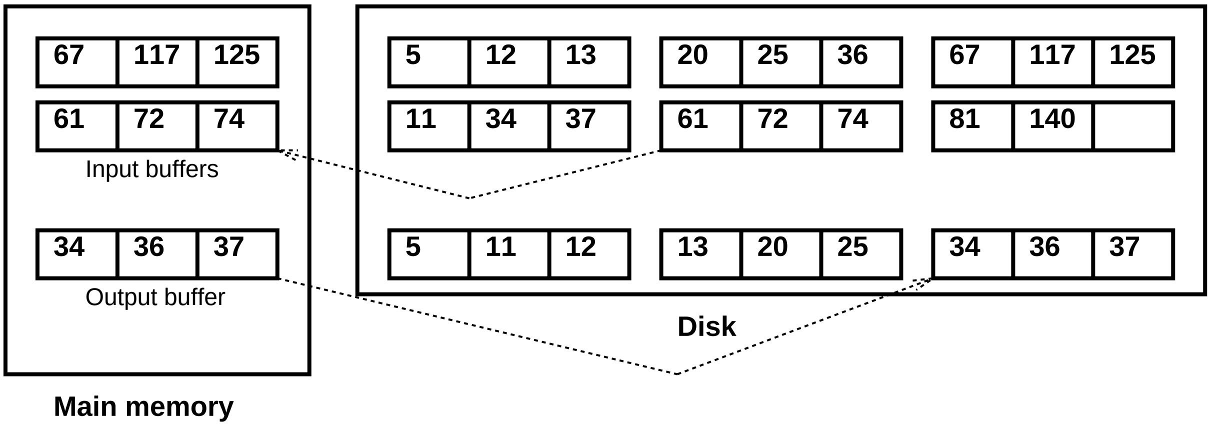


After ‘36’ is copied to output buffer, read next File 1 block:



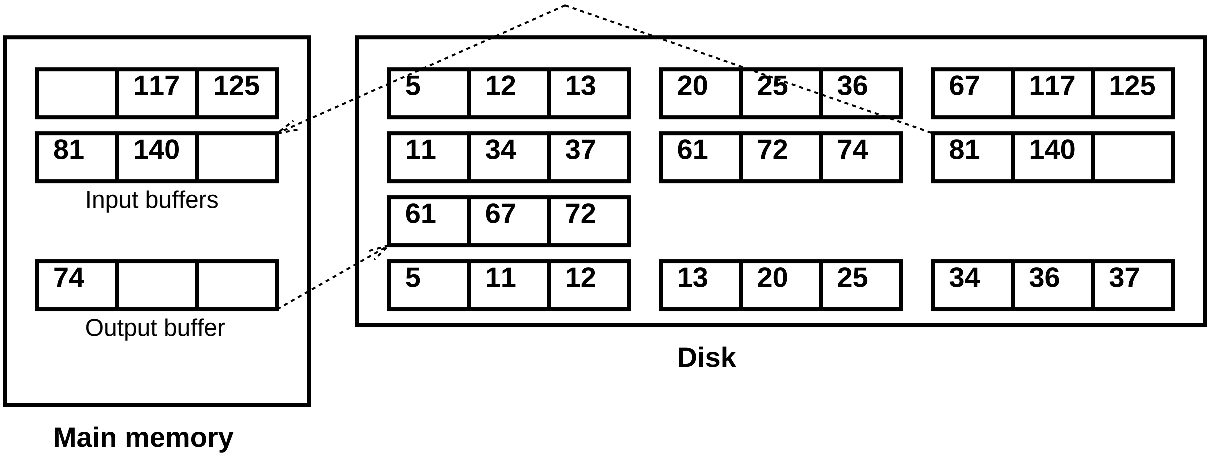
Next:

- Copy over ‘37’.
- Write current output block.
- Read next File 2 block.

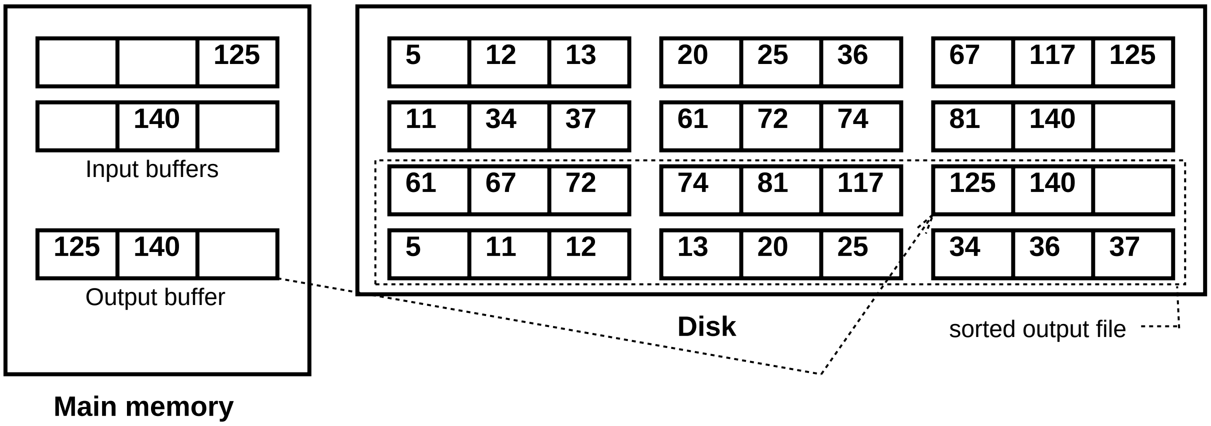


Next:

- Create next output block: ‘61-67-72’.
- Write output block.
- Copy ‘74’ over to output buffer
  ⇒ must read next File 2 block.



Continue in this fashion until finally:



- What does this have to do with sorting a single file?

  Key idea:

  – Break up file into smaller (sorted) runs.

  – Merge runs until a single file emerges.

- Example:
  Unsorted file:

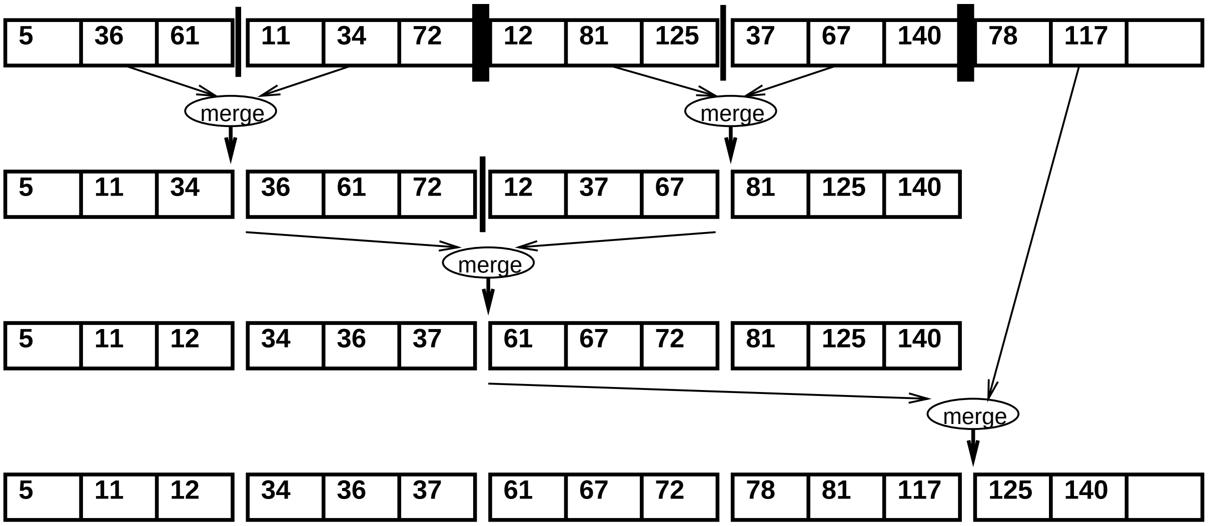
61	36	5	34	11	72	12	125	81	67	140	37	78	117	

  First sort individual blocks:

5	36	61	11	34	72	12	81	125	37	67	140	78	117	

  Next, repeatedly apply binary merges:

  - Divide file into groups of two blocks
    ⇒ 3 groups.
  - Sort each group using binary merge.
    ⇒ 3 sorted groups (runs).
  - Divide runs into groups of two runs
    ⇒ 2 groups.
  - Sort group using binary merge.
    ⇒ final result.



- How many steps (phases) does it take to sort a file of n blocks?

  For example, suppose $n = 400$:

  - Phase 0: sort individual blocks in file.
  - Phase 1: Create groups of 2 blocks and merge within groups.
    $\Rightarrow$ 200 groups (2 blocks each)
    $\Rightarrow$ 200 runs after merging (2 blocks each).
  - Phase 2: Group 200 runs into groups of 2 runs, then merge.
    $\Rightarrow$ 100 groups (2 runs each, of size 2 blocks)
    $\Rightarrow$ 100 runs (4 blocks each) after merging.
  - Phase 3: Group and merge:
    $\Rightarrow$ 50 groups (2 runs each, of size 4 blocks)
    $\Rightarrow$ 50 runs (8 blocks each) after merging.
  - Phase 4: 25 runs.
  - Phase 5: 13 runs (integral number of runs).
  - $\vdots$
  - Phase 8: 2 runs.
  - Phase 9: 1 run
    $\Rightarrow$ one sorted file.

  How many phases?
  $\Rightarrow$ 10 $(= 1 + 9)$.
  $\Rightarrow$ In general, $1 + \lceil \log_2 n \rceil$ phases.

  NOTE: we can avoid Phase 0 (sorting block contents during Phase 1).

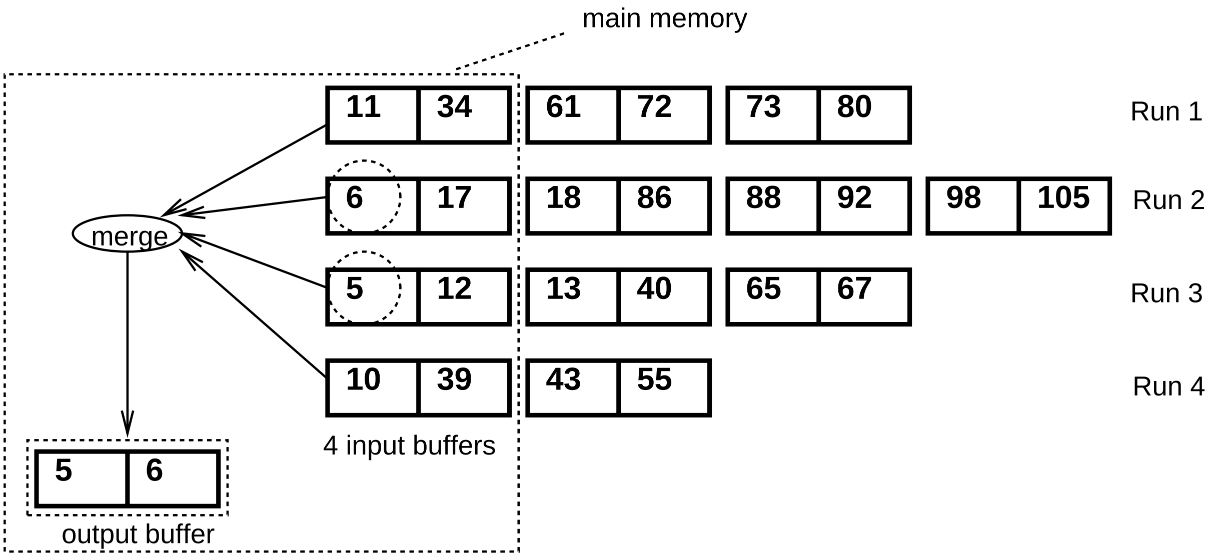
  In each phase: all blocks are read once, written once
  $\Rightarrow$ $2n\lceil \log_2 n \rceil$ block accesses.

- How much memory is needed?
  $\Rightarrow$ 2 blocks for input, 1 block for output
  $\Rightarrow$ only 3 blocks!

  Q: Can we do better by using more memory?

4.34 M-way Merge Sort

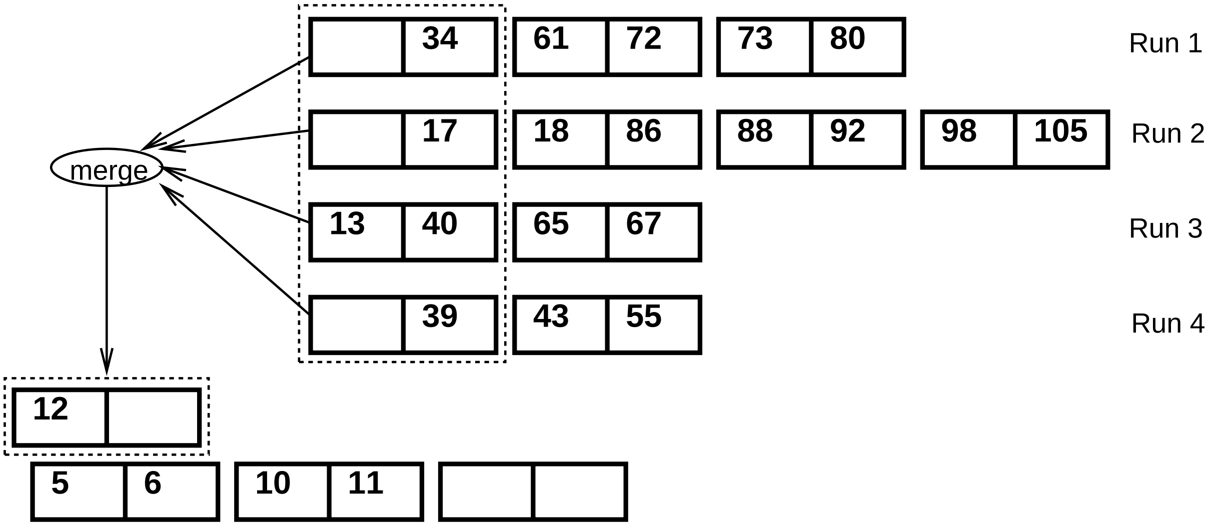
- A binary merge is easily generalized to an M-way merge:
  - Merge M input streams.
  - Use M input blocks and 1 output block.
- Example
  - 2 records per block.
  - M=4 runs.
    ⇒ 4 input buffers, 1 output buffer.



Next:

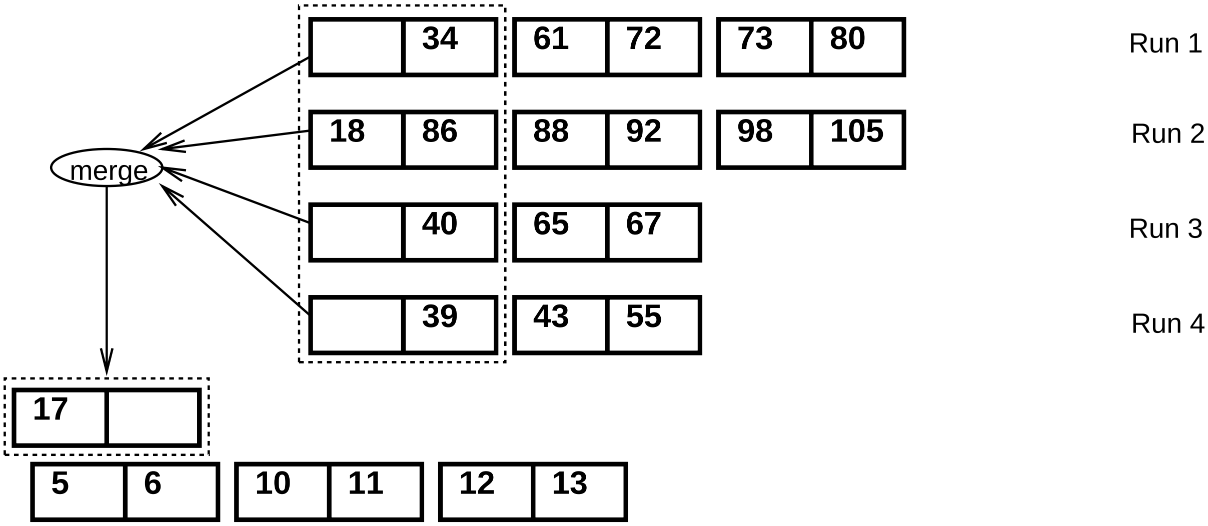
- Write '5-6' block.
- Copy '10' and '11' to output buffer.
- Write '10-11' block.

- Copy ‘12’ to output buffer.
- Read next block from Run 3.



Next:

- Copy ‘13’ over to output buffer.
- Write output buffer (‘12-13’ block).
- Copy ‘17’ to output buffer.
- Read next block from Run 2.



...and so on, until all runs are exhausted.

- Using an M-way merge to sort a file:

  Consider a file with n blocks.

  - Divide file into groups of M blocks each.
  - Merge each group into a sorted run
    $\Rightarrow \lceil \frac{n}{M} \rceil$ runs.
  - Create groups of M runs.
  - Merge each group into a single run...
  - ... and so on, until a single sorted file remains.

  For example, suppose $n = 400$, $M = 4$:

  - Phase 1: Create groups of 4 blocks
    $\Rightarrow$ 100 groups
    $\Rightarrow$ 100 runs (of 4 blocks each) after merging.
  - Phase 2: Create groups of 4 runs
    $\Rightarrow$ 25 groups
    $\Rightarrow$ 25 runs (of 16 blocks each) after merging.
  - Phase 3: 7 runs.
  - Phase 4: 2 runs.
  - Phase 5: 1 run (sorted file).

  In general: $\lceil \log_M n \rceil$ phases.
  $\Rightarrow 2n\lceil \log_M n \rceil$ block I/O's.

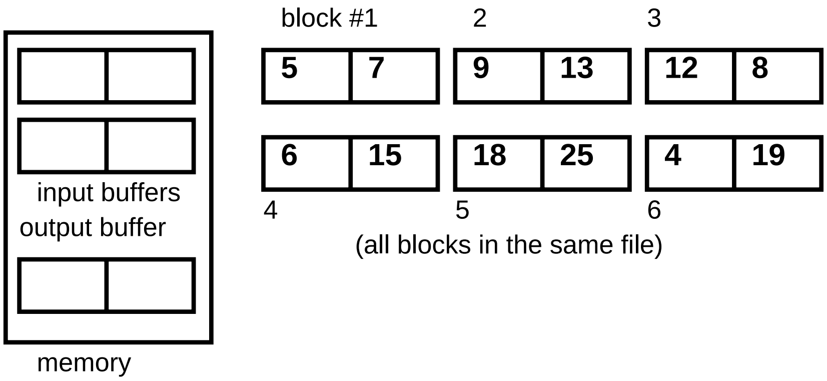
  Example: 200 Mb file, 1 Mb memory, 2K blocksize.
  $\Rightarrow$ 500 blocks of memory, 50000 file blocks
  $\Rightarrow$ 499-way merge
  $\Rightarrow$ 2 phases
  $\Rightarrow 2 \times 2 \times 50000$ block accesses
  $\Rightarrow$ 200,000 block accesses.

4.35 Sorting: Improvements

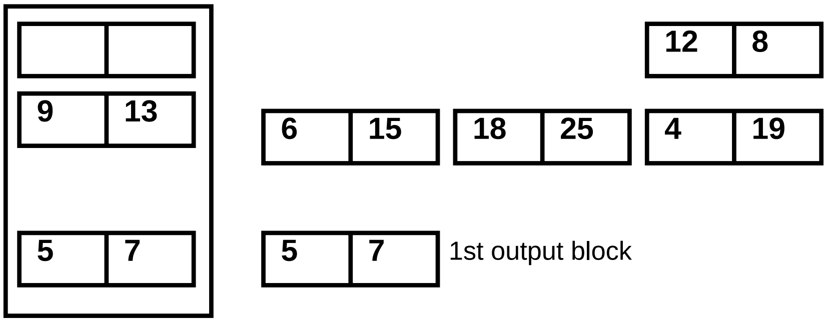
- In an M-way merge, how do we select the *least* element from the input buffers?
  - Naive approach: scan through blocks and find minimum each time.
  - What if M is large?
  - Better: use a heap
    ⇒ min-selection takes $O(\log_2 M)$.
- Partially sorted files:
  - Often, files are almost sorted (e.g., an initially sorted file after some random insertions and deletions).
  - What is the cost of sorting such a file?
  - Consider a fully-sorted file (worst-case):
    ⇒ merge-sort doesn't distinguish
    ⇒ takes as long as sorting any other file.
  - The problem: indiscriminate grouping.
- An alternative grouping method (that creates runs of variable length): the *snowplow method*.

  Example: M=2

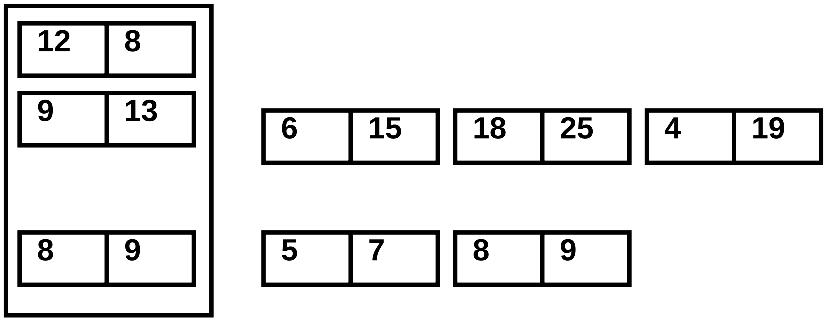
  Initially, the (single) file is on disk:



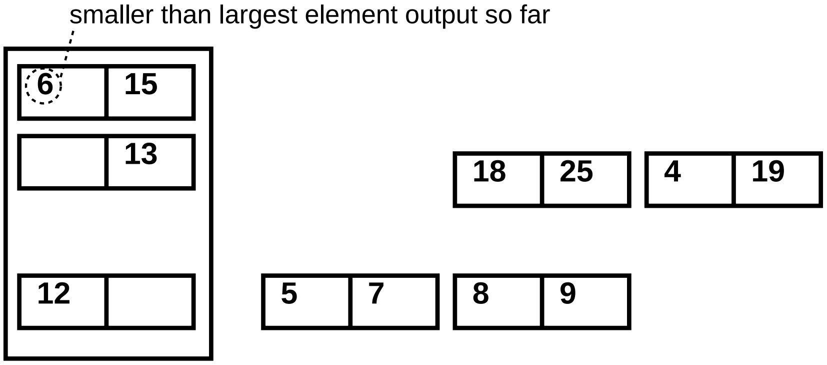
Read first two blocks in memory, merge, and output first block:



Now read in next block and create next output block (‘8-9’):

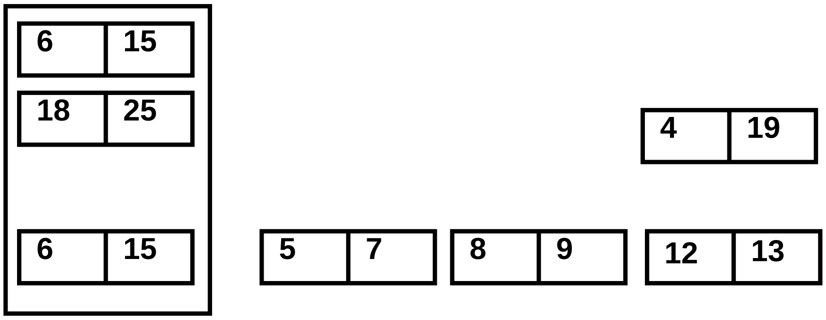


Copy ‘12’ to output buffer and read in next block:

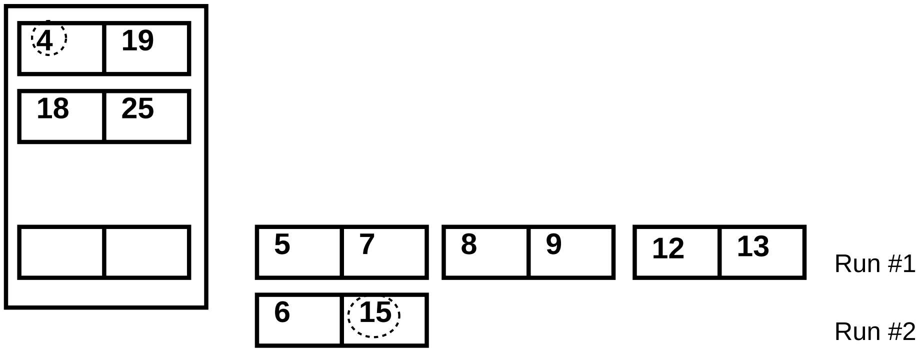


The '6' in '6-15' is smaller than the largest item already output
⇒ must end the current run and start a new one with '6'.

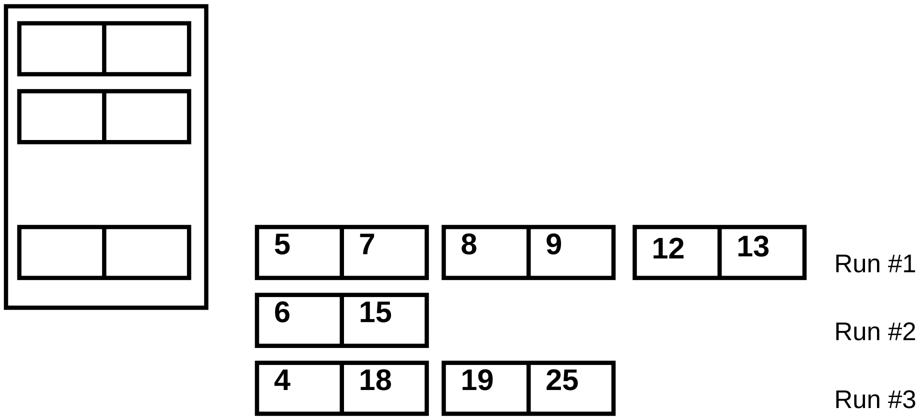
End current run with '12-13' and read in next block:



Output '6-15' and read in next block
⇒ '4' is less than '15'
⇒ run must end.



Finally, create the last run:



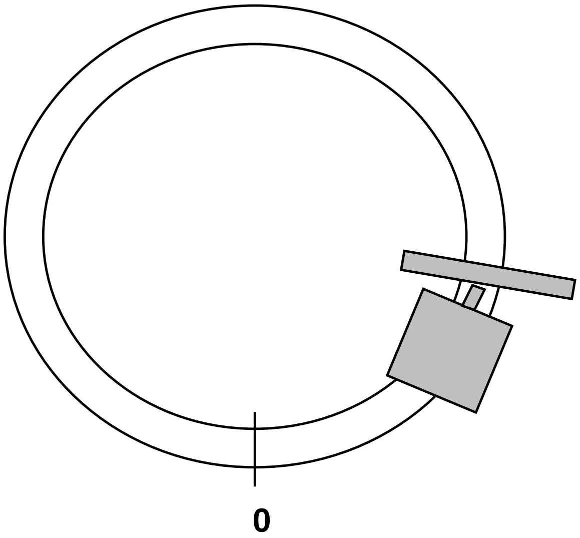
- What good is this approach?
  - A sorted file will pass through this method untouched (to result in a single run ⇒ a sorted file).
  - A mostly-sorted file will result in a few long runs.
  - Note: merging runs of uneven length is no different from merging runs of identical length.

  But consider a file sorted in reverse order
  ⇒ a new run every M blocks (M = # of input buffers)
  ⇒ average run length is M.

- What is the average length of a run?
  ⇒ somewhere between n (all blocks) and M.

  Let us obtain an approximate answer with some simplification:

  - Assume key values are real numbers in $[0, 1]$.
  - Assume uniform distribution of inputs.
  - Consider a snowplow plowing a circular road while it's snowing:



  - The circular road is the interval $[0, 1]$ bent around.
  - Associate a key value with each snowflake.

– Each snowflake falls at its designated place on the circle:
  ⇒ e.g., the value 0.5 falls at the top of the circle.

– The snowplow plows at a constant rate, the same rate at which snow falls.

– When the plow reaches the 0-mark:

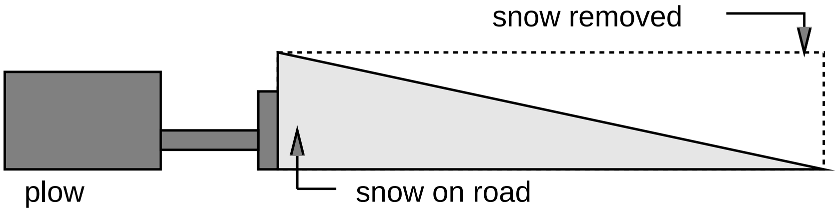
  ∗ A new run is started

  ∗ In the old run, the plow picked up numbers (snowflakes) that were increasingly larger.

  ∗ Smaller numbers fall behind the plow as it moves
    ⇒ they must be picked up in the next round.

– A run (of numbers in order) corresponds to the amount of snow removed in one round.

– Now consider a cross-section:



– Rate of snowfall equals rate of removal
(rate of data input to memory equals output rate).
  ⇒ a constant rate.

– How much snow is removed in one round?

  ∗ The plow always sees the maximum height immediately in front of it
    ⇒ plow removes one rectangle.

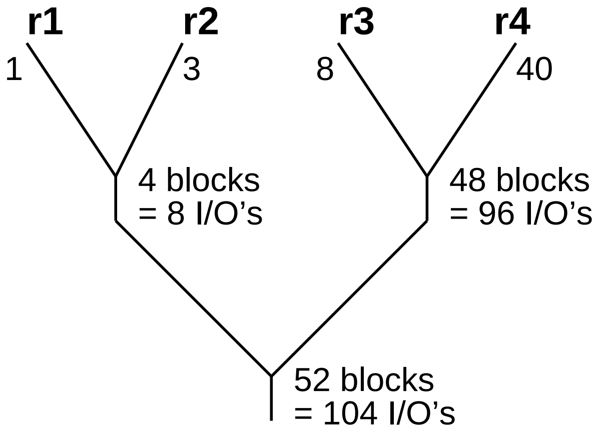
– But snow on track is one triangle
  ⇒ snow removed = 2 × snow present
  ⇒ run length = 2 × buffer space
  ⇒ average run length = $2M$.

Thus, the average run length is twice as long as the run lengths you get from the standard (grouping) method.

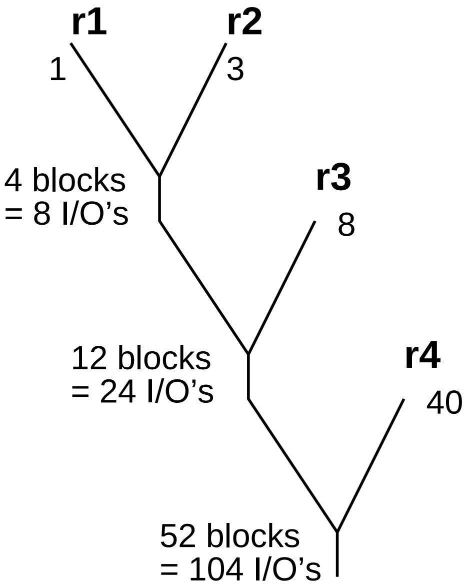
- Merging runs of different lengths:
  - Runs of different lengths can be merged in different ways.
  - Example: consider

run $r1$	1 block
run $r2$	3 blocks
run $r3$	8 blocks
run $r4$	40 blocks

  - Here are two ways of doing a *binary* merge:



Total I/O's: 8+96+104 = 208.



Total I/O's: 8+24+104 = 136 I/O's

  - The problem reduces to constructing optimal *weighted binary trees* ⇒ use *Huffman encoding* algorithm for binary case.

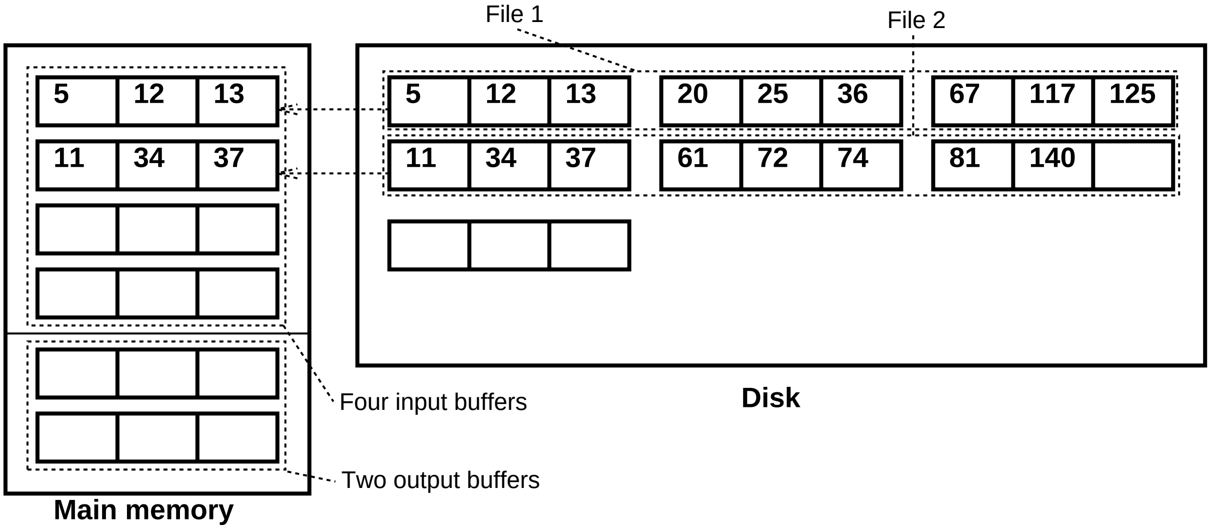
- Parallelism in I/O:
  - Suppose multiple disks are available
    ⇒ can read and write simultaneously.
  - Use 2 sets of input buffers and 2 sets of output buffers.
  - If $M+1$ buffers are available, choose K such that

$$\begin{aligned} 2+2K &\leq M+1 \\ K &\leq \frac{M-1}{2} \end{aligned}$$

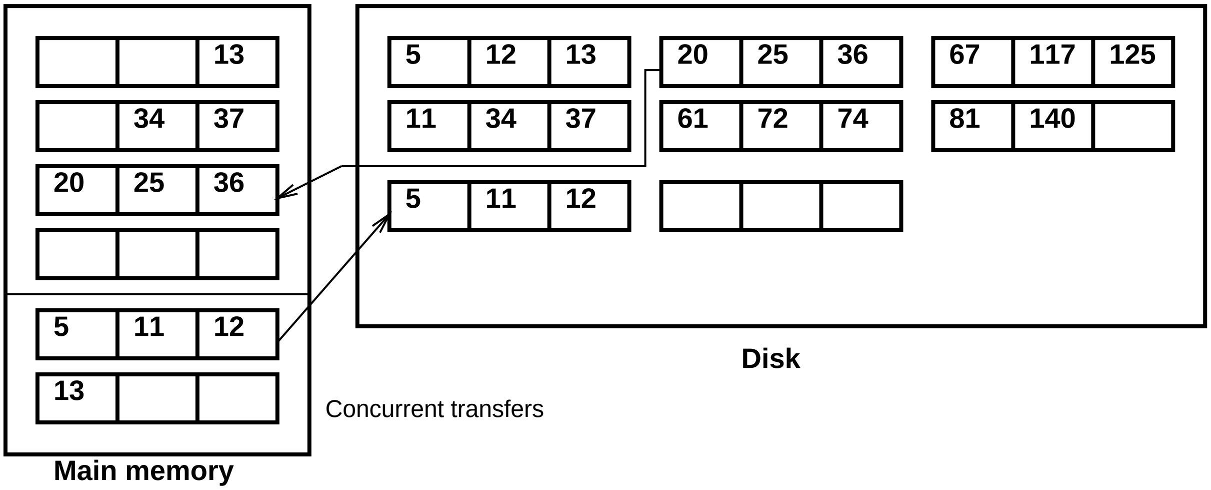
    ⇒ at best a K-way merge.
  - Example: $M = 6 \Rightarrow K = 2$.

- Example:

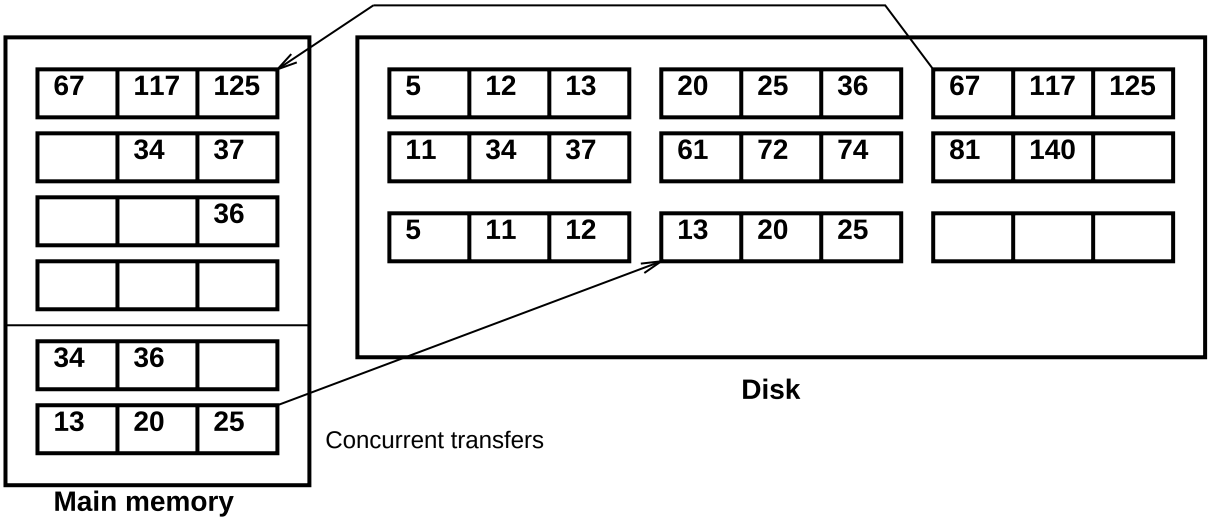
  Consider an example with $K = 2$. A block is read from each run.



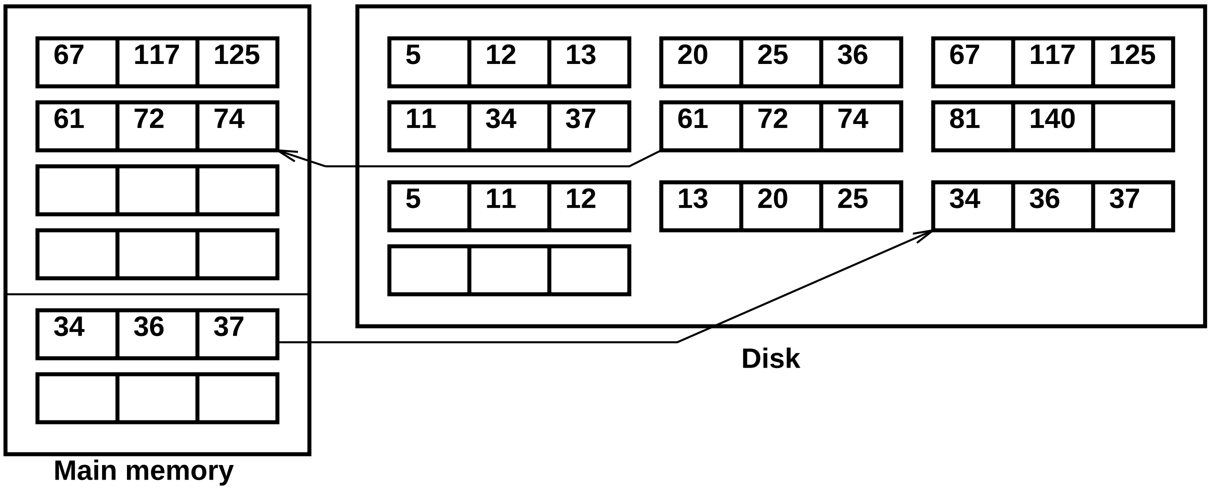
Next, while the first output block is being written, the next input block is read:



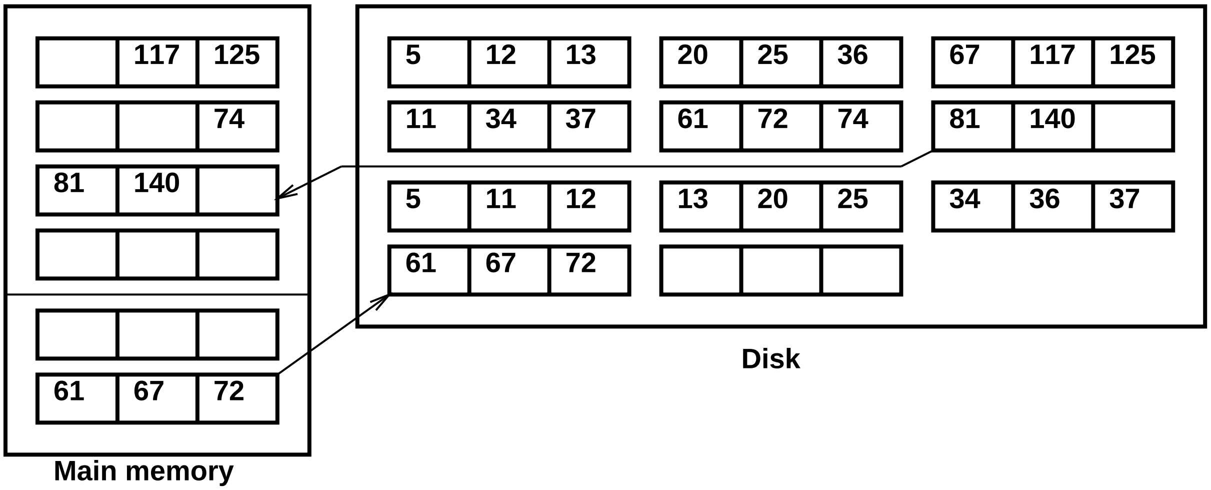
Write the ‘13-20-25’ block while reading in the ‘67-117-125’ block.



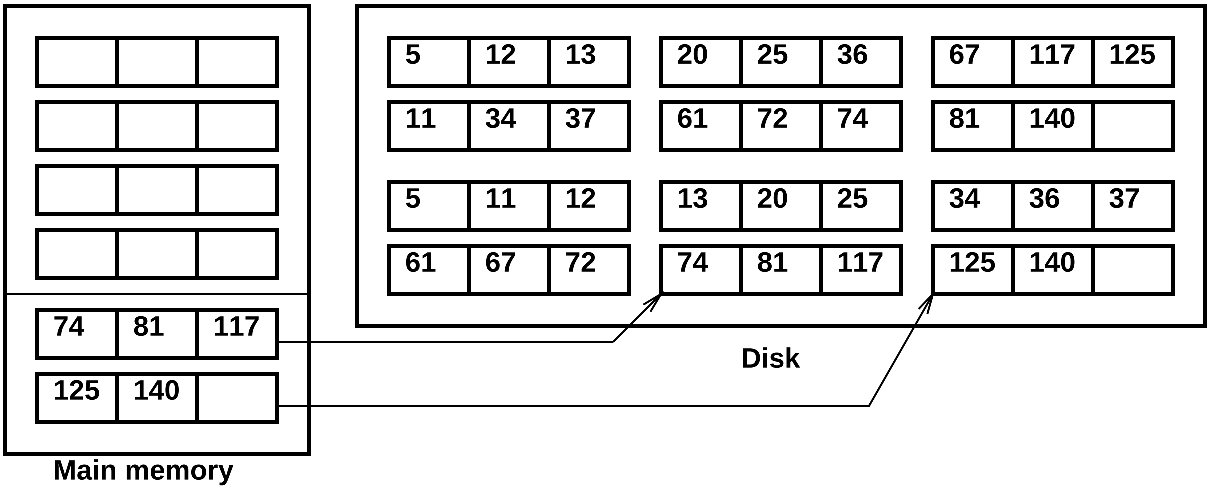
Write the ‘34-36-37’ block while reading in the ‘61-72-74’ block:



Write the ‘61-67-72’ block while reading in the ‘81-140’ block:



Finally,


74 81 117
125 140
Main memory
5 12 13
11 34 37
20 25 36
61 72 74
67 117 125
81 140
5 11 12
61 67 72
13 20 25
74 81 117
34 36 37
125 140
Disk

Chapter 5

Query Processing and Optimization

Course Notes on Database Systems

5.1 Query Processing: Introduction

- Consider the following relations:

  PASSENGER (NAME, SSN, FLT_ID, MILES)
  FLIGHT (FLT_ID, FLT_NO, STARTAPT, ENDAPT)
  AIRPORT (APT, NAME, CITY)

  and the following query: "List passengers flying into National airport that have at least 1000 miles along with their mileage".

  In SQL:

```
select  P.NAME, P.MILES
from    PASSENGER P, FLIGHT F, AIRPORT A
where   P.MILES > 1000
        and P.FLT_ID = F.FLT_ID
        and F.ENDAPT = A.APT
        and A.NAME = 'National';
```

  In relational algebra:

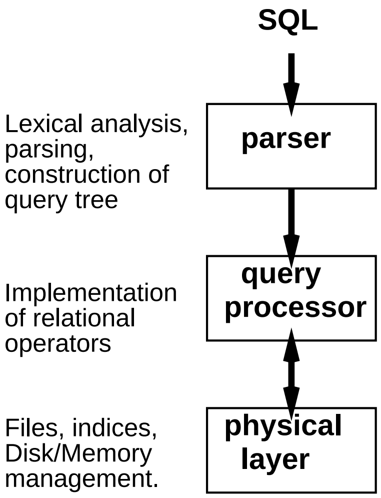
$$\Pi_{\text{P.NAME,P.MILES}}\ (\ \sigma_{\text{MILES>1000}}\ (\text{P} *\ \sigma_{\text{NAME='National'}}\ (\text{F} * \text{A})\ )\ )$$

  or

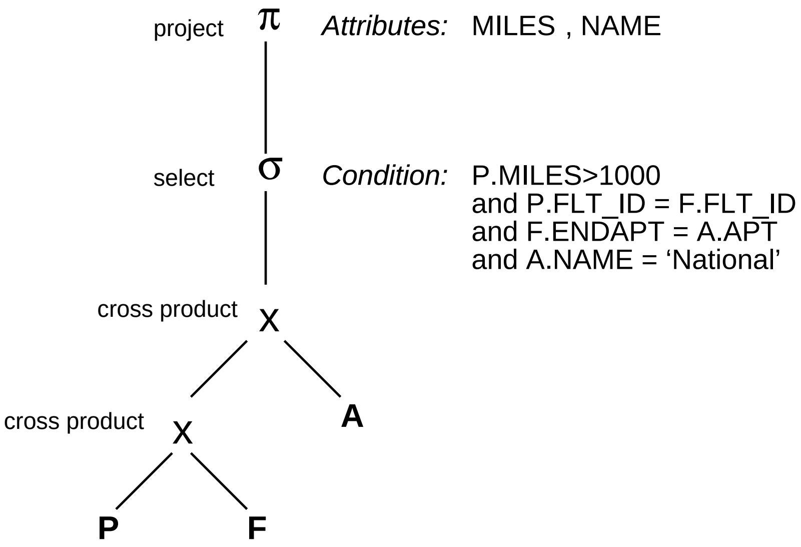
$$\Pi_{\text{P.NAME,P.MILES}}\ (\ \sigma_{\text{A.NAME='National'}}\ (\text{A})\ * \text{F} *\ \sigma_{\text{MILES>1000}}\ (\text{P})\ )$$

  (among several ways).

An SQL query is parsed by the *parser* to create a *query tree*, which is then executed by the *query processor*.


SQL
Lexical analysis, parsing, construction of query tree
parser
Implementation of relational operators
query processor
Files, indices, Disk/Memory management.
physical layer

A typical query tree produced by the parser:



NOTE:

- There are several ways of executing the same query tree.
- An *execution plan* specifies an order.
- A parser provides a simple tree.
- The query processor creates a better tree and an execution plan.

- For our example, assume the following sizes (with 1 Kb blocksize)

PASSENGER	50 bytes per tuple (20 tuples per block)
	100,000 tuples (5000 blocks)
FLIGHT	20 bytes per tuple (50 tuples per block)
	50,000 tuples (1000 blocks)

Assume: each block access takes 20 ms (milliseconds).

For example, a scan of PASSENGER requires 5000×20 ms $= 100$ seconds.

5.2 Implementing Selection

- First consider a single equality search, e.g.,

$$\sigma_{\text{NAME='Smith'}} \text{ (PASSENGER)}$$

1. Linear search on a heapfile:
   - 5000 blocks scanned (worst-case)
     $\Rightarrow$ 100 seconds.
   - If NAME is a *key*:
     $\Rightarrow$ 2500 blocks (average)
     $\Rightarrow$ 50 seconds.
2. Binary search on a sorted file (sorted by NAME):
   - Binary search takes $\lceil \log_2 n \rceil$ for a file of n blocks.
     $\Rightarrow \lceil \log_2 5000 \rceil$ blocks
     $\Rightarrow$ 13 blocks
     $\Rightarrow 13 \times 20$ ms
     $\Rightarrow$ 260 ms.
3. B+-tree index on NAME:
   - Suppose 100 index entries fit into a block
     $\Rightarrow 2m - 1 = 100$ (m = degree)
     $\Rightarrow m = 50$.
   - Leaf level has as many entries as tuples
     $\Rightarrow$ 100,000 entries
     $\Rightarrow$ 1000 leaf blocks (best-case: all packed)
     $\Rightarrow$ 2000 leaf blocks (worst-case)
   - How many B+-tree levels needed (worst-case)?
     $\Rightarrow$ Recall: $2m^{k-1}$ at level k
     $\Rightarrow$ we want least value of k such that $2m^{k-1} \geq 2000$

⇒ $k = 3$ (levels 0,1,2,3)
⇒ 4 levels

- Access time?
  ⇒ 5 accesses in all (data block requires 1 access)
  ⇒ 5 × 20 ms
  ⇒ 100 ms.

4. Hash index on NAME:
   - Assume each chain has 2 blocks (1 overflow block).
     ⇒ 1 block access for directory, 2 for data
     ⇒ 3 block accesses
     ⇒ 3 × 20 ms
     ⇒ 60 ms.

- Single range search, e.g.,

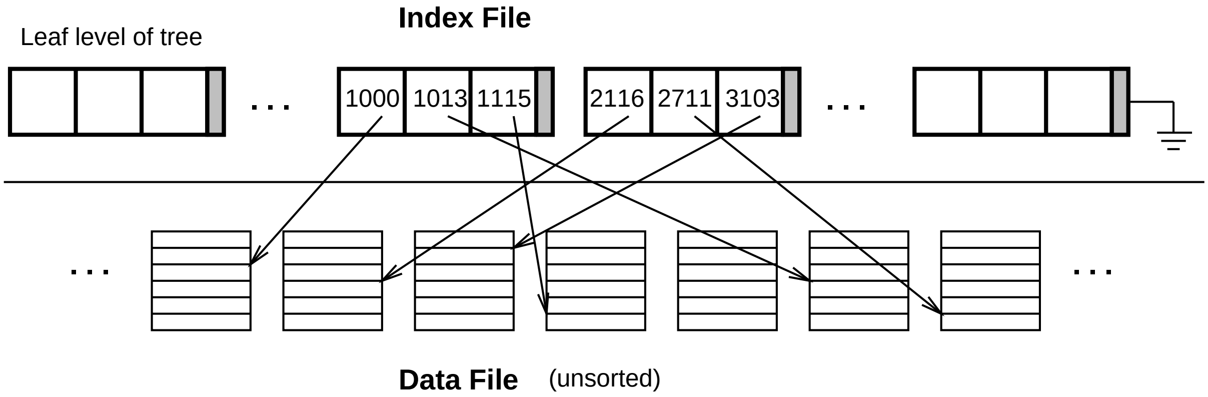
$$\sigma_{\text{MILES}>1000} \quad (\text{PASSENGER})$$

Suppose 2000 customers (tuples) satisfy the condition MILES > 1000.

1. Scan
   ⇒ 100 seconds.
2. Hash index is useless in this case.
3. Sorted file (sorted on MILES):
   - $\lceil \log_2 n \rceil$ accesses to get first tuple
     ⇒ 13 blocks (from before)
   - Scan from first tuple onwards.
     ⇒ 2000/20 = 100 blocks (20 tuples per block).
   - Total:
     ⇒ 113 blocks
     ⇒ 113 × 20 ms
     ⇒ 2.26 seconds.

4. B+-tree on MILES:

   - Since MILES is not a key, we need to distinguish between *clustered* and *unclustered* versions of the tree.
   - In an *unclustered* version, the data file is not sorted nor is there a sort-index on the data file.
   - The problem with no clustering:

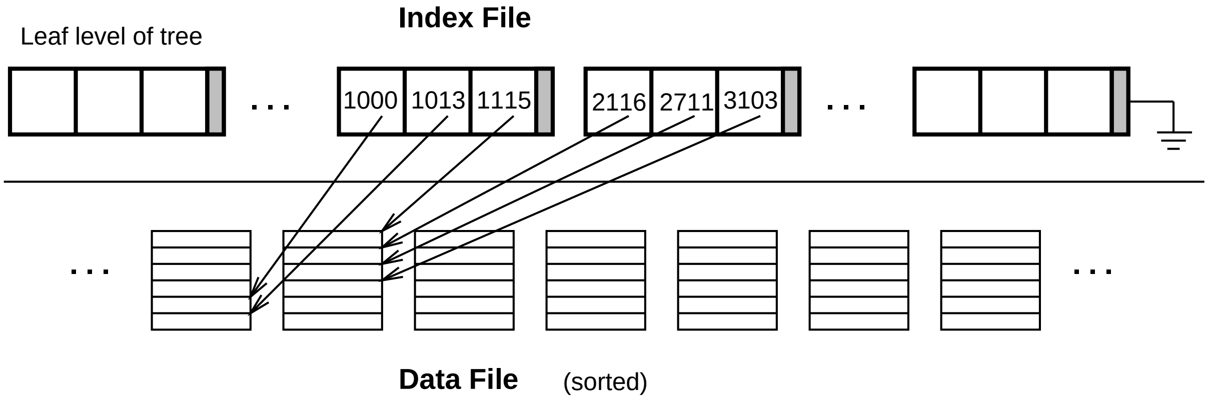


     * Worst-case, each index entry could point to a different data block.
     * In our example: 2000 data tuples satisfy MILES > 1000
       ⇒ worst-case 2000 data blocks retrieved.

   - Cost for *unclustered* version:
     * 4 block accesses to get to first leaf tuple
     * 20 index blocks scanned at leaf level (at 100 index entries per block)
     * 2000 blocks to retrieve tuples from data

   Total:
   ⇒ 2024 blocks
   ⇒ 2024 × 20 ms
   ⇒ 40.48 seconds.

– In a *clustered* version, the data file is sorted:



– Cost for a *clustered* version:
⇒ data will lie in successive blocks
⇒ 100 blocks accessed
⇒ 124 blocks overall (including tree blocks)
⇒ 124 × 20 ms
⇒ 2.48 seconds.

– Important: must be careful when using an unclustered B+-tree for range search:

- ∗ Range search is effective in a B+-tree when the number of identical values is small.
- ∗ Otherwise, range search can be more costly than a scan.

- Select's with multiple conditions – CNF representation:
  - The select condition is a boolean expression, e.g.,

    (MILES > 1000) **and** (NAME = 'Smith')

  - Every boolean expression in relational algebra consists of *terms* of the form:
    - `<attr>` **op** `<value>`, or
    - `<attr>` **op** `<attr>`.

    where **op** is a comparison operator $(=, <, >, \leq, \geq)$.
  - A boolean *expression* is a collection of terms joined by *boolean operators* (**and, or, not**).
  - Any boolean expression can be re-written in *Conjunctive Normal Form* (CNF).
  - A CNF boolean expression is a collection of *conjuncts* that are **and**'ed:

    `(conjunct1)` **and** `(conjunct2)` **and** `(conjunct3)` ...

    where each `conjunct` only has *or*'s or **not**'s, e.g.:

    `conjunct3` = (`term1` **or** `term2`).

    Thus, for example, the non-CNF expression

    (AGE<15 **and** HEIGHT>6) **or** WEIGHT>400

    is transformed into the CNF expression:

    (AGE<15 **or** WEIGHT>400) **and** (HEIGHT>6 **or** WEIGHT>400)

- Implementing *select* for CNF's without **or**'s:

  Consider a CNF expression with only terms as conjuncts:

  `(term1)` **and** `(term2)` **and** `(term3)` ...

  Example:

  $$\sigma_{\text{(MILES > 1000) and (NAME = 'Smith')}} \text{ (PASSENGER)}$$

Assume:

- 45 Smiths (3 blocks)
- only one tuple satisfies the condition.

1. Unsorted heapfile:
   ⇒ scan required
   ⇒ 5000 blocks scanned
   ⇒ 100 seconds.
2. Sorted file (on NAME):
   - Use binary search to find first Smith tuple
     ⇒ 13 blocks.
   - Scan remaining Smith-tuples and check MILES condition
     ⇒ 3 blocks.
   - Cost:
     ⇒ 16 blocks
     ⇒ 16 × 20 ms
     ⇒ 320 ms.
3. Sorted file (on MILES):
   - Use binary search to find first 1000-mile tuple
     ⇒ 13 blocks.
   - Scan remaining 1000-mile tuples for Smith's
     ⇒ 100 blocks scanned (2000 customers satisfy condition)
   - Cost:
     ⇒ 113 blocks
     ⇒ 113 × 20 ms
     ⇒ 2.26 seconds.

4. Hash index on NAME:
   - Search `hashvalue('Smith')` bucket for MILES condition
     ⇒ at least 3 blocks, probably more, say 6.
   - 1 block for directory access.
   - Cost:
     ⇒ 7 blocks
     ⇒ 7 × 20 ms
     ⇒ 140 ms.
5. A hash index would not be used on MILES (useless for range search).
6. B+-tree index on NAME:
   - 4 tree blocks during search, 1 possible additional block at leaf (45 Smith *index* tuples must fit into 2 blocks).
   - *Unclustered* version
     ⇒ 45 Smith tuples
     ⇒ 45 data blocks (worst-case)
     ⇒ 50 blocks overall
     ⇒ 50 × 20 ms
     ⇒ 1 second.
   - *Clustered* version
     ⇒ 3 data blocks
     ⇒ 8 blocks overall
     ⇒ 8 × 20 ms
     ⇒ 160 ms.

7. B+-tree index on MILES:
   - 4 tree blocks accessed, 20 on leaf level.
   - *Unclustered* version:
     ⇒ 2000 blocks (worst-case)
     ⇒ 2024 blocks overall
     ⇒ 2024 × 20 ms
     ⇒ 40.48 seconds.
   - *Clustered* version:
     ⇒ 100 data blocks
     ⇒ 124 blocks overal
     ⇒ 124 × 20 ms
     ⇒ 2.48 seconds.
8. Suppose a B+-tree is available for each attribute:
   - Use the more selective value (NAME, in this case).

- expressions with **or**'s:
  - Disjunctions (**or**'s) are difficult to optimize.
  - Example:

$$\sigma_{(\text{MILES} > 1000)\ \textbf{or}\ (\text{NAME} = \text{`Smith'})}\ (\text{PASSENGER})$$

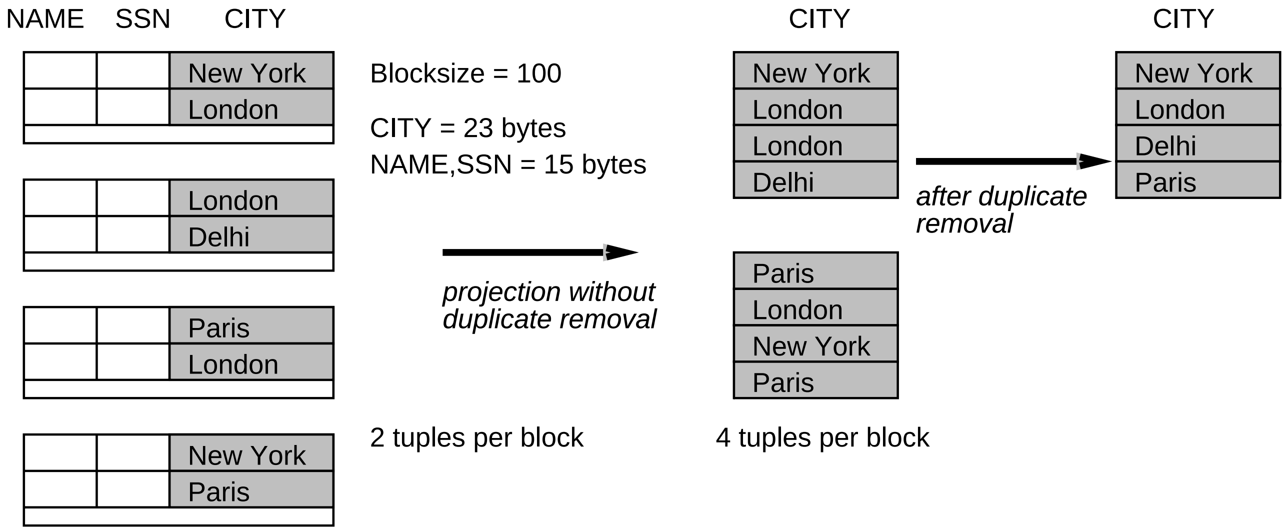
  - If no index is available on MILES
    ⇒ scan required
    ⇒ why bother with an index on NAME (since scan is required)?
    ⇒ during scan, also check NAME='Smith'.
  - If indices are available on both
    ⇒ use indices to retrieve on each condition
    ⇒ union the results.

5.3 Implementing Projection

- Two actions are required in implementing projection:

  1. Scan through relation and extract desired attributes.
  2. Remove duplicates.

  Example: Consider EMP (NAME, SSN, CITY) and the query Π_{CITY} (EMP) :



- Observe:
  - Step 1 (projection) is straightforward.
  - Step 2 (duplicate removal) requires more work.
  - Often, parts of Step 2 are integrated into Step 1.
  - Sometimes duplicate removal is not needed (if not specified with a **distinct** in SQL).

- Duplicate removal via sorting:

– Suppose we want to remove duplicates from a file:

  * Sort the file.
  * Scan sorted file and remove duplicates by comparing successive tuples.

– Note: the first set of runs (for sorting) can be created during Step 1 (while writing the output of projection).

– Cost estimate:
  $\Rightarrow 2n\lceil\log_M n\rceil$ block accesses for sorting

– Example:

$$\Pi_{\text{NAME}} \text{ (PASSENGER)}$$

  Suppose

  * $M = 100$ (100 blocks of input buffers).
  * The NAME field is 20 bytes long
    $\Rightarrow$ 50 tuples per block
    $\Rightarrow$ 2000 blocks for projected file.

– Cost:
  $\Rightarrow 2 \times 2000 \times \lceil\log_{100} 2000\rceil$
  $\Rightarrow$ 8000 block accesses
  $\Rightarrow 8000 \times 20$ ms
  $\Rightarrow$ 2.66 minutes.

– Using the snowplow method would be faster:
  $\Rightarrow$ run sizes approximately $2M = 200$ blocks
  $\Rightarrow$ 2000/100 such runs
  $\Rightarrow$ only one additional pass required (100 input buffers)
  $\Rightarrow$ 4000 block accesses
  $\Rightarrow 4000 \times 20$ ms
  $\Rightarrow$ 1.33 minutes.

- Duplicate removal using Hashing:
  - Basic idea:
    1. Scan file and hash records into a hash file.
    2. Scan individual buckets are remove duplicates.
  - Note: hashing has to be carefully implemented (using a lot of memory)

    e.g., In above example: file has 2000 blocks (100,000 tuples)
    - Worst-case: 2 block accesses for each insertion

      ⇒ 100,000 insertions

      ⇒ 200,000 block accesses

      ⇒ 200,000 × 20 ms

      ⇒ 66.66 minutes

      ⇒ wrong way to use hashing.
  - If M buffers are available, use a hash function with M buckets and allow overflow

    ⇒ hash file blocks are written only when full

    ⇒ block accesses ≈ number of file blocks
  - In practice, slight non-uniformity will cause slightly higher I/O.
  - Example: 2000 blocks

    ⇒ 2000 (approx.) hash file blocks.

    ⇒ 2000/100 = 20 blocks per bucket (19 overflow blocks)

    ⇒ each bucket fits into memory entirely

    ⇒ duplicates can be removed in one scan

    ⇒ 2000 + 2000 block accesses

    ⇒ 4000 × 20 ms

    ⇒ 1.33 minutes.
  - Note: creating the hash file can be done on-the-fly as the projection is being computed.
  - An advantage of sort-based projection: a nice side-effect is that the result is sorted.

5.4 Implementing Joins

- Joins vs. cross-products:
  - Every join can be expressed as a cross-product, e.g.,

    $$\text{PASSENGER} * \text{FLIGHT}$$

    is the same as

    $$\sigma_{\text{PASSENGER.FLT\_ID} = \text{FLIGHT.FLT\_ID}} \ (\text{PASSENGER} \times \text{FLIGHT})$$
  - SQL parsers typically only report cross-products.
  - Cross-products are large
    ⇒ system should recognize a join where possible.
- We will use the following example:

  $$\text{PASSENGER} * \text{FLIGHT}$$

  where

PASSENGER has	n_P = 5000 blocks, 20 tuples per block
	r_P = 100,000 tuples
FLIGHT has	n_F = 1000 blocks, 50 tuples per block
	r_F = 50,000 tuples

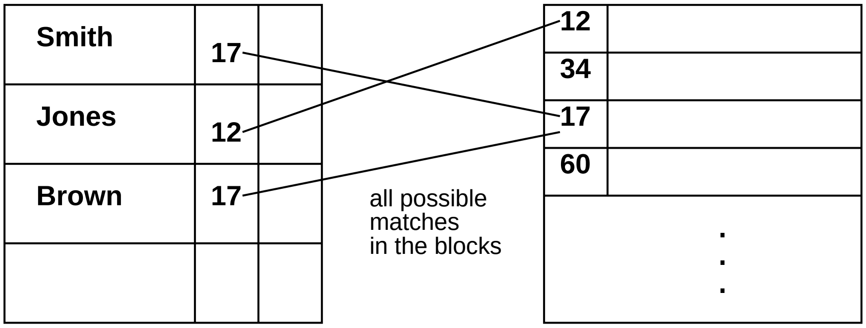
- Several ways to implement a join:
  1. Simple nested loops.
  2. Block-nested loops.
  3. Nested loops with indices.
  4. Sort-merge join.
  5. Hash-join.

- Simple nested loops.
  - Scan tuples in one file and for each one, scan the other file to find matching tuples.
  - For example, suppose we scan PASSENGER in the outer loop:

```
Algorithm:    Simple Nested Loops

for each tuple $x \in$ PASSENGER
    for each tuple $y \in$ FLIGHT
        if $x$.FLT_ID = $y$.FLT_ID
            put joined tuple in result
```

  - What is the cost?
    ⇒ 100,000 tuples of PASSENGER (in 5000 blocks)
    ⇒ 100,000 scans of FLIGHT (1000 blocks per scan)
    ⇒ $100{,}000 \times 1000 = 10^8$ blocks (of FLIGHT)
    ⇒ $10^8 + 5000$ blocks overall
    ⇒ 555.58 hours!
  - What if FLIGHT were the outer relation?
    ⇒ 50,000 tuples of FLIGHT (in 1000 blocks)
    ⇒ 50,000 scans of PASSENGER
    ⇒ $50{,}000 \times 5000$ blocks of PASSENGER
    ⇒ $25 \times 10^8 + 1000$ blocks overall
    ⇒ 1388.89 hours!
    ⇒ it's worse!
  - Using simple nested loops is a really stupid way of implementing a join.

- Block-nested loops:
  - Consider reading the first block of PASSENGER.
  - When we read the first block of FLIGHT, we can join all possible combinations in each block:



First PASSENGER block

First FLIGHT block

  - Then, we read the next block of FLIGHT and find all possible matches in the first block of PASSENGER. Then, the next block of FLIGHT
    ... and so on until all blocks of FLIGHT are joined with the
  - Then, the next block of PASSENGER is read and joined with the first, then second, then third ... etc blocks of FLIGHT.
  - ... And so on until all of PASSENGER has been scanned.
  - In pseudocode:

```
Algorithm:    Block-Nested Loops

for each block ∈ PASSENGER
    for each block ∈ FLIGHT
        if a pair in each block matches
            put joined tuple in result
```

  - Why does this work?
    ⇒ each pair of tuples needs to be tried against each other only once.

- Cost:
  - ⇒ 5000 blocks of PASSENGER
  - ⇒ 5000 scans of FLIGHT
  - ⇒ 5000 × 1000 blocks
  - ⇒ 5,005,000 blocks overall
  - ⇒ 27.78 hours
- In general: $n_P + n_P * n_F$.
- How much memory has been used?
  - ⇒ 2 input buffers (1 for each file).
- Can we do better with more buffers?
- Example: suppose we have $M = 100$ input buffers.
  - ⇒ use K buffers for PASSENGER and $M - K$ for FLIGHT.

  Suppose PASSENGER is outer relation:

  - ∗ Read first K blocks of PASSENGER.
  - ∗ Scan through all of FLIGHT and match tuples.
  - ∗ Read next K blocks of PASSENGER.
  - ∗ Scan through all of FLIGHT and match tuples.
  - ∗ ... and so on.
- Cost?
  - ⇒ $\frac{n_P}{K}$ groups of PASSENGER (exact: $\lceil\frac{n_P}{K}\rceil$)
  - ⇒ $\lceil\frac{n_P}{K}\rceil$ scans of FLIGHT
  - ⇒ $\lceil\frac{n_P}{K}\rceil * n_F$ blocks
  - ⇒ totally $n_P + \lceil\frac{n_P}{K}\rceil * n_F$
- What choice of K is optimal?
  - ⇒ as large as possible: $K = M - 1$
- In our example:
  - ⇒ $5000 + \lceil\frac{5000}{99}\rceil * 1000$ blocks
  - ⇒ 56000 blocks overall
  - ⇒ 56000 × 20 ms
  - ⇒ 18.6 minutes
  - ⇒ a great improvement!

- Can we do better?
  - $\Rightarrow$ use FLIGHT as outer relation
  - $\Rightarrow 1000 + \lceil \frac{1000}{99} \rceil * 5000$ blocks
  - $\Rightarrow$ 56000 blocks overall
  - $\Rightarrow$ same as before.
- Note: suppose FLIGHT had 90 blocks: With PASSENGER outside:
  - $\Rightarrow 5000 + \lceil \frac{5000}{99} \rceil * 90$ blocks
  - $\Rightarrow$ 9590 blocks. With FLIGHT outside:
  - $\Rightarrow 90 + \lceil \frac{90}{99} \rceil * 5000$ blocks
  - $\Rightarrow$ 5090 blocks (almost half the time)
  - $\Rightarrow$ use smaller relation in outer loop.

- Nested loops with indices:
  - If there's an index on the join attribute, we can use it.
  - Example: suppose we have an index on PASSENGER.
  - Use PASSENGER as inner relation:

```
Algorithm:    Nested Loops with Indices

for each block in FLIGHT
    for each tuple in the current block
        extract key;
        search for corresponding tuple in PASSENGER using index
        place joined tuple if match occurs
    endfor
```

  - Cost:
    - FLIGHT is scanned once
      $\Rightarrow n_F = 1000$ blocks.
    - For each *tuple* of FLIGHT, we probe index
      $\Rightarrow r_F \times$ number of blocks per access
    - The actual cost depends on the type of index.

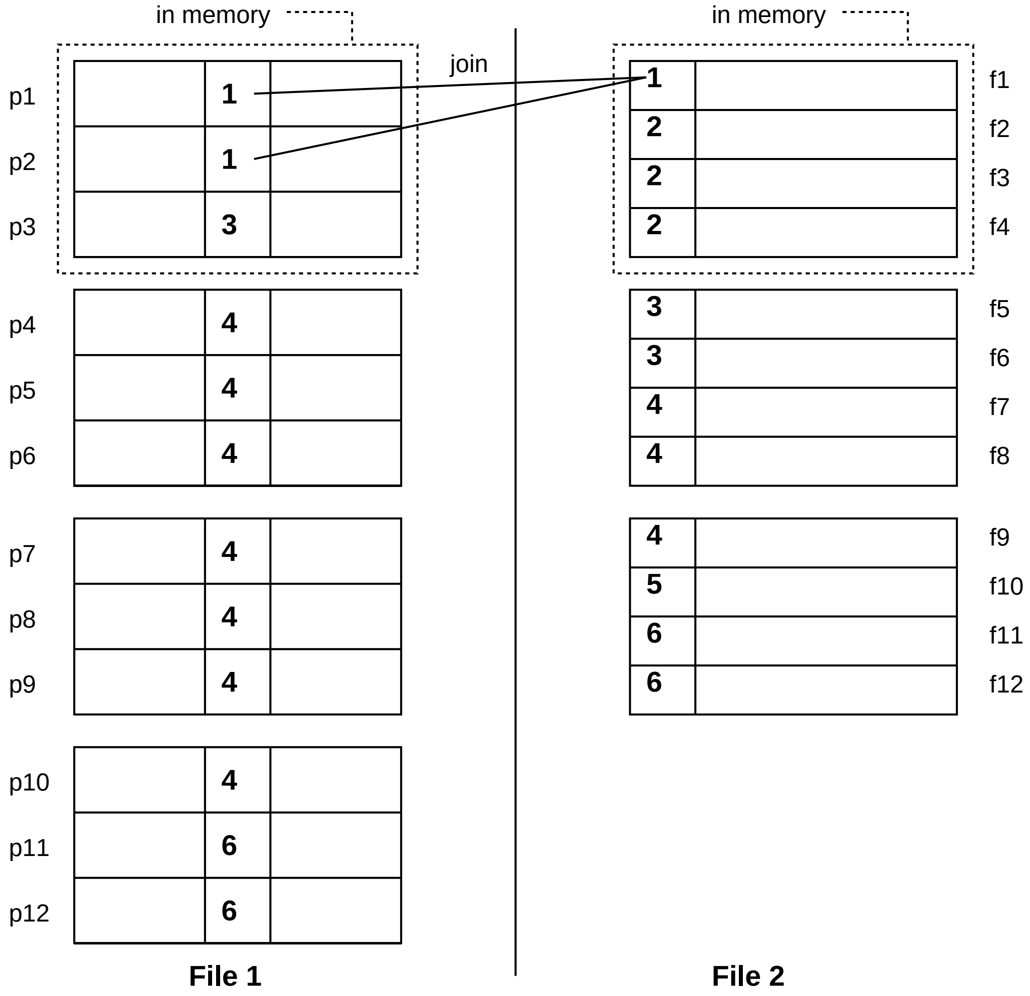
1. B+-tree index for PASSENGER:
   - ⇒ 5 blocks per access
   - ⇒ 50000 × 5 blocks of PASSENGER
   - ⇒ 251,000 blocks overall
   - ⇒ 1.39 hours.
2. Hash index for PASSENGER:
   - ⇒ about 2 blocks per access
   - ⇒ 101,000 blocks overall
   - ⇒ 33.7 minutes.

In general if α_P blocks are required by the index, then the total cost is

$$n_F + r_F * \alpha_P \text{ blocks.}$$

- Sort-merge Join
  - Key idea:
    - ∗ Sort PASSENGER.
    - ∗ Sort FLIGHT.
    - ∗ Set up sorted files as if they were going to be merged.
    - ∗ Process a merge, but only produce as output joined tuples.
  - How to merge?
    - ∗ Read first block of PASSENGER.
    - ∗ Read first block of FLIGHT.
    - ∗ Match all the tuples that can be matched.
    - ∗ Suppose last tuple of PASSENGER block has FLT_ID=6.
    - ∗ Suppose last tuple of FLIGHT block has FLT_ID=13.
    - ∗ Are there any other tuples in FLIGHT that need to be matched with FLT_ID=6?
      ⇒ No!
      ⇒ Don't need to look further in FLIGHT.
  - Example of sort-merge join:

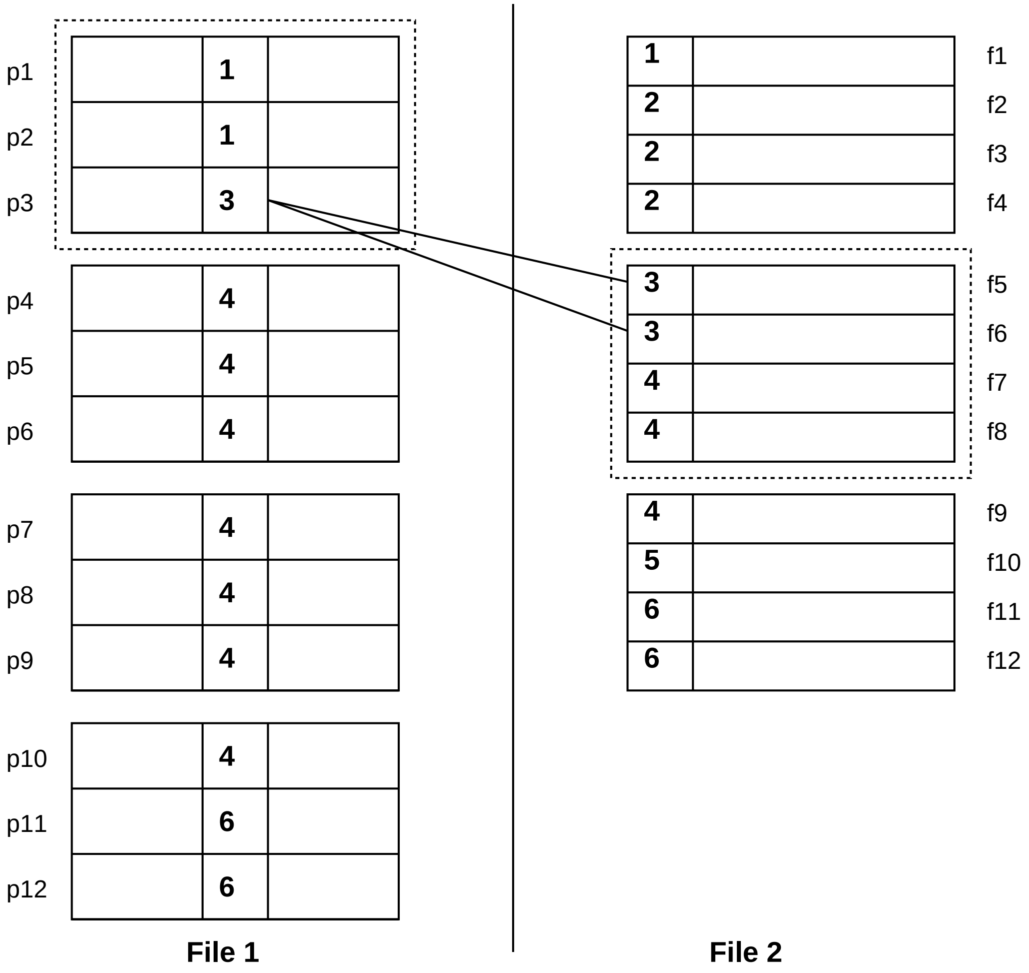
    Recall the relations: PASSENGER (NAME, SSN, FLT_ID, MILES) and FLIGHT (FLT_ID, FLT_NO, START_APT, END_APT).



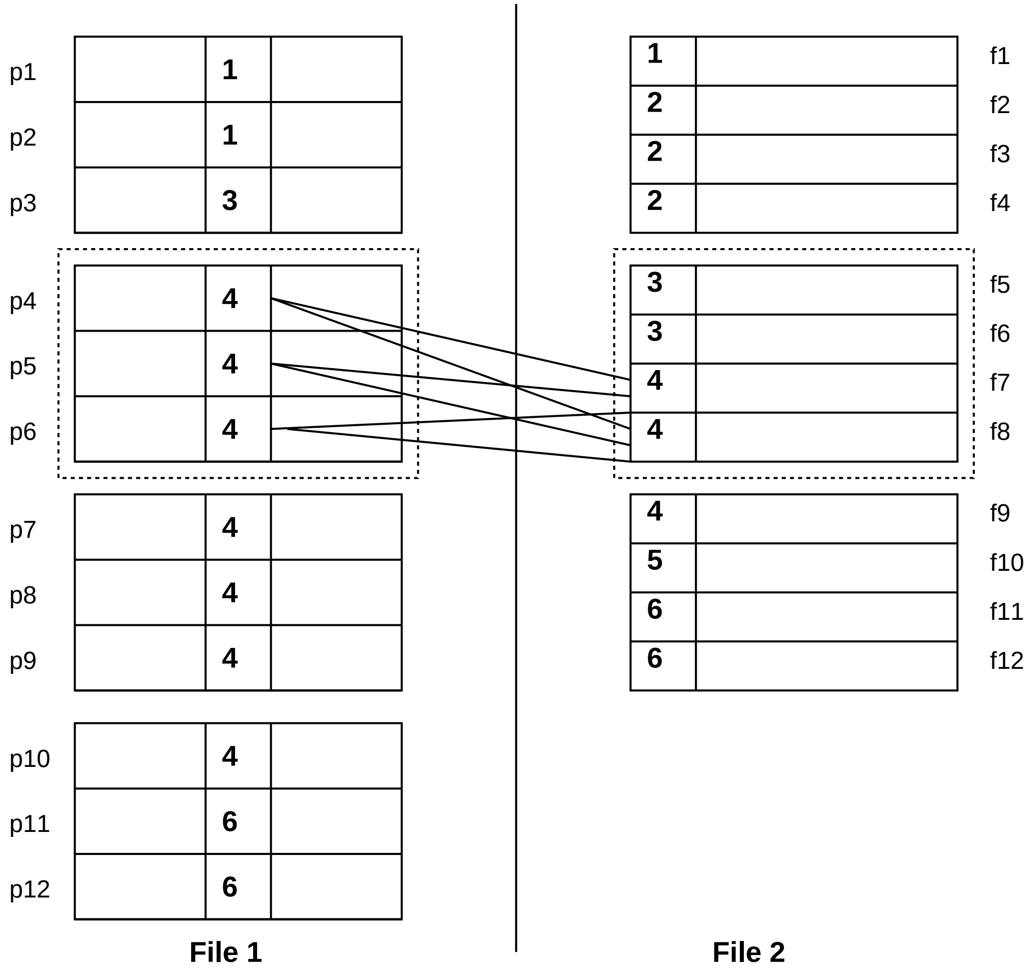
Let $p1, p2, ...$ denote the PASSENGER tuples and $f1, f2, ...$ denote the FLIGHT tuples.

1. First, we output `<p1,f1>, <p2,f1>`.

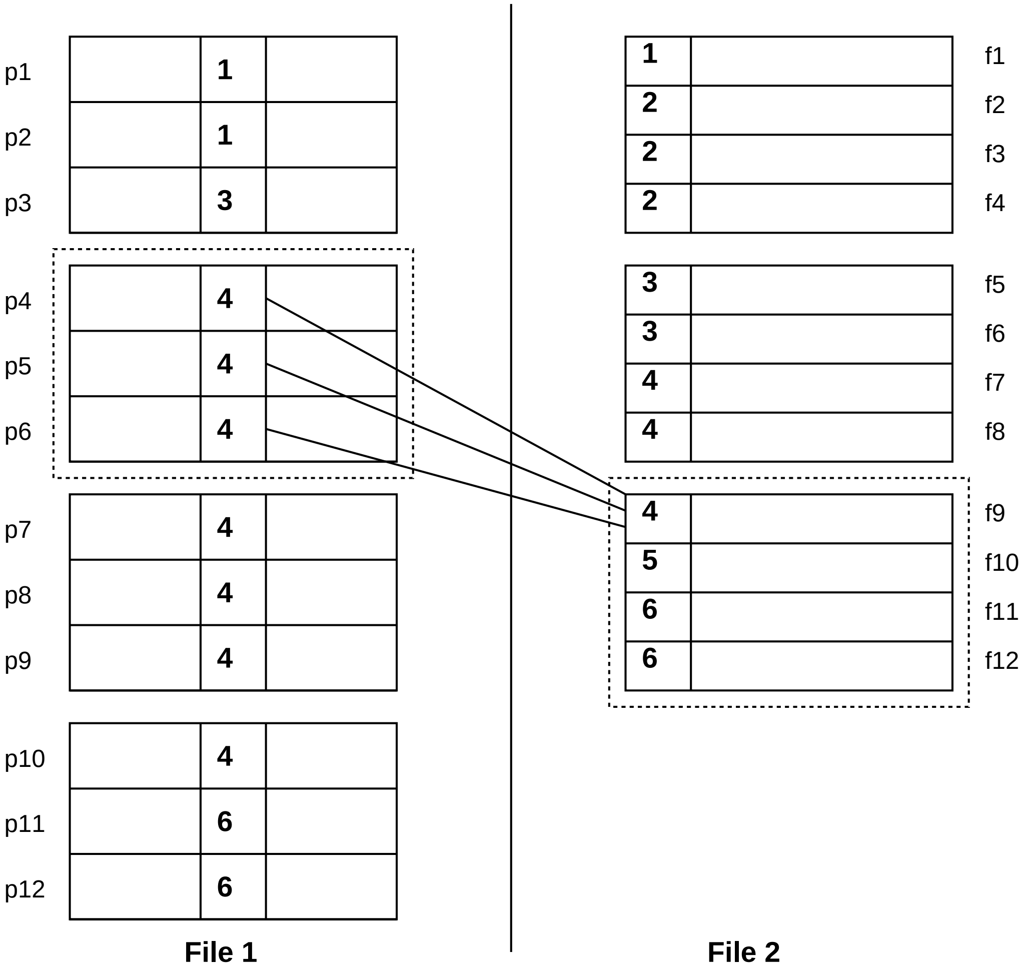
2. Then, read the second FLIGHT block and output `<p3,f5>, <p3,f6>`:



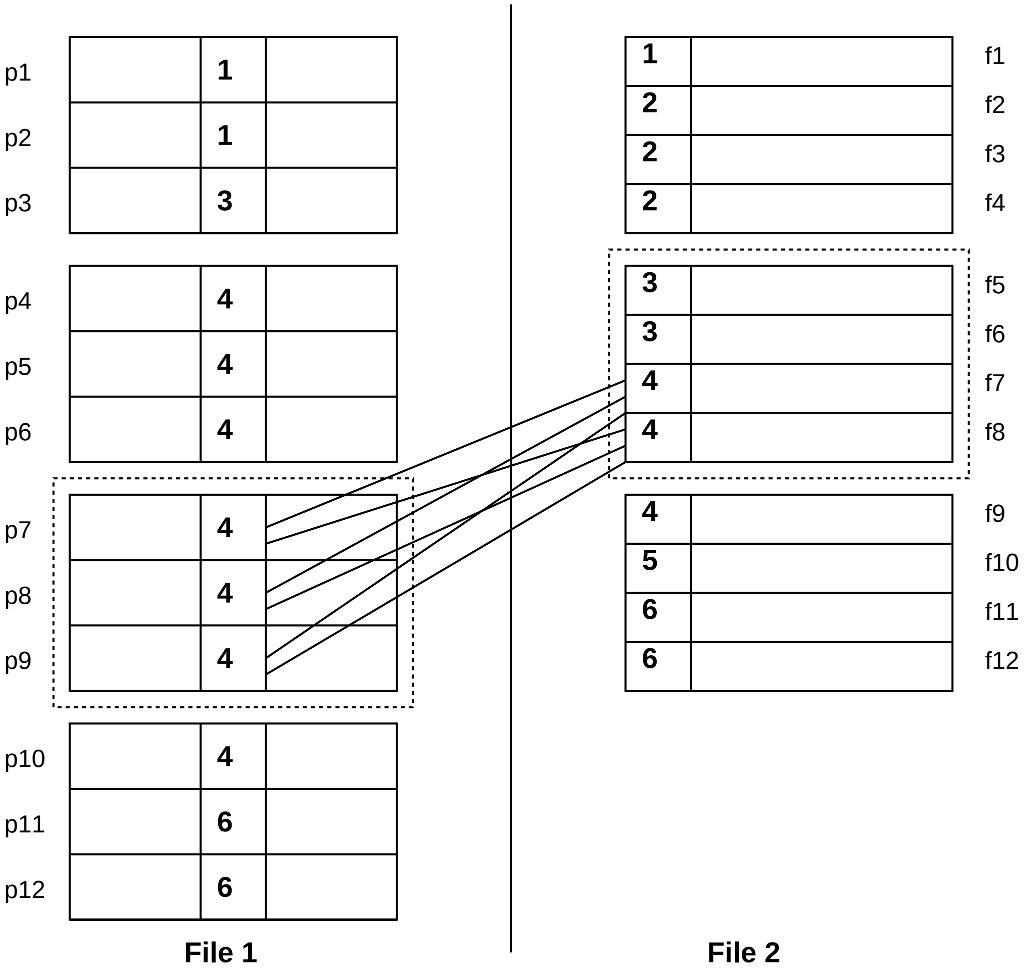
3. Read the second PASSENGER block and output `<p4,f7>, <p4,f8>, <p5,f7>, <p5,f8>, <p6,f7>, <p6,f8>`.



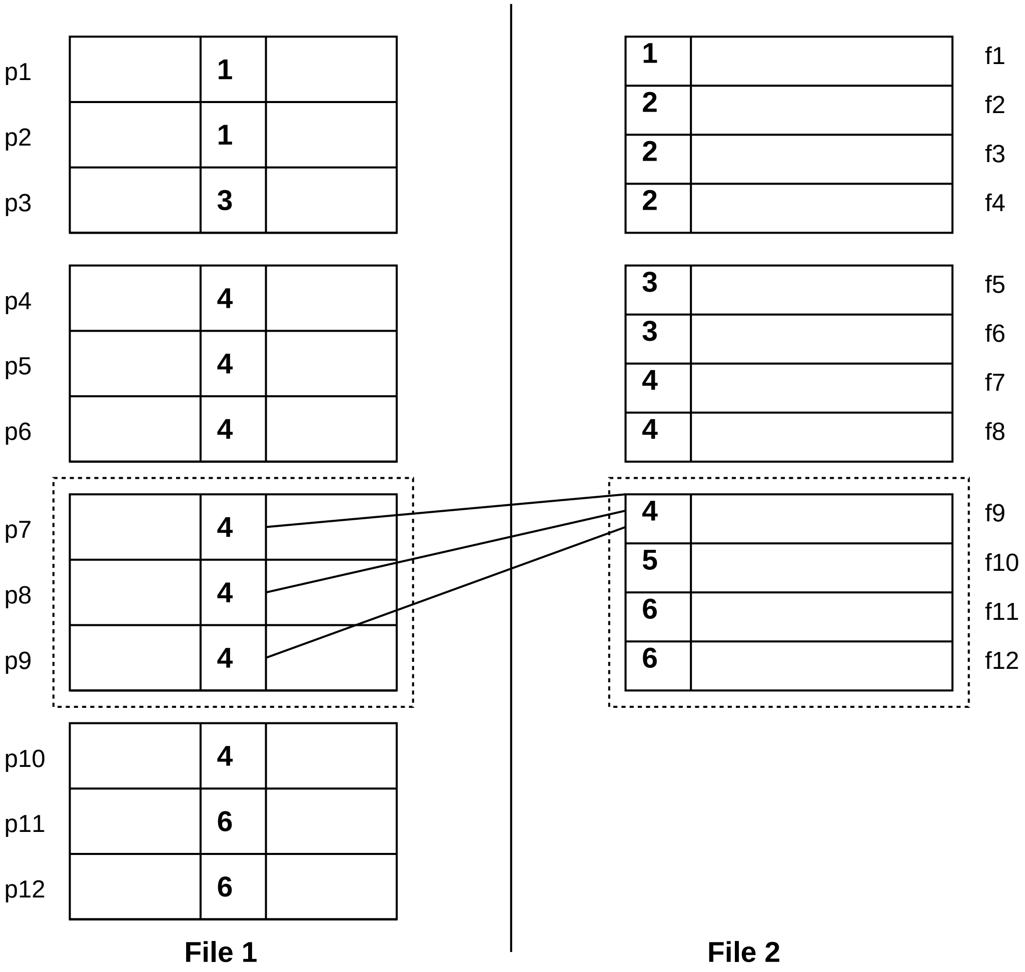
4. Read next FLIGHT block and
output `<p4,f9>`, `<p5,f9>`, `<p6,f9>`.



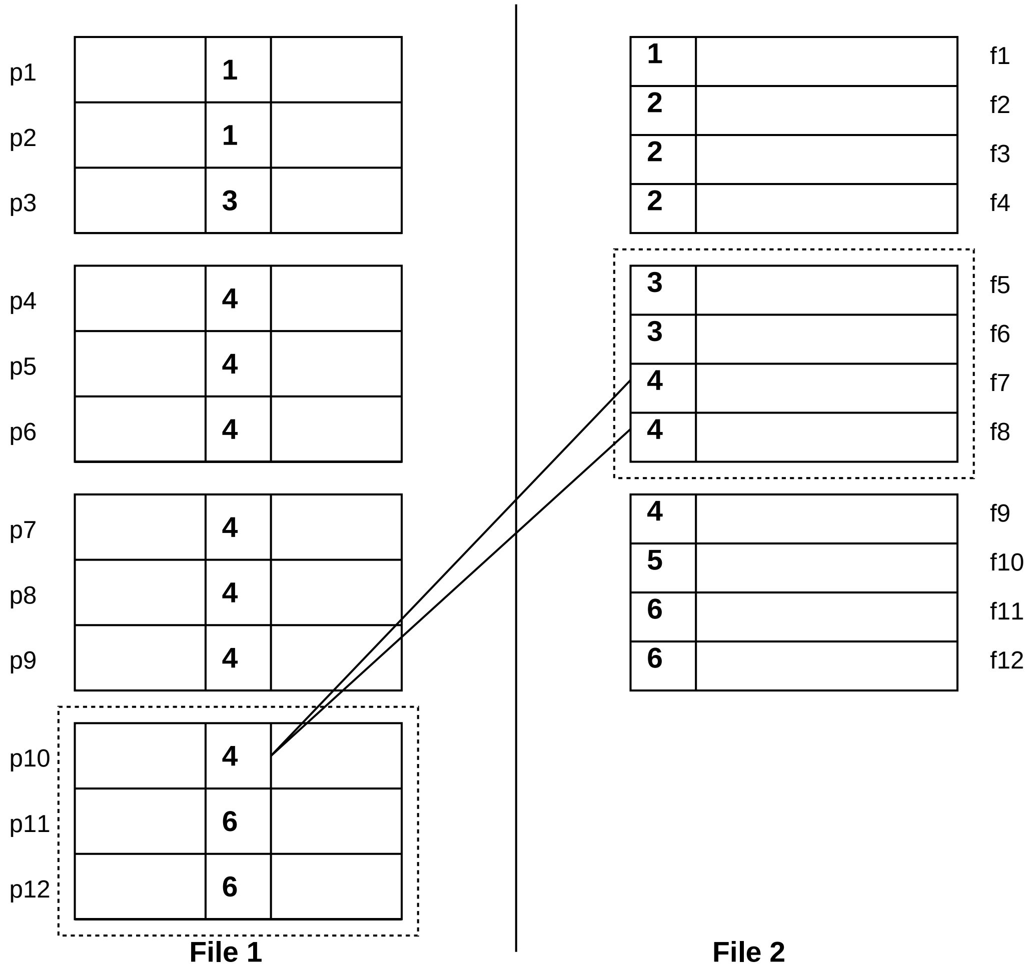
5. Read next PASSENGER block
   ⇒ the item ‘4’ continues
   ⇒ must go back to start of ‘4’ in FLIGHT
   ⇒ a reversal!
   Output `<p7,f7>, <p7,f8>, <p8,f7>, <p8,f8>, <p9,f7>, <p9,f8>`.



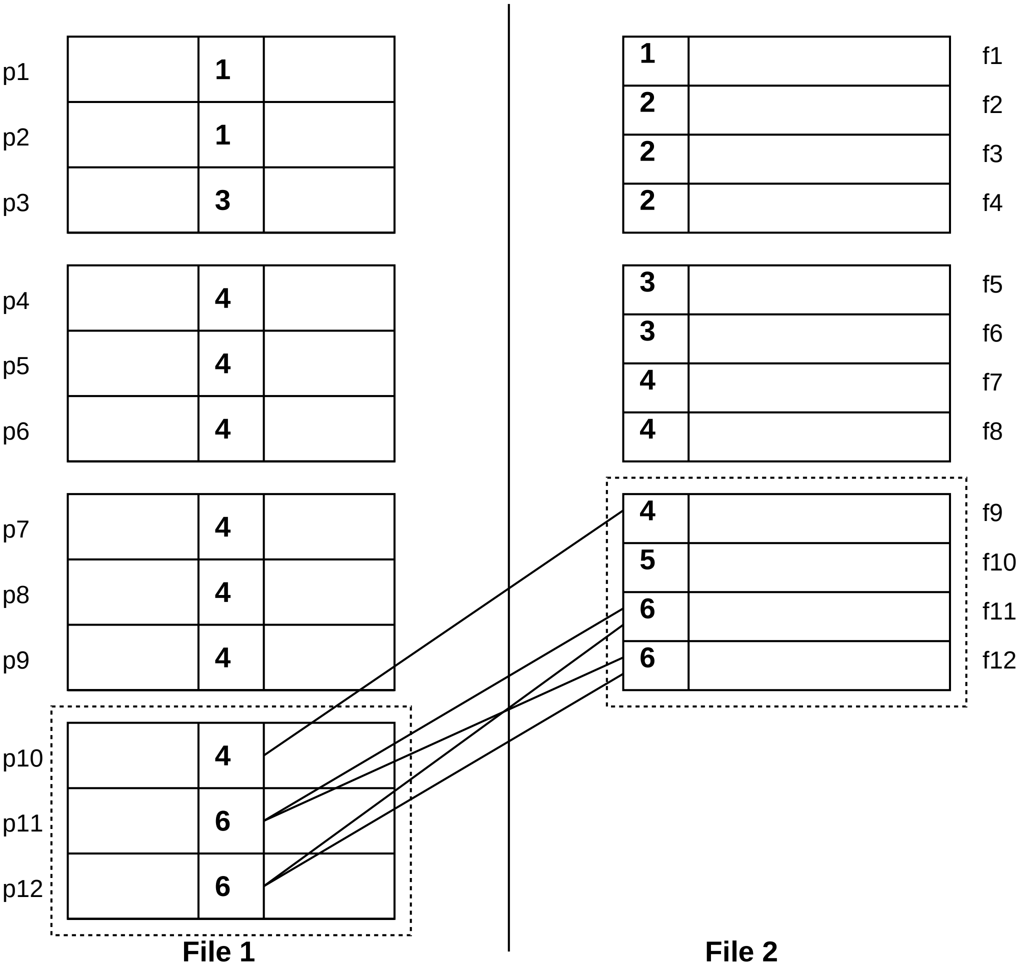
6. Read next FLIGHT block and output <p7,f9>, <p8,f9>, <p9,f9>.



7. Read next PASSENGER block
   ⇒ '4' continues
   ⇒ must go back to first '4' in FLIGHT
   ⇒ another reversal!
   Output `<p10,f7>`, `<p10,f8>`.



8. Read next FLIGHT block and
output `<p10,f9>`, `<p11,f11>`, `<p11,f12>`, `<p12,f11>`, `<p12,f12>`.



- Cost:
  - If there are enough buffers
    ⇒ reversal not needed.
  - Typically, reversal does not occur often
    ⇒ only one scan through each sorted file.
  - Time needed for sorting: $2n_P \log_M n_P + 2n_F \log_M n_F$.
  - Time needed for merging: $n_P + n_F$.
    ⇒ Total: $2n_P \log_M n_P + 2n_F \log_M n_F + n_P + n_F$.
  - Suppose $M = 100$ in our example:
    ⇒ $2 \times 5000 \log_{100} 5000 + 2 \times 1000 \log_{100} 1000 + 5000 + 1000$ blocks
    ⇒ 30,000 blocks
    ⇒ 10 minutes (best so far).
- Note:
  - Some runs of a particular FLT_ID could span several blocks
    ⇒ hopefully all fit into memory simultaneously.
  - If 'runs' of a particular value occur
    ⇒ try to reserve memory for these runs.
  - It is possible to speed up the process:
    1. Sort both files simultaneously.
    2. During each merge of the sort process, perform joins.
  - The sort-merge join is very general
    ⇒ it can be applied to any join condition.
  - A join on multiple attributes can be handled by sorting on the combination.

- Hash joins:
  - Key ideas:
    - Use a single hash function on the join attributes.
    - Hash the first file into a hash file using the function.
    - Hash the second file into the *same* file using the same hash function.
    - Records that are candidates for joining must lie in the same bucket.
    - Process the hash file bucket by bucket.
    - Read in a bucket and search for matches.
  - Some details:
    - If we use too many buckets, we will be doing a lot of I/O.

      ⇒ worst-case: 2 I/O's per tuple inserted

      ⇒ inserting 100,000 tuples means 100,000 block accesses!
      (at least one block access per *tuple* inserted).

      ⇒ Optimal number of buckets is M (memory size).
    - Assuming uniform distribution across buckets

      ⇒ each bucket has $\frac{n_P+n_F}{M}$ blocks.
    - To process a bucket, the bucket has to fit into memory.

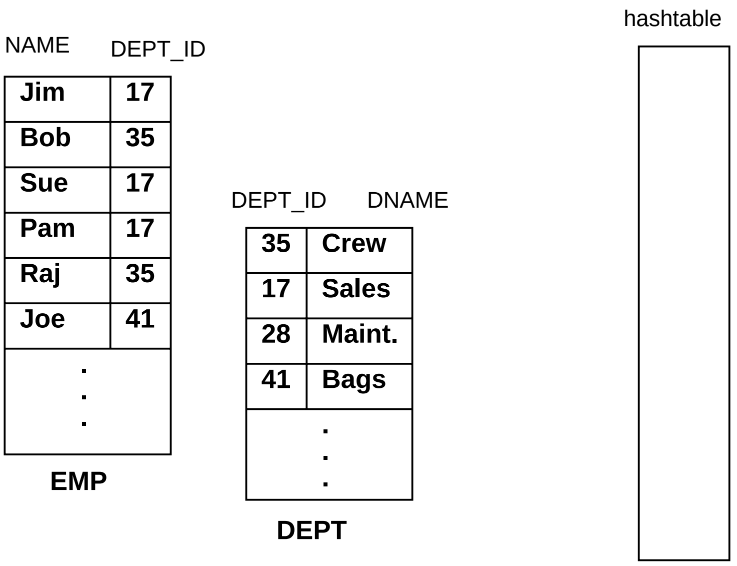
      ⇒ better to have $\frac{n_P+n_F}{M} \leq M$

      ⇒ $M \geq \sqrt{n_P + n_F}$ ideally.
    - It is not always possible to guarantee this condition

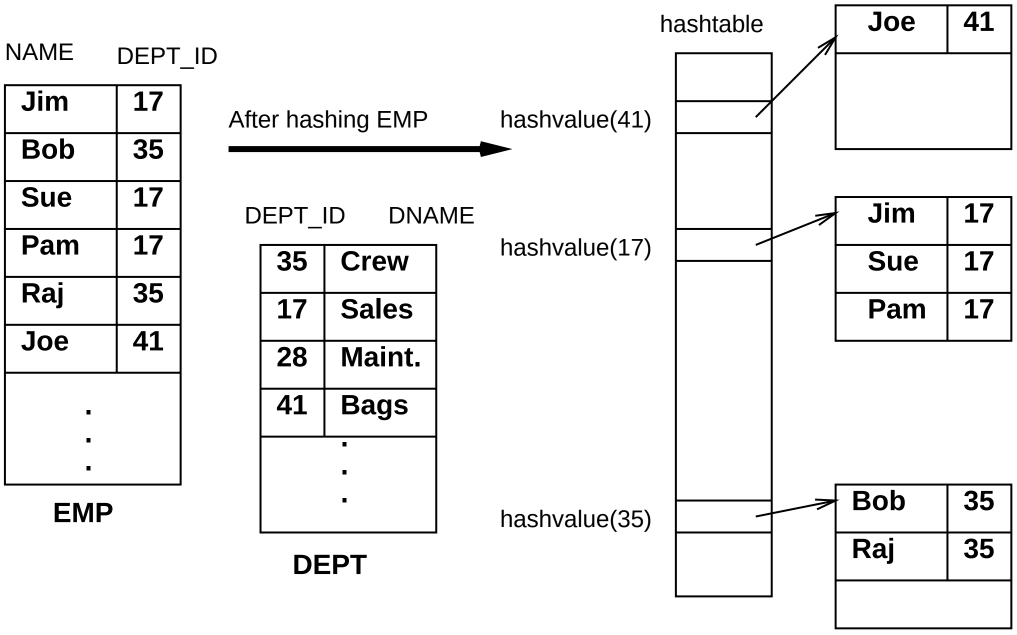
      ⇒ need a way to process a large bucket

      ⇒ use hashing again!
  - An example: consider a join of EMP and DEPT on the attribute DEPT_ID.

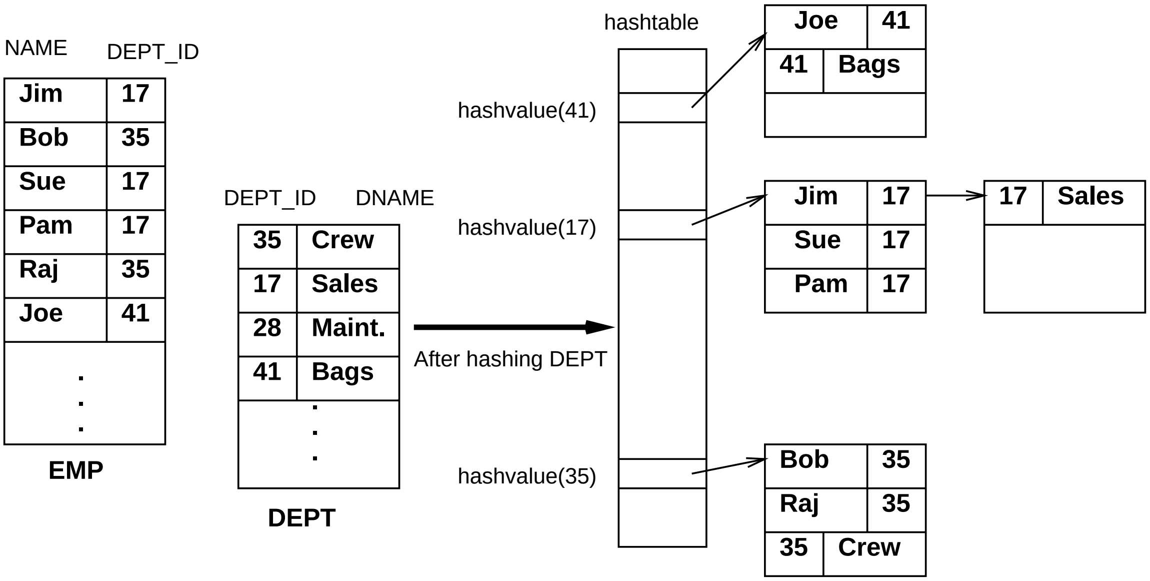
    Initially, the hashtable is empty:



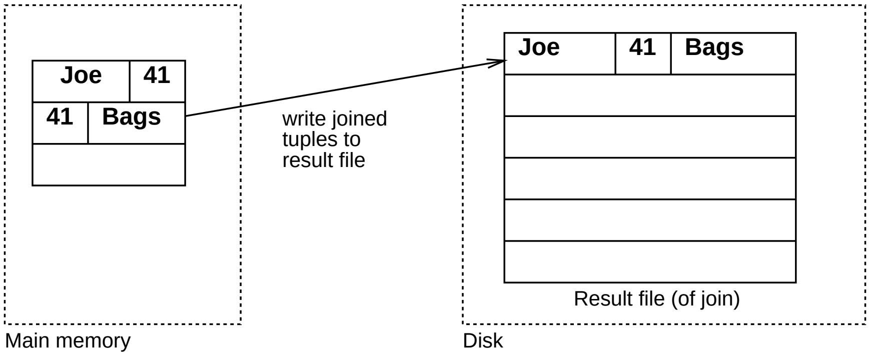
First, the file EMP is hashed (on DEPT_ID):



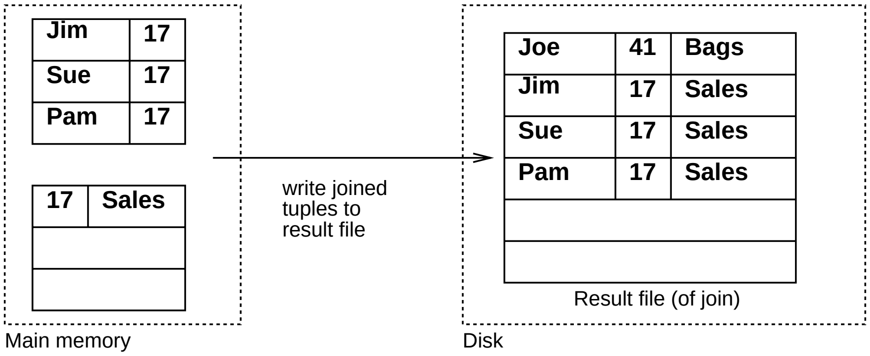
Then, the file DEPT is hashed into the same hash file:



Now, the buckets are processed one by one, e.g, bucket for ‘41’:



Process bucket for ‘17’:



And so on until all buckets have been processed.

- To process a large bucket:
  - ∗ Read in tuples from bucket and hash them to a temporary file.
  - ∗ Use a *different* hashing function than before.
  - ∗ Why? The same hashing function as before will only duplicate the bucket.
  - ∗ Again, process the new 'sub-buckets' one by one.
  - ∗ Hopefully the second hashing operation will create smaller buckets that fit into memory.
  - ∗ Worst-case, the procedure may have to be repeated.

- Memory Management in a DBMS:
  - The above discussion shows us why it's better for a DBMS to manage memory by itself.
  - In a block-nested join, once a block for the outer relation has been processed, it is not needed again
    ⇒ better to throw it out immediately and use the memory for the inner relation.
  - In other methods, we have explicitly assumed fixed buffer sizes (M)
    ⇒ DBMS memory management.

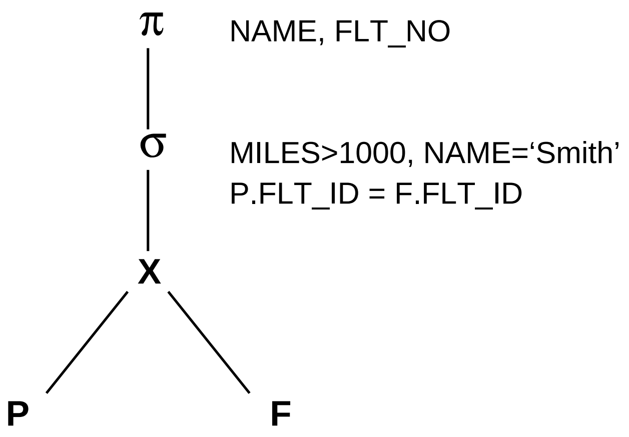
5.5 Query Optimization: Overview

- Key functions in query optimization:
  - Enumerate several query plans.
  - Evaluate the cost of each plan.
  - Select the best such plan.
- Enumerating plans:
  - A query plan includes:
    - A query tree.
    - Methods assigned to implement operators (e.g., Sort-merge for a given join).
    - An order of execution whenever different orders are possible.
  - Plans are enumerated by considering alternative ways of handling a query.
  - In general, enumeration is combinatorially explosive
    ⇒ heuristics must be considered.
  - It helps to use heuristic rules for specific operators, e.g., *push select's past joins*.
- Evaluating the cost of a particular plan:
  - Use a metric such as the *number of block accesses* to evaluate I/O cost.
  - Use techniques for cost estimation of relational operators (described earlier).
  - Put together the total cost for a plan.

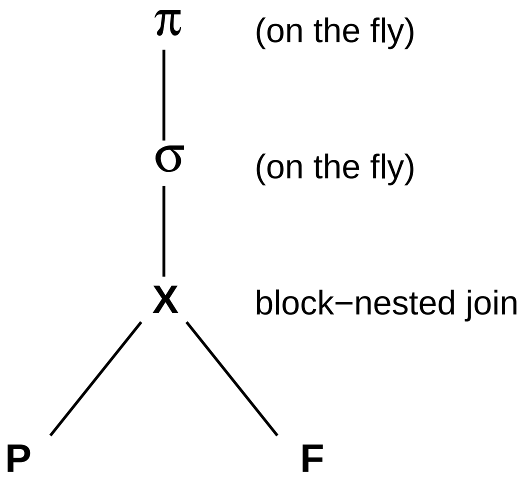
- Consider the following example:

```
select  NAME, FLT_NO
from    PASSENGER P, FLIGHT F
where   P.FLT_ID=F.FLT_ID
        and MILES>1000
        and NAME='Smith'
```

The query tree produced by the parser typically looks like:



- Several options are possible with the above query:
  1. Operator methods:
     - Use block-nested join with FLIGHT (F) as outer relation.
     - Compute both σ and Π on the fly (as results are produced by the join)

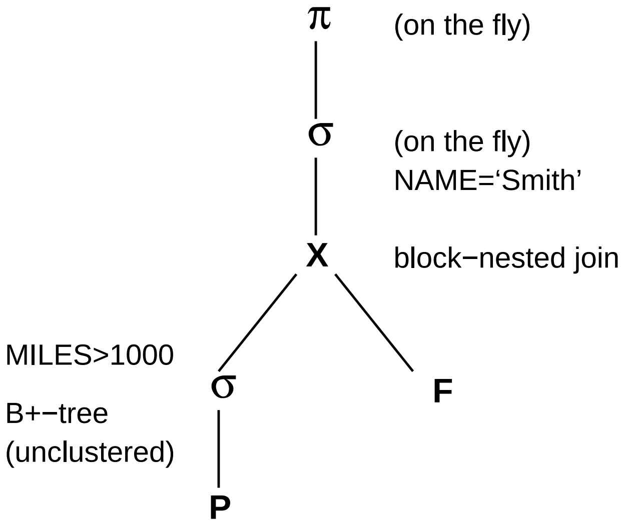


Cost estimation:

- Recall cost of block-nested join:
  $\Rightarrow n_F + \lceil \frac{n_F}{M-1} \rceil n_P$ blocks
  $\Rightarrow 1000 + \lceil \frac{5000}{99} \rceil 1000$ blocks
  $\Rightarrow$ 51506 blocks.
- No cost for computing σ or Π.
- Total: 51506 blocks
  $\Rightarrow 51506 \times 20$ ms
  $\Rightarrow$ 17.1 minutes.

2. Methods:

   - Push the "MILES>1000" condition past the join to PASSENGER.
   - Use B+-tree (unclustered, say) for select.
   - Use block-nested join with FLIGHT as outer relation.
   - Compute remaining operators on the fly.



Cost:

- 2000 tuples (of PASSENGER) satisfy MILES>1000
  ⇒ (worst-case) 2000 data blocks, 4+20 tree blocks (tree and leaf-level)
  ⇒ 2024 blocks accessed.
- 2000 tuples (of PASSENGER) satisfy MILES>1000
  ⇒ 100 blocks produced as a result
  ⇒ these 100 blocks will be joined with FLIGHT's 1000 blocks.
- Block nested join with FLIGHT:
  ⇒ $1000 + \lceil \frac{100}{99} \rceil 1000$ blocks
  ⇒ 3000 blocks.
- No cost for remaining select's and project.

- Total: 5024 blocks
  $\Rightarrow$ 5024 × 20 ms
  $\Rightarrow$ 1.67 minutes.
- Note: we could use FLIGHT as inner relation
  $\Rightarrow 100 + \lceil \frac{100}{99} \rceil 1000$ blocks
  $\Rightarrow$ 2100 blocks
  $\Rightarrow$ 4124 blocks overall
  $\Rightarrow$ 1.37 minutes.
- NOTE: an important point:
  * In the above analysis, we used the fact that 2000 tuples satisfy "MILES>1000".
  * How did we know this?
  * In practice, an *estimate* of the number of tuples is required.
  * Example: suppose the MILES field has 50 unique values
    $\Rightarrow$ 100,000 tuples / 50 = 2000 tuples have one value (uniform distribution).

    $\Rightarrow$ forms a very crude estimate.

    Better: use min(MILES) and max(MILES).

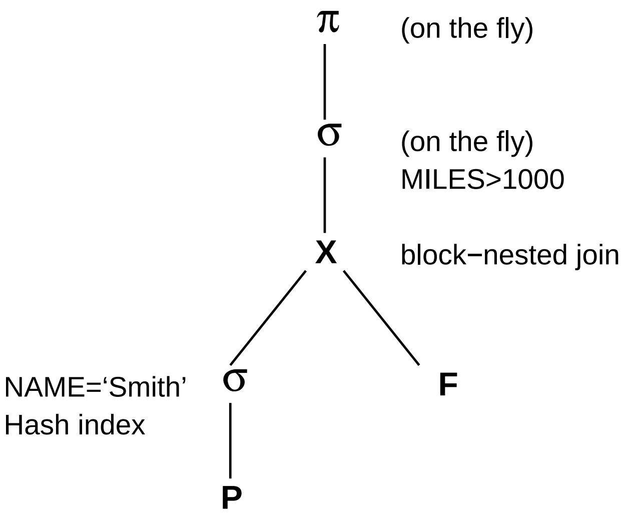
    E.g., suppose min(MILES)=0, max(MILES)=1020
    $\Rightarrow$ (1020-1000)/1020 fraction of tuples will have MILES>1000
    $\Rightarrow \approx 0.0196 \times 100{,}000$ tuples have MILES>1000
    $\Rightarrow \approx$ 1960 tuples.

3. Methods:

   – Push the other select down (NAME=‘Smith’).

   – Assume hash index exists on NAME.

   – Block nested join with FLIGHT as inner relation.



Cost:

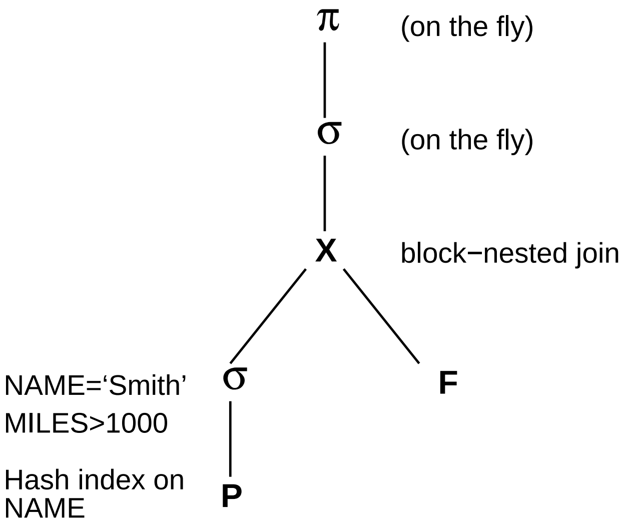
– 45 tuples satisfy NAME=‘Smith’
  ⇒ 3 blocks
  ⇒ 4 blocks overall (including directory block).

– Block-nested join with FLIGHT as inner relation:
  ⇒ $4 + \lceil \frac{4}{99} \rceil 1000$ blocks
  ⇒ 1004 blocks.

– Total: 1008 blocks
  ⇒ 20.2 seconds.

4. Methods:

   - Push both select's down.
   - Assume hash index exists on NAME.
   - Check MILES>1000 on the fly.
   - Block nested join with FLIGHT as inner relation.

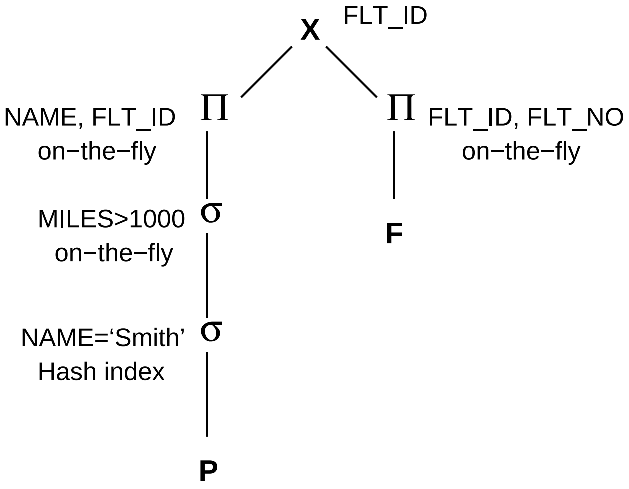


   Cost:

   - 45 tuples satisfy NAME='Smith'
     $\Rightarrow$ 3 blocks
     $\Rightarrow$ 4 blocks overall (including directory block).
   - Assume only 1 tuple satisfies both conditions
     $\Rightarrow$ 1 block in result.
   - Block-nested join with FLIGHT as inner relation:
     $\Rightarrow 1 + \lceil \frac{1}{99} \rceil 1000$ blocks
     $\Rightarrow$ 1001 blocks.
   - Total: 1005 blocks
     $\Rightarrow$ 20.1 seconds.

5. Methods:

   - Push project past joins.
   - However, FLT_ID is needed
     ⇒ cannot project it out.
   - Use sorting to remove duplicates.
   - Use Hash index for "NAME='Smith" condition.
   - Check "MILES>1000" on the fly.



Cost:

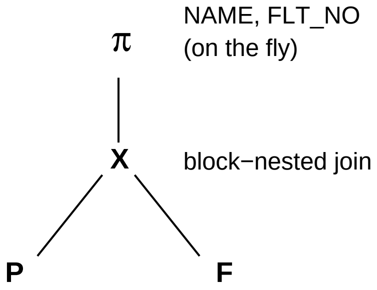
   - Consider the cost to sort FLIGHT:
     ⇒ $2n_F\lceil\log_M n_F\rceil$ blocks
     ⇒ $2 \times 1000\lceil\log_{100} 1000\rceil$ blocks
     ⇒ 4000 blocks
     ⇒ already more expensive than some previous methods!
   - Lesson: does not always help to push projects.

- Sometimes, it does help to push project:

```
select  NAME, FLT_NO
from    PASSENGER P, FLIGHT F
where   P.FLT_ID=F.FLT_ID
```

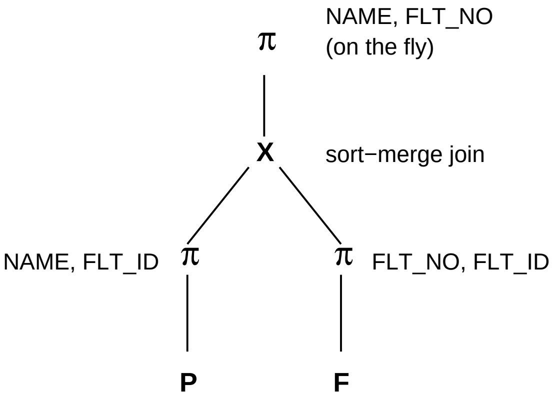
In this case, consider the following two plans:

1. Methods:
   - Use a block-nested join with FLIGHT as outer relation.
   - Compute projection on the fly.



   Cost:
   - Total: 51506 blocks (see prior analysis)
     ⇒ 17.1 minutes

2. Methods:
   - Push projects down
     ⇒ must retain join attributes (FLT_ID).
   - Project NAME, FLT_ID from PASSENGER.
   - Project FLT_NO, FLT_ID from FLIGHT.
   - Use sorting to remove duplicates.
   - Use sort-merge join (since results of project are sorted).



Cost:

- To project attributes of PASSENGER
  ⇒ scan PASSENGER
  ⇒ 5000 blocks.
- To project attributes of FLIGHT
  ⇒ scan FLIGHT
  ⇒ 1000 blocks.
- Assume (NAME,FLT_ID) take up 25 bytes.
  ⇒ 40 tuples per block
  ⇒ 2500 blocks.
- Assume (FLT_ID,FLT_NO) take up 10 bytes.
  ⇒ 100 tuples per block
  ⇒ 500 blocks.
- To remove duplicates in PASSENGER result:
  ⇒ sort result (and remove duplicates on the fly)
  ⇒ $2 \times 2500 \log_{100} 2500$ blocks
  ⇒ 10000 blocks.
- Total for projection of PASSENGER:
  ⇒ $10000 + 5000 = 15000$ blocks.
- To remove duplicates in FLIGHT result:
  ⇒ sort result (and remove duplicates on the fly)
  ⇒ $2 \times 500 \log_{100} 500$ blocks
  ⇒ 2000 blocks.

- Total for projection of FLIGHT:
  $\Rightarrow$ 2500 + 500 = 3000 blocks.
- Sort-merge join
  $\Rightarrow$ 2500 + 500 blocks
  $\Rightarrow$ 3000 blocks.
- Total: 15000 + 3000 + 3000
  $\Rightarrow$ 21,000 blocks
  $\Rightarrow$ 7 minutes.

5.6 Relation Statistics

- A DBMS usually keeps some statistical meta-data on each relation, for example:
  - *Cardinality* of each relation.
  - Number of blocks.
  - Number of distinct key values in each index (e.g., total number of entries in a B-tree).
  - Heights of tree indices.
  - Max and Min values of numeric attributes.
- The statistics are used to estimate quantities used in query optimization.
- Example:
  - Suppose max(MILES)=50,000 and min(MILES)=0.
  - Suppose total number of tuples is 100,000.
  - How many tuples satisfy MILES>10,000?
  - Use uniform distribution:

$$\frac{50{,}000 - 10{,}000}{50{,}000 - 0} \times 100,000 \ \ = 80,000 \text{ tuples.}$$

- Other estimates are harder:
  - For example: how many tuples are there in the join

    PASSENGER $*$ FLIGHT?
  - In this case, we know that FLT_ID is a key for FLIGHT
    ⇒ each tuple of PASSENGER joins with at most one tuple of FLIGHT
    ⇒ number of tuples in join = number of PASSENGER tuples.
  - Often, arbitrary “fudge factors” are used.

5.7 Manual Intervention

- A DB Administrator (DBA) can take steps to improve the performance of some “tough” queries:
  - Decide to create or remove indices.
  - Decide types of indices.
  - Use tools or system options to re-organize data on disk.
- A DBA needs to understand the system workload:
  - Frequently-used queries and how frequently they are used.
  - Updates and update frequencies.
  - User requirements (complaints).

  ⇒ identify the important queries (users?) and focus on them.
- It may be tempting to define an index on *every* attribute used in the important queries
  ⇒ indices make (read) access faster.

  However, indices also:
  - take up space (often as large as the relation itself)
    ⇒ drains virtual memory (code/data), clutters up disk (data)
  - are bad for insertions
    ⇒ compare insertion using a B-tree versus insertion into a heapfile.

  Fast updates are desirable in some applications, e.g., banking.
- General rules of thumb in index creation:
  - Create indices for attributes that occur in multiple queries.
  - Avoid creating indices for small relations.

- If some updates are important, be careful about creating indices for attributes involved in the update.
  Note: hash indices allow for fast insertion
- If a B+-tree index is defined on non-key attributes, consider using a clustered index.
- For equality selections, use a hash index provided range searches are not required.

Chapter 6

Database Schema Design

Course Notes on Database Systems

6.1 Database Schema Design: Introduction

- A *Database Administrator* (DBA) is responsible for designing the schema of a relational database
  ⇒ need to decide which attributes go into which relations.
- Other DBA responsibilities:
  - Analysis of user needs.
  - Performance.
  - Security and access (for users as well as dbase programmers).
- We will focus on schema design.
- Schema design usually proceeds in two phases:
  1. Use *informal guidelines* to create an initial design.
  2. Use *formal guidelines* to improve initial design.
- Consider an airline employee dbase:
  - Schema $S1$:

    EMP (NAME, SSN, FLT_ID, START_APT, END_APT, DEPTNO, DNAME, MGRSSN)

  - Schema $S2$:

    EMP (NAME, SSN, DEPTNO)
    CREW (SSN, FLT_ID)
    FLIGHT (FLT_ID, START_APT, END_APT)
    DEPT (DEPTNO, DNAME, MGRSSN)

  Which schema is better?
- Review of terminology:

- **superkey** – any group of attributes that can uniquely identify a tuple *in any instance.*

  e.g., for a particular instance (FNAME, LNAME) may appear to be a key – but it can't be guaranteed.
  ⇒ (FNAME, LNAME) is a poor choice for a key.
- **key** – a minimal superkey, e.g.,
  (NAME, SSN) is a superkey (but not a key)
  (SSN) is a key (and also a superkey)
- **primary key** – one key designated for general use.
- **foreign key** – a set of attributes in relation R that is the primary key for relation S.
- **Domain constraint** – proper typing of values.
- **Key constraint 1** – no two tuples can have identical keys
  If a tuple is found that violates the condition, then either
  - ∗ the tuple should be rejected or,
  - ∗ the key is not truly a key (i.e., a bad choice)
- **Key constraint 2** – no primary key value can be null.
- **Referential integrity constraint** – can't have a tuple whose foreign key values don't exist in the foreign relation *instance.*

  For example, consider

  EMP1 (NAME, SSN, FLT_ID, START_APT, END_APT, DEPTNO)
  DEPT (DEPTNO, DNAME, MGRSSN)

  e.g. <John, B, Smith, ... , 9> and DEPTNO=9 does not exist at present.

6.2 Informal Guidelines

1. **Try to make user interpretation easy**.

   For example, compare

   - Schema $S1$:

     EMP (NAME, SSN, FLT_ID, START_APT, END_APT, DEPTNO, DNAME, MGRSSN)

   - Schema $S2$:

     EMP (NAME, SSN, DEPTNO)

     CREW (SSN, FLT_ID)

     FLIGHT (FLT_ID, START_APT, END_APT)

     DEPT (DEPTNO, DNAME, MGRSSN)

   Perhaps $S1$ has too much information (to absorb) per tuple

2. **Try to reduce redundancy**.

   Suppose there are only a few departments
   ⇒ MGRSSN and DNAME are unnecessarily repeated too often in EMP.

NAME	...	DEPTNO	DNAME	MGRSSN
Abel	...	5	Crew	111-22-3334
Aitken	...	3	Ticketing	222-33-4445
Al Khwarizmi	...	5	Crew	111-22-3334
Archimedes	...	5	Crew	111-22-3334
⋮	⋮	⋮	⋮	⋮
Zeno	...	3	Ticketing	222-33-4445
Zuse	...	5	Crew	111-22-3334

   On the other hand, $S2$ repeats DEPTNO in DEPT
   ⇒ DEPTNO (integer) is smaller than DNAME (string) and MGRSSN (string)

3. **Try to avoid update anomalies**

   Consider the schemas:

   - Schema $S1$:

     EMP (NAME, SSN, FLT_ID, START_APT, END_APT, DEPTNO, DNAME, MGRSSN)

   - Schema $S2$:

     EMP (NAME, SSN, DEPTNO)
     CREW (SSN, FLT_ID)
     FLIGHT (FLT_ID, START_APT, END_APT)
     DEPT (DEPTNO, DNAME, MGRSSN)

   Both have same attibutes. Unfortunately, $S1$ can create problems called *update anomalies.*

   - **Insertion anomalies**

     (a) Consider inserting "John Smith works in Dept. 5", i.e.,
     <John, Smith, 123456789, ... , 5, <dname>, <mgrssn> >.

     Every time a tuple of this sort is entered, we have to check that DNAME is correct
     ⇒ we have to scan the whole relation (worst-case)

     (b) Consider creating a new department: DEPTNO=9, DNAME='Security' (with no employees yet)
     Only one way to insert this info
     ⇒ create NULL values for employee info
     ⇒ but that means a NULL primary key value (SSN)!

   - **Deletion anomalies**
     If we delete the last employee in the 'Crew' department, e.g.

     ```
     <John, Smith, 123456789, ... , 5, 'Crew', ... >
     ```

     then we will lose the information
     "Department 5 is the Crew department".

- **Modification anomalies**
  Suppose we change the manager of department 5
  ⇒ we have to change MGRSSN for all department 5 employees
  ⇒ full scan of database

Thus, schema $S1$ has many problems. On the other hand:

- $S1$ – has 1 relation.
- $S2$ – has 4 relations.
- For many queries, we will need more joins using $S2$.
- SQL code with $S2$ will be more complicated because of the extra joins
  (One solution: use $S2$ but create views based on needed joins)

4. **Try to avoid too many NULL values**.
   - This may occur in 'fat' relations (with too many attributes).
   - Space is wasted.
   - Problems occur when using aggregate functions like **count** or **sum**.
   - NULLs can have different intentions:
     (a) The attribute does not apply.
     (b) Value is unknown, and will remain unknown.
     (c) Value is unknown at present.
5. **Beware of the Spurious Tuple Problem**.
   Consider the following two schemas:
   - Schema $S1$:
     EMP (NAME, HOME_APT, SSN, FLT_ID)
   - Schema $S2$:
     EMPNEW (NAME, SSN, FLT_ID)
     HOMEBASE (NAME, HOME_APT)

   First, let us see how a relation in $S1$ can be converted to relations in $S2$, e.g., consider this data:

EMP	NAME	HOME_APT	SSN	FLT_ID
	Smith	National	111-22-3333	18
	Smith	JFK	222-33-4444	48
	Jones	La Guardia	333-44-5555	119

To create EMPNEW:

$$\text{EMPNEW} := \Pi_{\text{NAME, SSN, FLT\_ID}} \ (\text{EMP})$$

Thus,

EMPNEW	NAME	SSN	FLT_ID
	Smith	111-22-3333	18
	Smith	222-33-4444	48
	Jones	333-44-5555	119

Similarly, to create HOME_BASE:

$$\text{HOMEBASE} := \Pi_{\text{NAME, HOME\_APT}} \ (\text{EMP})$$

In this case,

HOMEBASE	NAME	HOME_APT
	Smith	National
	Smith	JFK
	Jones	La Guardia

Now, suppose we are using $S2$ and we want to recreate $S1$ (say, as a *view*):

$$\text{EMP} := \text{EMPNEW} * \text{HOME\_BASE}.$$

We get the following join:

EMP	NAME	HOME_APT	SSN	FLT_ID	
	Smith	National	111-22-3333	18	
	Smith	JFK	111-22-3333	18	*
	Smith	JFK	222-33-4444	48	
	Smith	National	222-33-4444	48	*
	Jones	La Guardia	333-44-5555	119	

Here, the $*$-tuples are *spurious*!

What happened? Since NAME is not a key, a careless join produced wrong results.

Summary of problems:

- Insertion, deletion and modification anomalies.
- Too many NULLs.
- Spurious tuples.

$\Rightarrow$ We need a theory of schema design
$\Rightarrow$ *functional dependencies* and *normalization*

6.3 Functional Dependencies

- First, some convenient notions (and notation):
  - Suppose our relational database has attributes $A_1, \ldots, A_n$.
  - Let R denote the schema $R = (A_1, \ldots, A_n)$.
  - Typically, of course, we will have several relations, e.g., EMP(A_1, A_3, A_9), DEPT(A_9, A_7, A_8) ... etc.
  - However, we will pretend there is a relation with *all* the attributes, i.e., with schema $R = (A_1, A_2, \ldots, A_n)$.
- **Definition**. A **functional dependency (FD)** between two sets of attributes X and Y, denoted by $X \to Y$, specifies a certain relationship or connection between X and Y. Specifically, it says:
  if t_1 and t_2 are any two tuples in any instance of R such that

  $$t_1[X] = t_2[X]$$

  then

  $$t_1[Y] = t_2[Y]$$

  Intuitively, $X \to Y$ means: if you know the X-values of a tuple, then that uniquely determines the Y-values.

  Note: X and Y can be single attributes or *groups* of attributes.
- Example:
  Consider the relational database schema:

  EMP (NAME, SSN, FLT_ID, START_APT, END_APT)

  Suppose

  $$\begin{aligned} X &= \{\text{SSN}\} \\ Y &= \{\text{NAME}\} \end{aligned}$$

Then $X \to Y$, i.e., SSN uniquely determines NAME

From the definition, if we are given two tuples t_1 and t_2, e.g.,

t_1 = <111-22-3333,...,Smith,...>
t_2 = <111-22-3333,...,Smith,...>

where

$$t_1[X] = t_2[X] \quad \text{(i.e., } t_1[\text{SSN}] = t_2[\text{SSN}]\text{)}$$

then it must be true that

$$t_1[Y] = t_2[Y] \quad \text{(i.e., } t_1[\text{NAME}] = t_2[\text{NAME}]\text{).}$$

Thus, we can't have two tuples with the same SSN and different NAME's.

Similarly, the functional dependency

$$\{\text{FLT\_ID}\} \to \{\text{START\_APT, END\_APT}\}$$

is a reasonable assumption or a reasonable constraint to declare.

- Functional dependencies are specified to capture dependencies for *all* instances.

  It may just happen that employee names (NAME) are all different for a particular instance.

  ⇒ we may be led to believe that $\{\text{NAME}\} \to \{\text{FLT\_ID}\}$.

  But, later on, another 'Smith' might join the airline
  ⇒ this would be a poor choice of a FD.

  A FD is a property of the meaning *of attributes*

  ⇒ it should hold for all possible instances.

- FD's for specific relations.

  Although we have defined FD's on the universe of attributes, we will often discuss FD's within particular relations.

  For example, the database schema might be:

  (NAME, SSN, FLT_ID, START_APT, END_APT, DEPTNO, DNAME, MGRSSN).

  We might have the relation

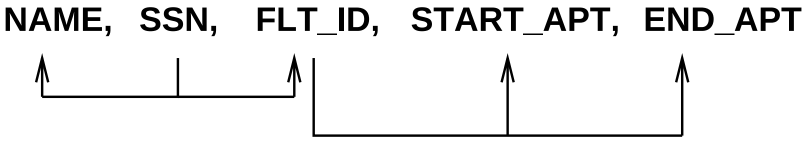
  FLIGHT (FLT_ID, START_APT, END_APT)

  Here, we can identify the FD

  $$\{\text{FLT\_ID}\} \rightarrow \{\text{START\_APT, END\_APT}\}$$

  in FLIGHT.

- Sometimes a diagram is used to show FD's, e.g.,



  Here the FD's shown are:

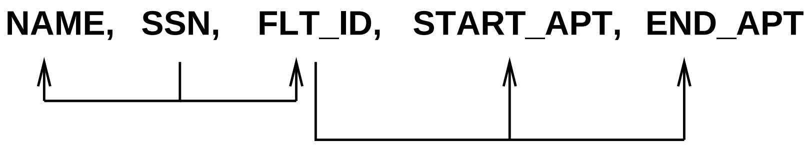
  $$\begin{array}{lcl} \{\text{SSN}\} & \rightarrow & \{\text{NAME, FLT\_ID}\} \\ \{\text{FLT\_ID}\} & \rightarrow & \{\text{START\_APT, END\_APT}\} \end{array}$$

- Summary:
  - A FD is defined between sets of attributes.
  - The FD $X \rightarrow Y$ says "The X-attributes completely determine the Y-attributes".

6.4 Sets of Functional Dependencies

- Consider the following example:

  EMP (NAME, SSN, FLT_ID, START_APT, END_APT)

  The obvious FD's are:

  $$\begin{array}{lcl} \{\text{SSN}\} & \to & \{\text{FLT\_ID}\} \\ \{\text{FLT\_ID}\} & \to & \{\text{START\_APT,END\_APT}\} \end{array}$$

  Let F denote the above *collection* of FD's.

  From F, we can infer

  $$\{\text{SSN}\} \to \{\text{START\_APT, END\_APT}\}.$$

  Why? Because, SSN uniquely determines FLT_ID and FLT_ID uniquely determines {START_APT, END_APT}.

  Also, the following FD's are examples of trivial FD's inferred from F.

  $$\begin{array}{lcl} \{\text{SSN}\} & \to & \{\text{SSN}\} \\ \{\text{SSN, NAME}\} & \to & \{\text{NAME}\} \end{array}$$

- **Definition**. Suppose F is a set of FD's. Then, F^+, the **closure** of F is the set of FD's that includes F and all the FD's that can be *inferred* from F.

6.5 Inference Rules for FD Sets

- NOTE the following:
  1. When we say $X \rightarrow Y$, X and Y are *subsets* of the universe of attributes.
  2. For convenience, we will sometimes drop the set notation and commas within sets.

     Suppose F is the set of FD's:

     $$\begin{array}{c} X \rightarrow Y \\ X \rightarrow Z. \end{array}$$

     Then, from F we can infer:

     $$X \rightarrow YZ.$$

     Here, YZ denotes the *union* of Y and Z.

     For example, F is

     $$\begin{array}{ccc} X = \{\text{SSN}\} & \rightarrow & \{\text{NAME }\} = Y \\ X = \{\text{SSN}\} & \rightarrow & \{\text{FLT\_ID}\} = Z. \end{array}$$

     From which we conclude

     $$X = \{\text{SSN}\} \rightarrow \{\text{NAME, FLT\_ID}\} = Y \cup Z = YZ$$

- From above, we can devise a rule: if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.

  Such a rule is called an **inference rule**.

- Standard inference rules for FD's:

  1. **Reflexive rule**. *If* $Y \subseteq X$ *then* $X \rightarrow Y$.

     e.g. {SSN, NAME} → {NAME}

  2. **Augmentation rule**. *If* $X \rightarrow Y$ *then* $XZ \rightarrow YZ$ *for any group of attributes* Z.

     e.g.

     $$\begin{array}{lll} \mathrm{X} & = & \{\mathrm{SSN}\} \\ \mathrm{Y} & = & \{\mathrm{NAME}\} \\ \mathrm{Z} & = & \{\mathrm{START\_APT}\} \end{array}$$

     Then, {SSN} → {NAME} implies

     $$\{\mathrm{SSN,\ START\_APT}\} \rightarrow \{\mathrm{NAME,\ START\_APT}\}$$

  3. **Transitive rule**. *If* $X \rightarrow Y$ *and* $Y \rightarrow Z$ *then* $X \rightarrow Z$.

     e.g., the FD's

     $$\begin{array}{lll} \{\mathrm{SSN}\} & \rightarrow & \{\mathrm{FLT\_ID}\} \\ \{\mathrm{FLT\_ID}\} & \rightarrow & \{\mathrm{END\_APT}\} \end{array}$$

     together imply

     $$\{\mathrm{SSN}\} \rightarrow \{\mathrm{END\_APT}\}.$$

  4. **Decomposition rule**. *If* $X \rightarrow YZ$ *then* $X \rightarrow Y$.

     e.g. the FD

     $$\{\mathrm{SSN}\} \rightarrow \{\mathrm{NAME,\ FLT\_ID}\}$$

     implies

     $$\{\mathrm{SSN}\} \rightarrow \{\mathrm{NAME}\}.$$

5. **Union rule**. *If* $X \to Y$ *and* $X \to Z$ *then* $X \to YZ$.
   e.g. the FD's

   $$\begin{array}{lcl} \{\text{SSN}\} & \to & \{\text{NAME}\} \\ \{\text{SSN}\} & \to & \{\text{FLT\_ID}\} \end{array}$$

   imply

   $$\{\text{SSN}\} \to \{\text{NAME, FLT\_ID}\}.$$

6. **Pseudotransitive rule**. *If* $X \to Y$ *and* $WY \to Z$ *then* $WX \to Z$.
   e.g. consider the relation

   OVERTIME (NAME, SSN, RANK, FLT_ID, START_APT, END_APT, BONUS)

   The FD's

   $$\begin{array}{lcl} \{\text{FLT\_ID}\} & \to & \{\text{START\_APT, END\_APT}\} \\ \{\text{RANK, START\_APT, END\_APT}\} & \to & \{\text{BONUS}\} \end{array}$$

   imply

   $$\{\text{RANK, FLT\_ID}\} \to \{\text{BONUS}\}.$$

- One can use Rules 1-6 to determine F^+.
- It turns out that Rules 1-3 are sufficient to completely determine F^+.

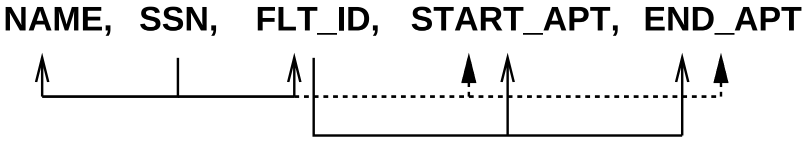
  Rules 1-3 are called **Armstrong's Rules** in honor of the person who proved this result.
- *Equivalent FD sets.*
  - For most FD-sets F, the closure F^+ is probably quite large.
  - While we are interested in the *theoretical* implications of F^+ for any F, we are rarely going to *compute* F^+.
    ⇒ Working with F turns out to be good enough.
  - **Definition**. FD-sets E and F are **equivalent** if $E^+ = F^+$, i.e., if their closures are equal.
  - If E is a set of FD's smaller than F and yet $E^+ = F^+$, then it is easier to work with E.
  - If $E^+ = F^+$ we also say that E **covers** F or is a **cover for** F.

6.6 Attribute Set Closures

- If X is an attribute set, we are often interested in *all* attributes (functionally) determined by X.
  i.e., what is the largest Y for which $X \rightarrow Y$?

  For example, consider

  EMP (NAME, SSN, FLT_ID, START_APT, END_APT)

  Here, SSN determines all the other attributes.
  $\Rightarrow$ {SSN}$^+$ = {NAME,SSN,FLT_ID,START_APT,END_APT}

- **Definition**. If F is a set of FD's and X is a set of attributes, then X^+ is the **closure of X under F** if X^+ is the largest set of attributes functionally determined by X using inference rules on FD's in F.

- Algorithm for determining X^+.

Algorithm: ATTRIBUTE-SET-CLOSURE (X, F)

Input: Attribute set X, FD set F.
Output: Closure of X, X^+.

```
1.  $X^+ := X$;
2.  repeat
3.    old_$X^+ := X^+$;
4.    for each FD $Y \to Z$ in $F$ do
5.      if $Y \subseteq X^+$ then
6.        $X^+ := X^+ \cup Z$;
7.  until old_$X^+ = X^+$;
8.  return $X^+$;
```

For example, suppose F is:

$$\begin{array}{lcl}
\{\text{SSN}\} & \to & \{\text{NAME}\} \\
\{\text{FLT\_ID}\} & \to & \{\text{START\_APT, END\_APT}\} \\
\{\text{SSN}\} & \to & \{\text{FLT\_ID}\}
\end{array}$$

and suppose we want the closure of X={SSN, NAME} under F.

– Initially, X^+ := {SSN, NAME}.

– In the first iteration of the outerloop, in line 3 old_X^+ = {SSN, NAME}.

– Then, the FD {FLT_ID} → {START_ID, END_APT} is processed in the **for**-loop.
  ⇒ since {FLT_ID} is not in X^+, it is ignored.

– Then, the FD {SSN} → {FLT_ID} is processed
  ⇒ it results in X^+ = {SSN, NAME, FLT_ID}.

– After the first iteration of the **repeat**-loop, X^+={SSN, NAME, FLT_ID} and old_X^+={SSN, NAME}
  ⇒ must continue.

- In the second iteration, the FD {FLT_ID} → {START_ID, END_APT} is processed in the **for**-loop.
  ⇒ it results in X^+ = {SSN, NAME, FLT_ID, START_APT, END_APT}.
- After the second iteration of the **repeat**-loop, X^+={SSN, NAME, FLT_ID, START_APT, END_APT} and old_X^+={SSN, NAME, FLT_ID}
  ⇒ must continue.
- No changes in third iteration
  ⇒ stop.

Finally, X^+ = {SSN, NAME, FLT_ID, START_APT, END_APT}.

6.7 Normal Forms

- Normal forms are properties of *relations*.
- There are many normal forms: First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), Boyce-Codd Normal Form (BCNF), etc.
- We say a relation is **in** xNF if its attributes satisfy certain properties (via their FD's).
- Generally, these properties are desirable.
- For example, if we desire the 3NF for a relational database:
  - We will test the relations in the database to see which are in 3NF.
  - Those that are not in 3NF will be decomposed into smaller relations (smaller in numbers of attributes) until we have each relation satisfying the 3NF properties.
- In the real world, most people try to achieve at least 3NF.

  It is slightly better to achieve BCNF (Boyce-Codd Normal Form), but 3NF is considered 'not bad'.
- The end result is: if a database is in BCNF (or 3NF), many anomalies are avoided.
- FD's were designed to test for Normal Forms.
- It is easier to understand normal forms by first considering a simpler version – *normal forms for primary keys.*
- Recall some definitions and notation:

- A **prime attribute** – an attribute belonging to *some candidate key* (not necessarily the primary key).
- A **nonprime attribute** – not belonging to any candidate key.

6.8 1NF: First Normal Form

- **Definition**. A relation is in 1NF if:
  1. the value of any attribute in any tuple is a single value, and
  2. domains of attributes contain only atomic values.
- Example of relations *not* satisfying 1NF:

multiple−valued attribute

PASSENGER	NAME	SSN	FLT_ID	MILES
	Smith	111−22−3333	17	40000
	Jones	222−33−4444	{12, 53, 119}	64000
	Brown	333−44−5555	27	575

⇒ Does not satisfy first part of definition.

Nested structure

PASSENGER	NAME	SSN	FLT_ID	START−APT
	Smith	111−22−3333	17	National
	Jones	222−33−4444	12	JFK
			53	JFK
			119	National
	Brown	333−44−5555	27	Logan

⇒ Does not satisfy second part of definition (contains nested relations).

- It is easy to transform the above relations to satisfy 1NF by:

  1. adding tuples or
  2. creating new relations

  For the first example:

PASSENGER	NAME	SSN	FLT_ID	MILES
	Smith	111−22−3333	17	40000
	Jones	222−33−4444	12	64000
	Jones	222−33−4444	53	64000
	Jones	222−33−4444	119	64000
	Brown	333−44−5555	27	575

Added tuples

  For the second example:

PASSENGER	NAME	SSN	FLT_ID
	Smith	111−22−3333	17
	Jones	222−33−4444	12
	Jones	222−33−4444	53
	Jones	222−33−4444	119
	Brown	333−44−5555	27

FLIGHT	FLT_ID	START−APT
	17	National
	12	JFK
	53	JFK
	119	National
	27	Logan

- 1NF is nowadays taken for granted (i.e., later formulations of relational theory assume relations to be in 1NF).
  ⇒ we will assume all relations are in 1NF.

- Sometimes, raw data not in 1NF is called *unnormalized* data.

6.9 2NF: Second Normal Form

- **Definition**. A FD $X \to Y$ is a **partial dependency** if there exists an attribute $A \in X$ such that

  $$X - A \ \to \ Y.$$

  We say that Y is **partially dependent** on X.

  Example: consider the following relation with primary key {SSN, FLT_ID}

  OVERTIME (NAME, <u>SSN, FLT_ID</u>, BONUS)

  and suppose that bonuses are based on flight duration, and on the crew member's rank and salary.

  Here we can identify some FD's such as

  $$\begin{array}{lcl} \{\text{SSN, FLT\_ID}\} & \to & \{\text{BONUS}\} \\ \{\text{SSN}\} & \to & \{\text{NAME}\} \end{array}$$

  Now, neither one of

  $$\begin{array}{lcl} \{\text{SSN}\} & \to & \{\text{BONUS}\} \\ \{\text{FLT\_ID}\} & \to & \{\text{BONUS}\} \end{array}$$

  is true.
  Thus, {SSN, FLT_ID} $\to$ {BONUS} is not a partial dependency.

  But, {SSN} $\to$ {NAME}
  $\Rightarrow$ NAME is partially dependent on the primary key {SSN,FLT_ID}
  $\Rightarrow$ {SSN, FLT_ID} $\to$ {NAME} is a partial dependency.

- **Definition**. A relation schema is in 2NF if *no* nonprime attribute is partially dependent on the primary key
  (in other words, depends on part of the primary key).

- Thus, in the above example, OVERTIME is not in 2NF.
  (It is, however, in 1NF).

- Why is this a problem?
  Suppose an instance looks like:

OVERTIME	NAME	SSN	FLT_ID	BONUS)
	Smith	111-22-3333	17	450
	Jones	222-33-4444	28	375

Suppose we want to insert the tuple

<Brown, 111-22-3333, 18, 950>.

Because we have the FD: $\{\text{SSN}\} \rightarrow \{\text{NAME}\}$ we will have to check that

<Brown, 111-22-3333,...>

is valid, i.e., that it matches other SSN,NAME values for SSN=111-22-3333
⇒ we have to scan the whole relation (worst-case) to check
⇒ *insertion anomaly.*

Note that

<Brown, 111-22-3333,...>

is not valid.

Similarly, if Flight # 28 is deleted
⇒ we will lose the information "222-33-4444 is the SSN of Jones"
⇒ *deletion anomaly.*

This relation also has a *modification anomaly*:
⇒ if Smith changes name to Brown
⇒ have to propagate change to all relevant tuples.

To solve the problem, consider the decomposition of

OVERTIME (NAME, $\underline{\text{SSN, FLT\_ID}}$, BONUS)

into

OVERTIME (SSN, FLT_ID, BONUS)
PERSONAL_INFO (SSN, NAME)

These relations are in 2NF, with the important FD's:

$$\begin{array}{lcl} \{\text{SSN, FLT\_ID}\} & \rightarrow & \{\text{BONUS}\} \\ \{\text{SSN}\} & \rightarrow & \{\text{NAME}\} \end{array}$$

There are no partial dependencies of nonprime attributes on primary keys
$\Rightarrow$ the new set of relations is in 2NF.

6.10 3NF: Third Normal Form

- **Definition**. The FD $X \rightarrow Y$ is a **transitive dependency** in relation R if there exists a set of attributes Z in R such that

  1. $X \rightarrow Z$ and $Z \rightarrow Y$
  2. Z is not a subset of any key of R

  Example: consider the relation

  EMP (NAME, <u>SSN</u>, POSITION, DEPTNO, DNAME, MGRSSN).

  - Observe that EMP is in 2NF since no partial dependencies exist at all (and hence partial dependencies on the primary key don't exist).
  - Next, consider these FD's (there are others):

  $$\begin{array}{lcl} \{\text{SSN}\} & \rightarrow & \{\text{DEPTNO}\} \\ \{\text{DEPTNO}\} & \rightarrow & \{\text{MGRSSN}\} \end{array}$$

  Note that DEPTNO is not part of any key.
  ⇒ there is a transitive dependency of MGRSSN on SSN.

- **Definition**. A relation R is in 3NF if

  1. it is in 2NF and,
  2. no nonprime attribute is transitively dependent on the primary key.

  In the above example:

  - EMP is in 2NF
  - MGRSSN is a nonprime attribute transitively dependent on the primary key SSN.

  ⇒ EMP is *not* in 3NF.

- Why should we care about 3NF?

  Consider the following instance of EMP:

EMP	NAME	SSN	POSITION	DEPTNO	DNAME	MGRSSN
	Smith	...	...	5	Crew	111-22-3333
	Jones	...	...	6	Ticketing	222-33-4444

  Suppose we insert the tuple <Brown,...,5, Security, 333-44-5555>
  ⇒ we would have to check that the DEPTNO matches the MGRSSN (wrong in this case)
  ⇒ scanning the database
  ⇒ *insertion anomaly.*

  Similarly, if Smith's DEPTNO changes to 6
  ⇒ we will have to also insert the correct MGRSSN in Smith's tuple
  ⇒ *modification anomaly.*

  A deletion anomaly also occurs, if we delete Smith's tuple and Smith is the last Crew employee
  ⇒ we will lose the information "Dept# 6 is Crew"
  ⇒ *deletion anomaly.*

- To solve a 3NF problem, we can decompose relations.

  In the above example:

  E1 (NAME, <u>SSN</u>, POSITION, DEPTNO)
  E2 (<u>DEPTNO</u>, DNAME, MGRSSN)

  Note:

  1. E1 and E2 are in 3NF.
  2. EMP can be recovered by *joining* E1 and E2.
  3. The join will not create spurious tuples.

6.11 General Definitions of 2NF and 3NF

- We have defined 2NF and 3NF using primary keys.
- General definitions based on any candidate key are desirable.
- 2NF:
  - *Primary key version*: A relation schema R is in 2NF if every nonprime attribute A in R is not partially dependent on the primary key.
  - *General version*: A relation schema R is in 2NF if every nonprime attribute A in R is not partially dependent on any key of R.

Consider the following example:

AIRCRAFT_PARTS (MANUF, CODE, PART_ID, DESCR, URL, PRICE).

Here,

- The airline keeps information about aircraft parts.
- PART_ID is the primary key
  ⇒ it is a a unique number assigned by the airline to each part.
- Each part is manufactured by a single manufacturer (MANUF) and the manufacturer uses a code (CODE) to identify the part
  ⇒ the combination {MANUF, CODE} is a key.
- The URL is the (internet) address of the manufacturer's webpage.

Now, each manufacturer has a web page
⇒ we have the FD $\{\text{MANUF}\} \rightarrow \{\text{URL}\}$
⇒ a dependency from part of a key to something else.

Then, AIRCRAFT_PARTS is in 2NF according to the primary version of the definition (no partial dependency on the primary key).

But the partial dependency on MANUF causes the general definition to fail
$\Rightarrow$ AIRCRAFT_PARTS is not in 2NF.

- We *want* the general definition to hold, because otherwise we will have to check the FD $\{\text{MANUF}\} \rightarrow \{\text{URL}\}$ for every insertion.

  We can decompose

  AIRCRAFT_PARTS (MANUF, CODE, PART_ID, DESCR, URL, PRICE).

  into

  PARTS (MANUF, CODE, PART_ID, DESCR, PRICE)
  WEBSITES (MANUF, URL)

  This schema is in 2NF.

- Thus we see how our elaborate definitions of normal forms helps us catch problems in seemingly innocuous schemas (like AIRCRAFT_PARTS).

- 3NF:

  - *Primary key version*: A relation R is in 3NF if
    1. it is in 2NF, and
    2. no nonprime attribute is transitively dependent on the primary key of R.
  - *General version*: A relation R is in 3NF if
    1. it is in 2NF, and
    2. no nonprime attribute is transitively dependent on any key of R.

  Now observe this:

  - If $X \subseteq \{A_1, \ldots, A_n\}$ is any key then $X \rightarrow Y$ for any $Y \subseteq \{A_1, \ldots, A_n\}$
  - Suppose for some relation R
    1. X is the primary key.

2. Y is some other key.
3. Z is *transitively dependent* on Y, i.e., there are FD's

$$Y \to W \text{ and } W \to Z.$$

But $X \to W$ since X is a key.
$\Rightarrow Z$ is transitively dependent on X (the primary key)
$\Rightarrow$ the two 3NF definitions are identical.

For example, consider

AIRCRAFT_PARTS (MANUF, CODE, <u>PART_ID</u>, EMAIL, URL, PRICE).

Then, the FD {EMAIL} → {URL} may be a reasonable choice
⇒ there is a transitive dependency PART_ID → {EMAIL} → {URL}.
But, since {MANUF, CODE} is a key
⇒ {MANUF, CODE} → {PART_ID} trivially
⇒ {MANUF, CODE} → {EMAIL} → {URL}
⇒ transitive dependency on a nonprimary key.

6.12 BCNF: Boyce-Codd Normal Form

- Consider the relation

  $$\text{PARTS } (\underline{\text{MANUF, CODE}}, \text{ URL}).$$

  Suppose that each manufacturer has a webpage form that depends on the part being ordered
  ⇒ a different URL for each part.

  We can identify the natural FD:

  $$\{\text{MANUF, CODE}\} \rightarrow \{\text{URL}\}.$$

  Note that we also have the FD:

  $$\{\text{URL}\} \rightarrow \{\text{MANUF}\}$$

  since a given URL can only correspond to a unique manufacturer.

  Is PARTS in 2NF?

  - Recall: no nonprime attribute should have a partial dependence on a key.
  - Here, a manufacturer has different URL's
    ⇒ no dependence of URL on MANUF.

  ⇒ it passes the 2NF test

  Is PARTS in 3NF?

  - It is in 2NF.
  - Recall: we should not have any transitive dependence on a key.
  - Now, {MANUF, CODE} is the only key
    ⇒ URL is the only attribute left
    ⇒ can't have a transitive dependency with only one nonprime attribute.

  ⇒ PARTS is in 3NF.

- So, what is the problem?

  Unfortunately, PARTS (MANUF, CODE, URL) has all the anomalies (insertion, deletion and modification).

  - The FD {URL} → {MANUF} is the real problem.
  - Suppose, we delete the tuple

    <Boeing, 3395, http://www.boeing.com/parts/737/wing>.

    If this is the only Boeing tuple in the relation, we will lose Boeing's URL
    ⇒ *deletion anomaly.*

  It is easy to check that insertion and modification anomalies are also present.

- The problem appears to be:

  - In 2NF: we did not allow FD's *from* parts of keys *to* nonprime attributes.
  - Here we have an FD *from* an attribute *to* part of a key.

- One option is to introduce the following rule for every relation:

  1. it should be in 3NF
  2. there should be no FD $X \rightarrow Y$ such that Y is part of a key.

  Unfortunately, this rule is too *restrictive.*
  e.g., consider

  AIRCRAFT_PARTS (MANUF, CODE, PART_ID, DESCR, PRICE).

  Here,

  - PART_ID is the primary key.
  - {MANUF, CODE} is another key.
  - The FD {PART_ID} → {MANUF} follows.

Thus, if we disallow relations of this sort, we will be essentially barring all non-primary keys from having multiple attributes.
⇒ may be too restrictive in practice.

- Let us try to soften the rule:

  1. it should be in 3NF
  2. for every FD $X \rightarrow Y$, such that Y is part of a key, X should itself be a key.

  That is, we do allow *part of keys* to be dependent on things – provided those things are keys.

  This is a reasonable assumption because:

  - If X is a key, we would likely have to check uniqueness anyway (and that's all we have to do – using the **unique** keyword in SQL).
  - Deletion causes less of an anomaly, e.g., in

    AIRCRAFT_PARTS (MANUF, CODE, <u>PART_ID</u>, DESCR, PRICE).

    the FD {PART_ID} → {MANUF} is not important.
    Deleting the only tuple with 'Boeing', e.g.,

    <Boeing, 423, 12, Coat-rack, $50>,

    we lose the information "Part_id 12 is made by Boeing".
    But, if we delete the tuple

    <Boeing, 423, Coat-rack, $50>

    we are really saying "Boeing is the only company who makes the coat-racks we use, and we don't need coat-racks"
    ⇒ it's OK to lose "Part_id 12 is made by Boeing".

- The softened rule above needs a small modification:
  Observe that in the above example, we have the FD

  {PART_ID} → {MANUF}.

This passes our new test.

However, the following is also an FD:

$$\{\text{PART\_ID, PRICE}\} \rightarrow \{\text{MANUF}\}.$$

This fails the test because {PART_ID,PRICE} is not a key.

However, it is a *superkey* (contains a key).

- Final form:
  Definition. A relation R is in Boyce-Codd Normal Form (BCNF) if:

  1. it is in 3NF
  2. for every FD $X \rightarrow Y$ such that Y is part of a key, X is a superkey.

6.13 3NF and BCNF: An Alternate Definition

- First recall the 3NF definition:
  1. 2NF
  2. no transitive dependency from a key to a nonprime attribute should exist.

  Here, a transitive dependency means:

  - $X \to Y$ and $Y \to Z$
  - Y is not part of any key
  - X is a key
  - Z is a nonprime attribute

  Now, since X is a key, the FD $X \to Y$ must be true for any Y.

  Thus, the condition is really saying (given X is a key) that for $Y \to Z$:

  - (a) Y is not part of any key
  - (b) Z is a nonprime attribute

  Next, recall that 2NF is essentially:

  - if $Y \to Z$ then Y cannot be a proper subset of a key.

  Combine this with the first item (a) in 3NF and write (b) separately: A relation R is in 3NF if for every FD $Y \to Z$ either

  1. Y is a superkey, or
  2. Z is a prime attribute.

- This is an alternate definition of 3NF which does not mention 2NF.

- Note that if Z is a prime attribute, we allow $Y \to Z$ even if Y is not a superkey, e.g. in

  PART (MANUF, CODE, URL)

  we allowed {URL} → {MANUF} because {MANUF} is a prime attribute (it is part of the key {MANUF, CODE}).

  But BCNF does not allow this.

  Hence, an alternate definition of BCNF is:
  A relation R is in BCNF if for every FD $Y \to Z$, Y is a superkey of R.

  - This definition does not use 3NF.

- NOTE:

  - We must be careful to rule out trivial dependencies from consideration:
    - The dependency $X \to A$ where attribute $A \in X$ is called a *trivial dependency.*
    - Example: {SSN, NAME} → {SSN}.
    - We rule out trivial dependencies because they occur with any subset of attributes.
  - Suppose $Y = \{A_1, ..., A_k\}$ is a subset of attributes and $X \to Y$.
    - We know that $X \to A_i$ for $i = 1, ..., k$ by the decomposition rule.
    - Thus, the BCNF definition can also be stated as: *a relation is in BCNF if every nontrivial dependency is one in which a superkey determines an attribute.*

  - Informally, the only functional dependencies in a BCNF go from keys to other attributes.
  - Example:
    - Suppose X is a superkey and $X \rightarrow A$ in a BCNF relation, R.
    - Suppose B is some other attribute.
    - Consider the following tuples:

R	X	A	B
	x	a	b
	x	a	?

    - The value $X = x$ determines the value $A = a$.
    - But could we have different B-values?
    - Different B-values raise the familiar problem of $X \rightarrow A$ anomalies.
    - But since $R \in BCNF$, X is a superkey and so $X \rightarrow B$ (simply by being a superkey).
    - Thus, the B-value must be b.
    - Since we can't have duplicate tuples, it won't be allowed.

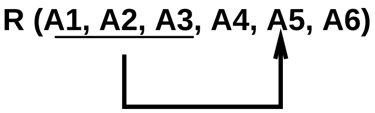
- Finally, observe that

$$\begin{array}{rcl} R \text{ is in BCNF} & \Rightarrow & R \text{ is in 3NF} \\ & \Rightarrow & R \text{ is in 2NF} \\ & \Rightarrow & R \text{ is in 1NF} \end{array}$$

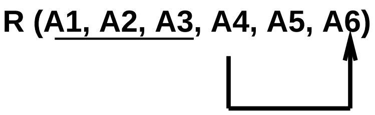
6.14 Another View of Normal Forms

- If the discussion so far has been confusing, let us try to explain normal forms a little differently.
- First, some simplifications:
  - Let us only consider relations with a single key – a primary key.
  - Assume this key has several attributes.

  Note: this simplification is only for conveying the key idea behind normal forms. In practice you would have to use the full definition.
- Let $R(A_1, \ldots, A_6)$ be a relation with primary key $\{A_1, A_2, A_3\}$.
- 2NF says: FD's like $A_2 \to A_5$ are not allowed.
  $\Rightarrow$ a proper subset of a key should not be on the left side of an FD:

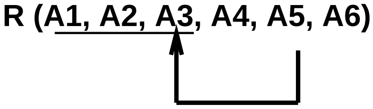


- 3NF says:
  1. at least 2NF, and
  2. FD's like $A_4 \to A_6$ are not allowed.
     $\Rightarrow$ an FD between non-key attributes is not allowed:



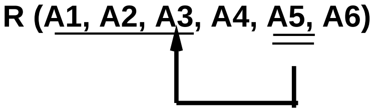
- Next, BCNF:

  Unfortunately 3NF allows an FD like $A_5 \rightarrow A_3$, where A_5 is nonprime and A_3 is part of the key:



  We saw why this was a problem in the BCNF example.

  On the other hand we did not want to be too restrictive: if A_5 happened to be a key we would allow it:



- The key ideas above are generalized to allow for:
  - multiple keys in a relation
  - keys consisting of groups of attributes.

6.15 Decomposition and its Problems

- We have seen that it is desirable to have relations in BCNF (or at least 3NF).
- We have seen how to *test* for BCNF and 3NF.
- But how do we create a BCNF (or 3NF) database?
  - One approach: Ad-hoc
    - Create relations intuitively
    - Test each for BCNF
  - More formal approach:
    - Start with a single large relation with all attributes
    - Systematically decompose relations not in BCNF
    - Repeat until all relations are in BCNF
- Unfortunately, decomposition can create problems:
  - Dependencies may be *lost* after decomposition.
  - Joins of decomposed relations may create *spurious tuples.*
  - Joins of decomposed relations may *lose tuples.*
- Note: thus far, we have identified problems with *individual* relations
  ⇒ we have not placed constraints *among* multiple relations.

6.16 Dependency Preservation

- Suppose we decompose $R = (A_1, \ldots, A_n)$ into relations $R_1, \ldots, R_m$.
- Of course, we should have **attribute preservation**, i.e., attributes should not be lost in the shuffle:

$$R_1 \cup R_2 \cup \ldots R_m = R$$

- Unfortunately, FD's can be lost, e.g.,
  - Suppose F is a set of FD's containing the dependency

    $$\{\text{PART\_ID}\} \rightarrow \{\text{PRICE}\}.$$

  - Suppose also that {PART_ID} is put in relation R_3 and {PRICE} is put in relation R_7
    ⇒ we can't check the dependency.
- In the above example, the lost FD would be OK, if the dependency were somehow not important
  ⇒ we need to consider the *closure* of FD's.
- **Definition**. Let F be a set of FD's and suppose R is decomposed into relations $R_1, \ldots, R_m$. Let E be the set of FD's in $R_1, \ldots, R_m$. Then the decomposition is **dependency preserving** if $E^+ = F^+$.
- **Fact**: It is always possible to decompose any relation R into 3NF relations $R_1, \ldots, R_m$ such that the decomposition is dependency preserving.

6.17 Nonadditive and Lossless Decompositions

- Suppose we decompose R into $R_1, \ldots, R_m$. Later, we wish to recover R (perhaps as a view).
  - The natural join on $R_1, \ldots, R_m$ should return R.
  - If we're not careful, this join can create spurious tuples e.g, consider the relation r with schema R =(CAR,OWNER,COLOR):

CAR	OWNER	COLOR
Toyota	Smith	blue
Ford	Jones	blue

Suppose we decompose this into r_1 and r_2 with schemas R_1=(CAR,COLOR) and R_2=(OWNER,COLOR).

How? Let $r_1 = \Pi_{\text{CAR,COLOR}}(r)$ and $r_2 = \Pi_{\text{OWNER,COLOR}}(r)$:

CAR	COLOR
Toyota	blue
Ford	blue

OWNER	COLOR
Smith	blue
Jones	blue

What happens when we join r_1 and r_2?

CAR	OWNER	COLOR
Toyota	Smith	blue
Toyota	Jones	blue *
Ford	Smith	blue *
Ford	Jones	blue

⇒ Spurious tuples!

- A decomposition of R into $R_1, \ldots, R_m$ is **nonadditive** if for every instance r of R, the natural join of the corresponding instances $r_1, \ldots, r_m$ is equal to r, i.e.,

$$r_1 * r_2 * \ldots * r_m = r$$

where $r_i = \Pi_{R_i}(r)$.

- Note: **nonadditive** is the same as 'creates no spurious tuples'.
- Sometimes, one can inadvertently *lose* tuples in a join.
- Example:

  Suppose EMP (SSN, NAME, FLT_ID, DEPTNO) has too many NULLs in the DEPTNO attribute (because many employees have no assigned department).

EMP	SSN	NAME	FLT_ID	DEPTNO
	111-22-3333	Smith	12	NULL
	222-33-4444	Jones	55	6
	333-44-5555	Brown	119	NULL

⇒ One solution is to decompose EMP into two relations:

EMP1 (SSN, NAME, FLT_ID)
DEPT (SSN, DEPTNO)

e.g.,

EMP1	SSN	NAME	FLT_ID
	111-22-3333	Smith	12
	222-33-4444	Jones	55
	333-44-5555	Brown	119

DEPT	SSN	DEPTNO
	222-33-4444	6

Here,

- Only those employees assigned to a department will have department numbers
  ⇒ DEPT is small.
- Both EMP1 and DEPT are in BCNF.
- The decomposition is nonadditive and dependency preserving.

Consider the join EMP1 $*$ DEPT:

EMP1 * DEPT	SSN	NAME	FLT_ID	DEPTNO
	222-33-4444	Jones	55	6

⇒ the 'Smith' and 'Brown' tuples are lost!
⇒ we have to be careful in letting joins replace relations.

- We call a decomposition is *lossless* if it does not lose tuples in recovering the original relation.
- Note:
  - Nonadditivity and losslessness are two sides of the same coin.
  - We will use the term *lossless* to refer to both.
  - Some books use *additive* to refer to both.
- It would be useful, if given a decomposition, to test whether the decomposition is lossless.
- **A useful fact**. A decomposition of R into R_1 and R_2 is nonadditive with respect to a set of FD's F, if and only if either one of the FD's
  - $R_1 \cap R_2 \ \rightarrow R_1 - R_2$
  - $R_1 \cap R_2 \ \rightarrow R_2 - R_1$

  is in F^+.

  Intuition:
  - Observe: the attributes $R_1 \cap R_2$ are in both R_1 and R_2
    ⇒ these are the join attributes.
  - Suppose the FD $R_1 \cap R_2 \ \rightarrow R_1 - R_2$ holds.
    This is the same as $R_1 \cap R_2 \ \rightarrow R_1 - (R_1 \cap R_2)$
    ⇒ $R_1 \cap R_2$ is a *key* for R_1.
    ⇒ weird, unwanted tuple combinations can't occur.
- In the R=(CAR,OWNER,COLOR) example:

We joined

CAR	COLOR
Toyota	blue
Ford	blue

OWNER	COLOR
Smith	blue
Jones	blue

to get

CAR	OWNER	COLOR
Toyota	Smith	blue
Toyota	Jones	blue *
Ford	Smith	blue *
Ford	Jones	blue

Note that {COLOR} is not a key for either relation.

Here, R_1=(CAR,COLOR) and R_2=(OWNER,COLOR) and,

$$\begin{aligned} R_1 \cap R_2 &= \text{COLOR} \\ R_1 - R_2 &= \text{CAR} \\ R_2 - R_1 &= \text{OWNER} \end{aligned}$$

Clearly, the neither of FD's

$$\begin{aligned} \{\text{COLOR}\} &\rightarrow \{\text{CAR}\} \\ \{\text{COLOR}\} &\rightarrow \{\text{OWNER}\} \end{aligned}$$

hold
⇒ the decomposition is *not* nonadditive.

6.18 Algorithms for Decomposition of Relations

- First, recall that an FD set E *covers* FD set F if $E^+ = F^+$, i.e., the closure of E is the closure of F
  ⇒ if E is smaller it will be easier to work with
  ⇒ it is useful to determine the *minimal cover* for an FD set F.

- *Minimal covers* can be defined in a number of ways:

  – E is an *FD-minimal cover* of F if E covers F and no other FD set covers F that has fewer FD's than E.

  – E is an *attribute-minimal cover* of F if E covers F and no other FD set covers F with fewer attributes.

  – E is a *left-minimal cover* of F if E covers F and no other FD set covers F with smaller left-hand-sides.

  Note:

  – Computing an attribute-minimal cover is a hard (NP-complete) problem
    ⇒ no fast algorithm is known.

  – Computing FD-minimal covers and left-minimal covers is fairly straightforward.

  – Left-minimal covers are all that's needed for 3NF decompositions.

- Key ideas in finding a left-minimal cover:

  – Consider the relation
  EMP (NAME, SSN, FLT_ID, START_APT, END_APT).

  and the FD set F

$$\begin{array}{lllll}
(1) & \{\text{SSN, NAME}\} & \rightarrow & \{\text{FLT\_ID, START\_APT}\} \\
(2) & \{\text{SSN}\} & \rightarrow & \{\text{FLT\_ID}\} \\
(3) & \{\text{FLT\_ID}\} & \rightarrow & \{\text{START\_APT}\} \\
(4) & \{\text{SSN}\} & \rightarrow & \{\text{NAME}\} \\
(5) & \{\text{SSN, NAME}\} & \rightarrow & \{\text{FLT\_ID}\}
\end{array}$$

Note that, given (2) and (4), we don't need (5) since the combination of (2) and (4) will imply (5).

- The first step is to break up the FD's so that the right-hand-sides are only single attributes:

$$\begin{array}{lll}
\{\text{SSN, NAME}\} & \rightarrow & \{\text{FLT\_ID}\} \\
\{\text{SSN, NAME}\} & \rightarrow & \{\text{START\_APT}\} \\
\{\text{SSN}\} & \rightarrow & \{\text{FLT\_ID}\} \\
\{\text{FLT\_ID}\} & \rightarrow & \{\text{START\_APT}\} \\
\{\text{SSN}\} & \rightarrow & \{\text{NAME}\}
\end{array}$$

- Next, see if left-hand-side attributes can be removed.
    - For example, consider the FD {SSN, NAME} → {FLT_ID}.
    - The left-hand-side here is {SSN, NAME}.
    - Suppose we remove SSN: {SSN, NAME} - {SSN}.
    - Can we replace the earlier FD with {NAME} → {FLT_ID}?
    - To check, we compute the attribute closure of the new left-hand-side, i.e., check whether $(\{\text{SSN, NAME}\} - \{\text{SSN}\})^+$ contains the right-hand-side $\{FLT\_ID\}$.
    - In this case, $\{NAME\}^+$ does *not* contain {FLT_ID}
      ⇒ cannot remove {SSN} from {SSN, NAME} → {FLT_ID}.

- **Definition**: An FD $X \rightarrow Y$ where Y has more than one attribute is called a *multiple-RHS* FD.

- Algorithm for computing a minimal cover:

```
Algorithm: LEFT-MIN-COVER ($F$)

Input: An FD set $F$.
Output: A left-minimal cover $E$.
$E$ := $F$;
for each multiple-RHS FD $X \rightarrow A_1 A_2 ... A_k$ in $E$
   $E$ := $E - \{X \rightarrow A_1 A_2 ... A_k\}$;
   for $i \leftarrow 1$ to $k$
      $E$ := $E \cup \{X \rightarrow A_i\}$;
endfor
// All multiple-RHS FD's have been replaced by single-RHS FD's
for each FD $X \rightarrow A$ in $E$
   $X^+$ := ATTRIBUTE-SET-CLOSURE ($X$, $E - \{X \rightarrow A\}$);
   if $A \in X^+$
      $E$ := $E - \{X \rightarrow A\}$;
endfor
// Now, we are rid of unnecessary FD's. Next, reduce left-hand-sides
for each FD $X \rightarrow A$ in $E$
   for each attribute $B \in X$
      $D$ := $E - \{X \rightarrow A\} \cup \{(X - B) \rightarrow A\}$;
      $(X - B)^+$ := ATTRIBUTE-SET-CLOSURE ($X - B$, $D$);
      if $A \in (X - B)^+$
         $E$ := $E - \{X \rightarrow A\}$;
         $E$ := $E \cup \{(X - B) \rightarrow A\}$;
      endif
   endfor
return $E$;
```

- The following algorithm decomposes a relation R into a set of 3NF relations $R_1, R_2, ...$ that are dependency-preserving and nonadditive.

```
Algorithm: 3NF-DECOMPOSITION ($R$, $F$)

Input: Relation $R = (A_1, ..., A_k)$ with FD set $F$.
Output: 3NF decomposition $R_1, R_2, ...$
  1.  $E$ := LEFT-MIN-COVER ($F$);
  2.  for each left-hand-side $X_i$ in $E$
  3.      $R_i$ := $\{X_i\}$;
  4.      for each $X_i \rightarrow A_j$ in $F$
  5.          $R_i$ := $R_i \cup \{A_j\}$;
  6.  endfor
      // At this point we have a collection of relations $R_1, ..., R_n$
  7.  $R_{n+1}$ := $\emptyset$;
  8.  for each $A_i \notin R_1 \cup ... \cup R_n$
  9.      $R_{n+1}$ := $R_{n+1} \cup \{A_i\}$;
  10. return $R_1, ..., R_{n+1}$;
```

- Why does this work?
  - First note that all the FD's find their way (see lines 4-5) into the decomposition
    $\Rightarrow$ it is dependency-preserving.
  - Are the resulting relations in 3NF?
    - Consider a transitive dependency $X \rightarrow Y \rightarrow Z$ in the original relation.
    - The FD $X \rightarrow Z$ will be removed in minimal cover since $X \rightarrow Y$ and $Y \rightarrow Z$ are sufficient to generate $Z \in Z^+$.
    - Thus, the decomposition (lines 4-5 above) will not create a relation with attributes (X, Y, Z) and thus transitive dependencies will be removed.
  - Is the decomposition nonadditive?
    - Note this general property: if R is a relation and $X \rightarrow A$ is an FD

then the decomposition $R - A$ and $R' = (X, A)$ is nonadditive. Why? Because a join of $R - A$ and R' only involves X and since X determines A, no spurious tuples will be created.

  - In the above algorithm, all decomposition steps are of the above type.

- An algorithm for decomposing a relation into a collection of *nonadditive BCNF* relations.

```
Algorithm:    NONADDITIVE-BCNF-DECOMPOSITION ($R$, $F$)

Input: Relation $R$, FD set $F$.
Output: A nonadditive BCNF decomposition $R_1, R_2, \ldots$
  $R_1, ..., R_k$ := 3NF-DECOMPOSITION ($R$, $F$);
  while $\exists R_i \in R_1, ..., R_k$ not in BCNF
     if $X \rightarrow Y$ is an FD in $R_i$ that violates BCNF
        $R_i := R_i - Y$;
        $k := k + 1$;
        $R_k := (X, Y)$;
     endif
  return $R_1, ..., R_k$;
```

Intuition:

If $X \rightarrow Y$ is in some R_i and it violates BCNF
 $\Rightarrow$ X is not a superkey of R_i (definition of BCNF)
 $\Rightarrow$ we create a relation $R_k = X \cup Y$
Here $X \rightarrow Y$ implies X is a superkey for R_k
 $\Rightarrow$ since Y is removed from R_i, it does not cause the BCNF violation.

- Unfortunately, we can't always decompose R into BCNF relations that are *both* dependency preserving and nonadditive.

  – The nonadditive property is preserved by the above algorithm.

- It may produce a decomposition that is not dependency-preserving.
- In general, it is impossible to achieve both.
- Example: consider the relation

  $$\text{PARTS } (\underline{\text{MANUF, CODE}}, \text{ URL})$$

  where each part has a unique URL.

  - The FD's are:

    $$\begin{array}{lcll} \{\text{MANUF, CODE}\} & \rightarrow & \{\text{URL}\} & \text{(unique URL for each part)} \\ \{\text{URL}\} & \rightarrow & \text{MANUF} & \text{(knowing a URL tells you the manufacturer)} \end{array}$$

  - The PARTS relation is not in BCNF since we have a dependency from an attribute (URL) to part of a key (MANUF).
  - Any decomposition will have to separate URL from MANUF.
  - This means the FD {MANUF, CODE} → {URL} cannot be preserved in the decomposition
    ⇒ no BCNF decomposition of PARTS can be dependency-preserving.

6.19 Formal Schema Design: A Summary

- We saw that ad-hoc designs led to anomalies with insertions, deletions and modifications.
- To analyze relations, we developed the theory of normalization:
  - Definition of functional dependency (FD).
  - Properties of FD's.
  - Computation of attribute closures.
  - Definition of Normal forms (primary and general versions).
- In practice: try to achieve BCNF. If not possible, live with 3NF.

  If your design is not in 3NF
  ⇒ you have a weird schema.
- Also need to check for *nonadditivity* and *dependency preservation*.
- Before using a join to replace existing relations, check to see tuples don't get lost.
- Sometimes, a BCNF decomposition or a 3NF decomposition can lead to inefficiencies
  ⇒ many queries require expensive joins.
  In this case, one sometimes permits BCNF and 3NF violations for efficiency reasons
  ⇒ violations can be checked separately at leisure.
- Some issues we have not covered:
  - Formal proofs asserting the correctness of algorithms.
  - Finding minimal FD-sets with other definitions of minimality.

- General mechanisms for testing nonadditivity (an algorithm called the Tableau Chase Method).
- Multivalued FD's and 4NF.
- Other dependencies and normal forms.

Chapter 7

Transaction Processing: Recovery and Concurrency Control

Course Notes on Database Systems

7.1 What is a Transaction?

- To explain what a transaction is, we'll first consider an example:
  - Consider the McVALUE Airlines database with the following additional relations: CORP_ACCOUNTS and BILLING:

CORP_ACCOUNTS	CORP_ID	CNAME	BALANCE
	3	IBM	649,314
	19	Intel	213,617
	7	GM	65,973
	42	DuPont	143,112

BILLING	PNAME	CORP_ID	BALANCE
	Sam	19	615
	Joe	7	700
	Sue	42	419
	Pam	0	445

    BILLING contains the outstanding balance for individual passengers. CORP_ACCOUNTS contains the amount owed by some corporate clients.

  - Some individuals belonging to a corporation can have their ticket charged to the corporate account.

    For example, Joe's outstanding balance is \$700 – it can be charged to his corporate (GM's) account by transferring the amount \$700 to 65,973.

    ⇒ Joe's balance becomes 0, GM's balance becomes 66,673.

  - A corporate id of 0 indicates no corporate affiliation.
  - An embedded SQL program to achieve corporate billing looks like this:

```
EXEC SQL DECLARE SECTION
  varchar pass_name[50];      // Passenger name
```

```
    int cid;                      // Corporate id
    int amount;                   // Amount outstanding
  EXEC SQL END DECLARE SECTION
  // Read in passenger name from screen (not shown)
  amount = 0;
  EXEC SQL   SELECT B.CORP_ID, B.BALANCE  // Get the guy's
             INTO :cid, :amount           // Corp_id and balance
             FROM BILLING B
             WHERE B.PNAME = :pass_name;
  if (cid > 0) {                          // Corporate employee?
    EXEC SQL   UPDATE BILLING             // Zero-out balance
               SET BALANCE = 0
               WHERE PNAME = :pass_name;
    EXEC SQL   UPDATE CORP_ACCOUNTS       // Add to corporation
               SET BALANCE = BALANCE + :amount
               WHERE CORP_ID = :cid;
  }
```

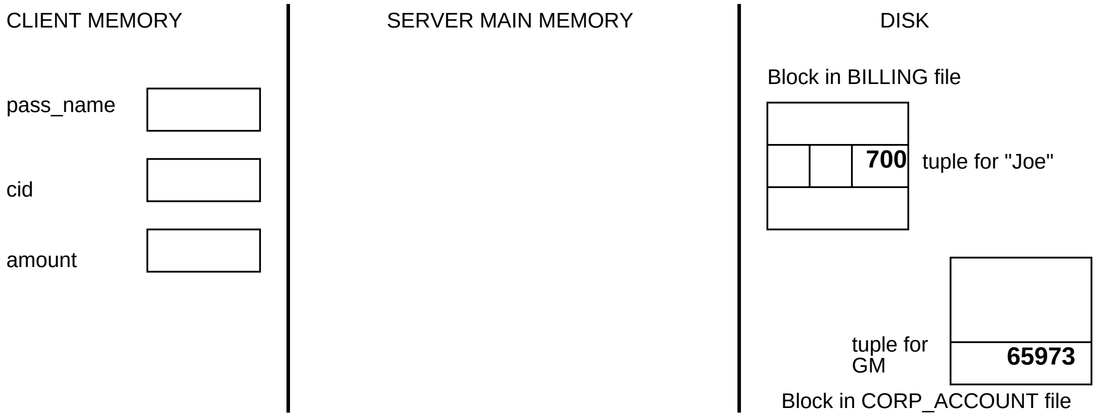
– For example, if the passenger name entered is "Joe", the relations are updated to:

CORP_ACCOUNTS	CORP_ID	CNAME	BALANCE	
	3	IBM	649,314	
	19	Intel	213,617	
	7	GM	66,673	←
	42	DuPont	143,112	

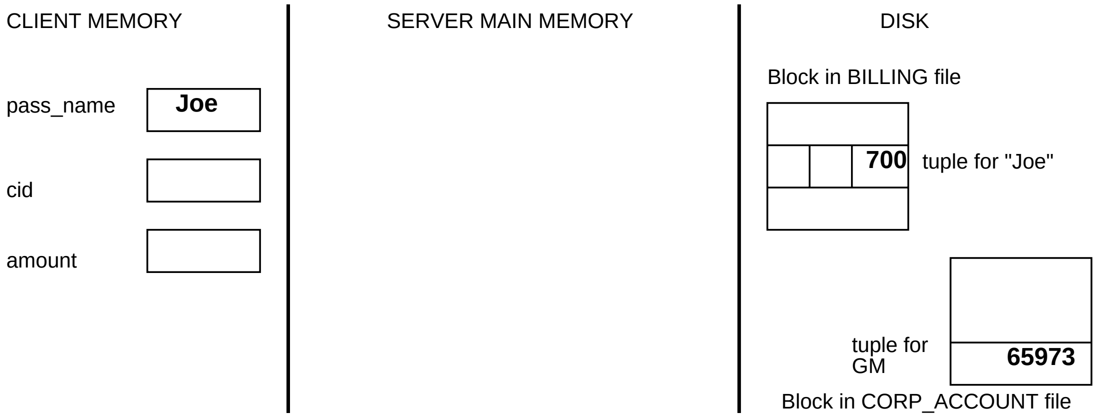
BILLING	PNAME	CORP_ID	BALANCE	
	Sam	19	615	
	Joe	7	0	←
	Sue	42	419	
	Pam	0	445	

- Let us observe the interaction between main memory and disk when this change is made:

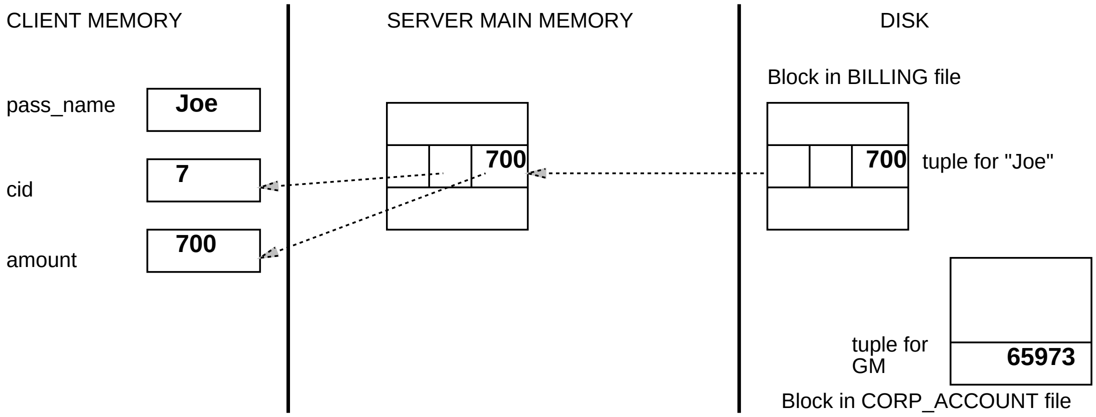
1. Initially, data is only on disk:



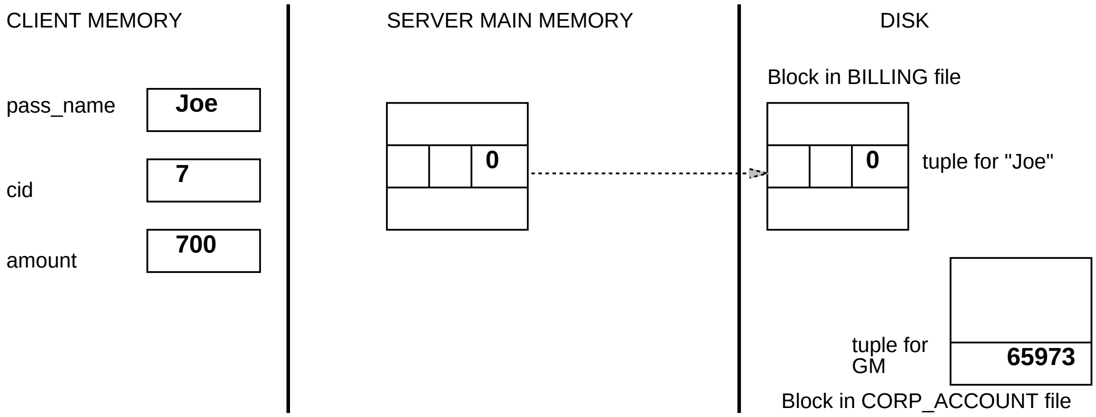
2. Read in passenger name ("Joe") from keyboard:



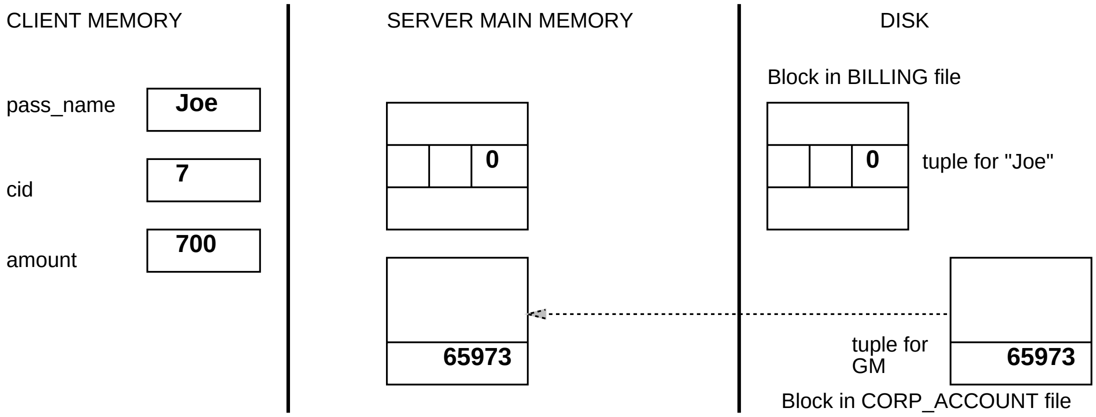
3. After first SQL statement, Joe's corporate id (7) and outstanding balance ($700) are read into the client variables `cid` and `amount`:



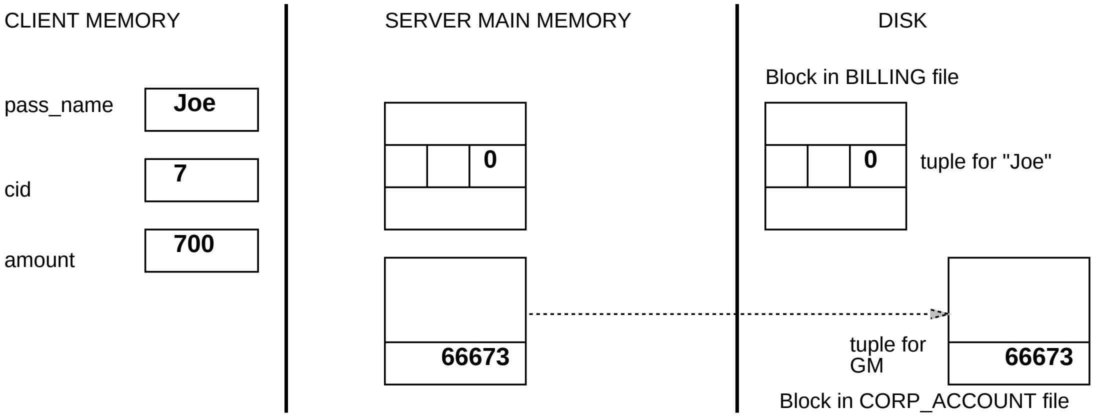
4. After second SQL statement, Joe's balance is zeroed out:



5. For third SQL statement, the block containing GM's tuple in CORP_ACCOUNTS is first fetched:



Then, it is updated and written to disk:



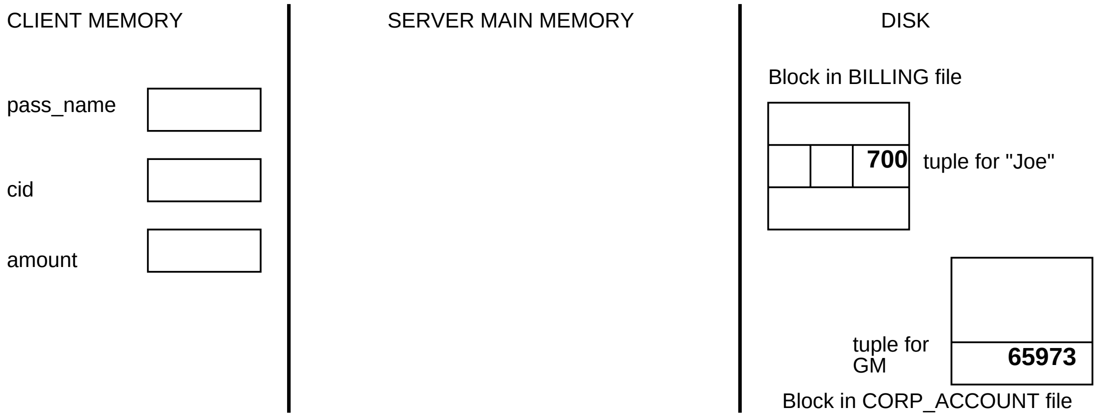
- Note: the method of immediately writing (*Immediate-Write Method*) to disk every update can be very inefficient for multiple updates to a block.
  - For example, consider the following SQL statement:

    update CORP_ACCOUNTS
    set BALANCE = BALANCE $*$ 0.9;

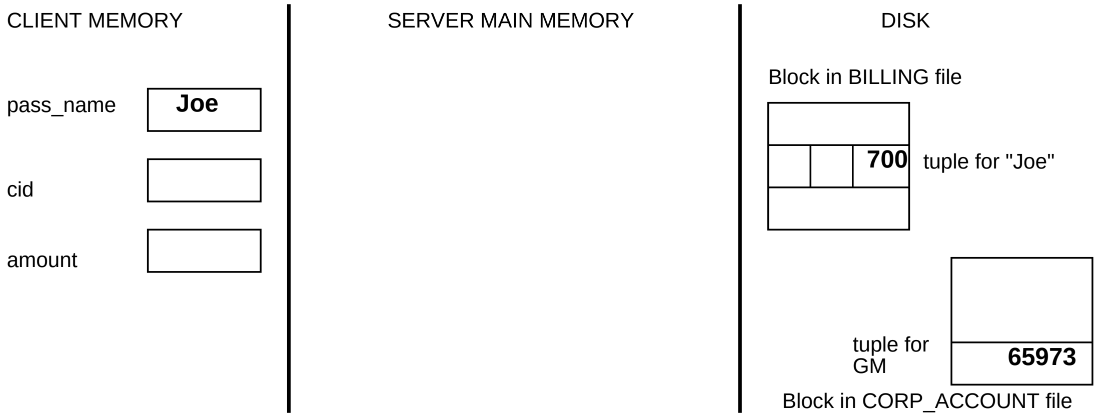
  - Next, suppose a block contains 50 tuples
    ⇒ 50 disk writes for each block
    ⇒ 1 second per block (at 20 ms per access)
  - It's better to postpone writing a block until either:
    1. Memory space is needed for other blocks
       ⇒ must write some block to disk.
    2. All the work for the block is done.

  Thus, an alternative scenario (*Deferred-Write Method*) for the BALANCE example is:

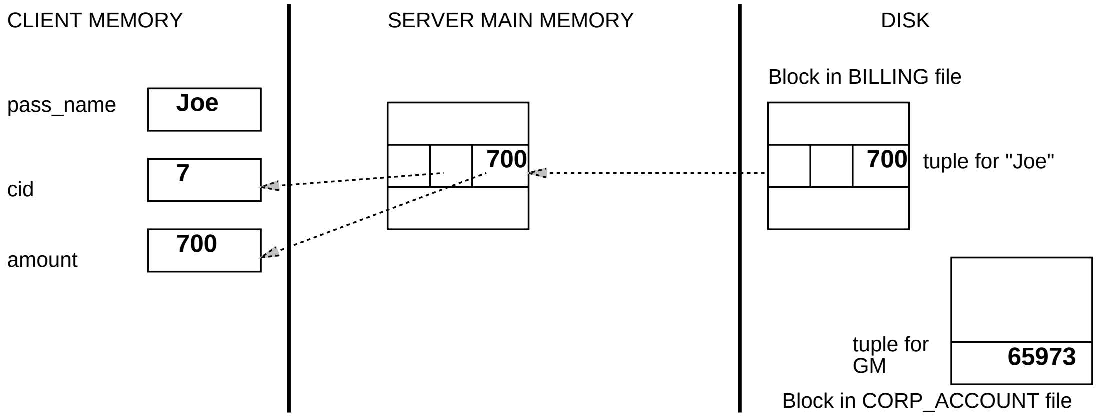
- Initially, data is only on disk:



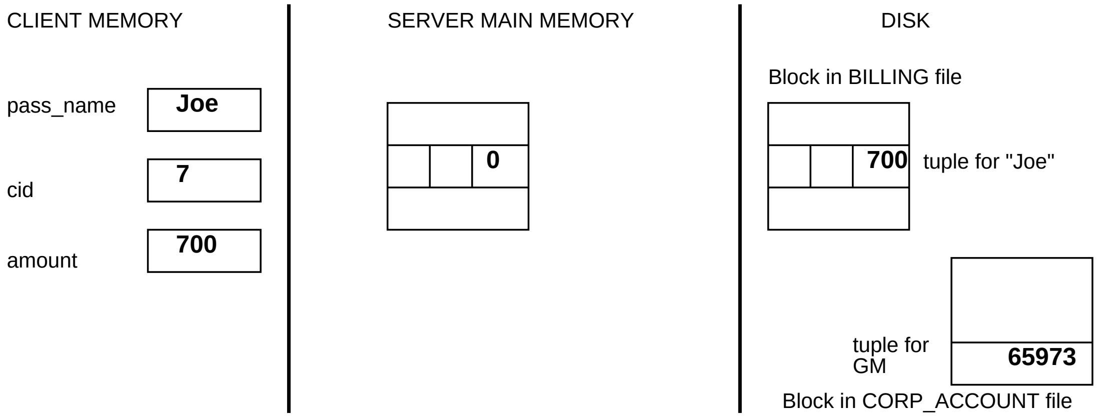
- Read in passenger name ("Joe") from keyboard:



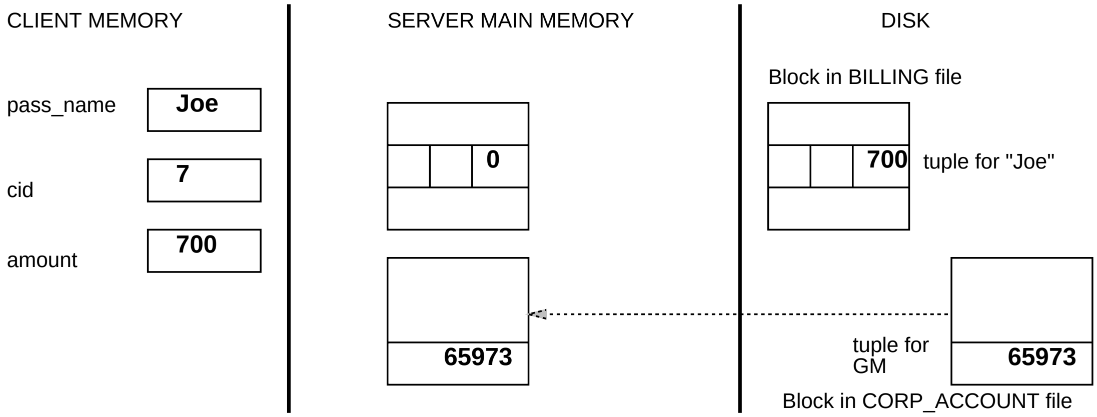
- After first SQL statement, Joe's corporate id (7) and outstanding balance ($700) are read into the client variables `cid` and `amount`:



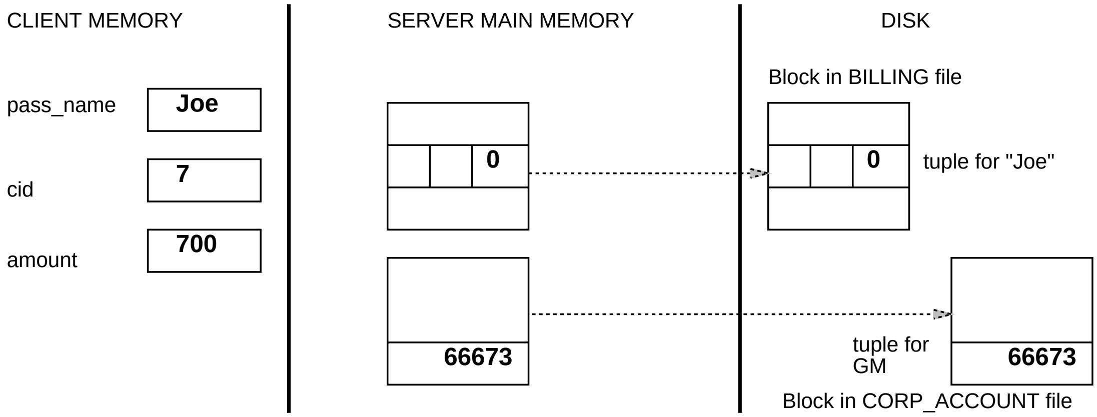
- After second SQL statement, Joe's balance is zeroed out *only in memory*:



- For third SQL statement, the block containing GM's tuple in CORP_ACCOUNTS is first fetched:



- After all modifications are made, the updated blocks are written to disk:



- In either scenario, an important question arises: what happens if the system crashes before updates to disk are complete?
  - In the first scenario, system could crash after '0' is written to Joe's tuple, but the corporate tuple is untouched
    ⇒ inconsistent data left in system.
  - In the second scenario, system could crash during either disk write's
    ⇒ inconsistent data or incomplete execution.
- Thus, we would like the embedded SQL program to change the database *completely* or *not at all*.
- **Definition**: Computations that are required to execute completely or not have any effect on the database are called *transactions*.
- Typically, most transactions are of short duration (like the above example). However, some can be long, e.g.,

  ```
  update  CORP_ACCOUNTS
  set     BALANCE = BALANCE * 0.9;
  ```

  (affects all the data in one relation).
- More formally, a transaction is a piece of code that has the **ACID** properties:
  - **A***tomicity*: Either a transaction completes its changes to the database or it does not modify the database at all.
  - **C**onsistency: A transaction should not violate integrity constraints on the data.

    For example, an account transfer transaction should maintain the total to be the same as it was before the transaction.
    ⇒ programming should be correct.
  - **I**solation: If transactions X_A and X_B run concurrently, then the effects to the database should be the same as if they ran in some serial order (either X_A, X_B or X_B, X_A).

  - **D**urability: When a transaction completes, its effects on the database must be guaranteed to be written to disk.

- **Definition**: when a transaction “completes its work”, we say it *commits*.

7.2 Specifying Transactions in SQL

- SQL provides commands for managing transactions.
- Transaction processing commands in SQL:
  - **commit** or **commit work**
    - declare that a transaction's work is completed.
  - **rollback** or **rollback work**
    - cancel a transaction
  - **set transaction** [...]
    - set isolation properties (how exclusively it locks its data).
- For example, let's consider a small variation of our running example:
  - Transfer a particular passenger's balance to his/her corporate account.
  - *A corporation's outstanding balance is not allowed to be higher than $100,000.*
  - The following embedded SQL program is a transaction that achieves the task:

```
EXEC SQL DECLARE SECTION
  varchar pass_name[50];      // Passenger name
  int cid;                    // Corporate id
  int cbalance;               // Corporate balance
  int amount;                 // Amount outstanding
EXEC SQL END DECLARE SECTION
// Read in passenger name from screen (not shown)
```

```
amount = 0;
EXEC SQL   SELECT B.CORP_ID, B.BALANCE  // Get the guy's
           INTO :cid, :amount           // Corp_id and balance
           FROM BILLING B
           WHERE B.PNAME = :pass_name;
if (cid > 0) {                          // Corporate employee?
  EXEC SQL   UPDATE BILLING             // Zero-out balance
             SET BALANCE = 0
             WHERE PNAME = :pass_name;
  EXEC SQL   SELECT C.BALANCE           // Fetch corp. balance
             INTO :cbalance
             FROM CORP_ACCOUNTS C
             WHERE CORP_ID = :cid
  if (cbalance + amount <= 100000) {
    EXEC SQL   UPDATE CORP_ACCOUNTS       // Add to corporation
               SET BALANCE = BALANCE + :amount
               WHERE CORP_ID = :cid;
    EXEC SQL   COMMIT;                    // Commit transaction
  }
  else {  // Cancel transaction
    EXEC SQL   ROLLBACK;
  }

}
```

– Note: start of transaction is implicit.

7.3 Transaction Processing: The Read-Write Model

- The *read-write model* is a way to abstract away SQL detail and concentrate on what matters in a transaction:
  - when data is written to and from disk;
  - when data is written to and from main memory.
- SQL details we'd like to avoid: data types, the particular SQL statement (e.g., **select** vs. **update**), the names of the relations, etc.
- Define the following commands of the read-write model:
  - **diskread**(x) - read block containing data item x from disk.
  - **diskwrite**(x) - write block containing data item x to disk.
  - **memread**(x) - read or use data item x from block in memory.
  - **memwrite**(x) - write into item x in memory.
  - **commit** - commit the current transaction.
  - **rollback** - rollback or cancel the current transaction.

  The idea is:

  - We only care about knowing whether a value is being changed in memory or whether it's changed on disk.
  - We don't really care whether the value is an int, float, char or whatever.
    ⇒ we will use algebraic labels like x.
- Consider the previous example:
  - Let the balance in Joe's billing tuple be called x.

– Let Joe's corporate balance in CORP_ACCOUNT be called y.

Let's identify corresponding read-write statements in the embedded SQL program:

```
EXEC SQL DECLARE SECTION
  varchar pass_name[50];      // Passenger name
  int cid;                    // Corporate id
  int cbalance;               // Corporate balance
  int amount;                 // Amount outstanding
EXEC SQL END DECLARE SECTION
// Read in passenger name from screen (not shown)
amount = 0;
EXEC SQL   SELECT B.CORP_ID, B.BALANCE  // diskread(x)
           INTO :cid, :amount           // memread(x)
           FROM BILLING B
           WHERE B.PNAME = :pass_name;
if (cid > 0) {
  EXEC SQL   UPDATE BILLING             // memwrite(x)
             SET BALANCE = 0            // diskwrite(x)
             WHERE PNAME = :pass_name;
  EXEC SQL   SELECT C.BALANCE           // diskread(y)
             INTO :cbalance             // memread(y)
             FROM CORP_ACCOUNTS C
             WHERE CORP_ID = :cid
  if (cbalance + amount <= 100000) {
    EXEC SQL   UPDATE CORP_ACCOUNTS       // memwrite(y)
               SET BALANCE = BALANCE + :amount
               WHERE CORP_ID = :cid;      // diskwrite(y)
    EXEC SQL   COMMIT;                    // commit
  }
  else {  // Cancel transaction
    EXEC SQL   ROLLBACK;                  // rollback
  }
 }
```

If we drop the SQL, we get the following program using the read-write model:

```
diskread($x$)                   // Get Joe's balance from disk
memread ($x$)                   // Read from memory
if (condition 1) {              // if cid>0
    memwrite($x$)                   // Zero out Joe's balance in memory
    diskwrite($x$)                  // Write Joe's block to disk
    diskread($y$)               // Get GM tuple
    if (condition 2) {          // Check total balance
        memread($y$)
        memwrite($y$)           // Write new GM balance in memory
        diskwrite($y$)          // Write it to disk

        commit
    }
    else {
        rollback
    }
}
```

Note: using *Deferred-Write* the **diskwrite**'s can be done later:

```
diskread($x$)                   // Get Joe's balance from disk
memread ($x$)                   // Read from memory
if (condition 1) {              // if cid>0
    memwrite($x$)                   // Zero out Joe's balance in memory
    diskread($y$)               // Get GM tuple
    memread($y$)
    if (condition 2) {          // Check total balance
        memwrite($y$)           // Write new GM balance in memory
        commit
        }
        else {
            **rollback**
        }
    **diskwrite**($x$)                     // Write Joe's block to disk
    **diskwrite**($y$)                     // Write GM's block to disk
    }
```

- Using the read-write model helps us discuss algorithms for recovery and concurrency without the clutter of SQL.

7.4 Recovery: The Problem

- *Recovery* refers to the problem of recovering from a failure, usually a system failure.

- Various kinds of failures:

  1. System crash: (hardware or operating system).
     - For example, an operating system may seize if too many processes are spawned.
  2. Logical errors, interrupts and traps, e.g.,
     - Division by zero.
     - Bugs in C (embedding SQL) that cause a core dump.
     - User-driven interrupts.
  3. Programmed rollbacks (when **rollback** is executed).
  4. Major system failures, e.g.,
     - Disk failures
     - Power outages.
     - Sabotage.

  Whatever the reason for the failure, we want to make sure the ACID properties hold for all transactions.

- The recovery problem: how to ensure that ACID properties hold in the presence of failures.

- In general,
  - For a transaction that commits, we want to ensure that changes to the database are made on disk.
  - For a transaction that does not, we want to ensure that it doesn't affect the database on disk.

7.5 Recovery: The Shadow Paging Method

- Note: in disk I/O jargon, a *block* is often called a *page*.
- Key ideas in shadow paging:
  - Bring data blocks into memory.
  - Make all writes to memory blocks first (**memwrite**'s).
  - Write modified blocks to new locations on disk.
  - Retain old versions (shadows) in their current locations on disk.
  - After new version is completely written to disk, mark old version as deleted.
  - A transaction is allowed to commit only after the new version is successfully written to disk.
  - If a failure occurs before or during writing the new version, then simply use old version upon recovery (reboot).
  - Thus, moments before committing, both old and new versions are on disk
    ⇒ At the moment of committing, simply choose between the two versions.
- Example:
  - Consider the following modifications to the FLIGHT relation:

```
UPDATE FLIGHT
SET FLT_NO = 'F12'
WHERE FLT_ID = 155;
UPDATE FLIGHT
SET FLT_NO = 'F63'
WHERE FLT_ID = 773;
COMMIT;
```

– Suppose the tuple for FLT_ID=155 has FLT_NO='F13' and the tuple for FLT_ID=773 has FLT_NO='F45'. Thus, the desired actions are:

Change 'F13' to 'F12'
Change 'F45' to 'F63'

– If x denotes the 155-tuple and y denotes the 773-tuple, then the read-write version can be written as:

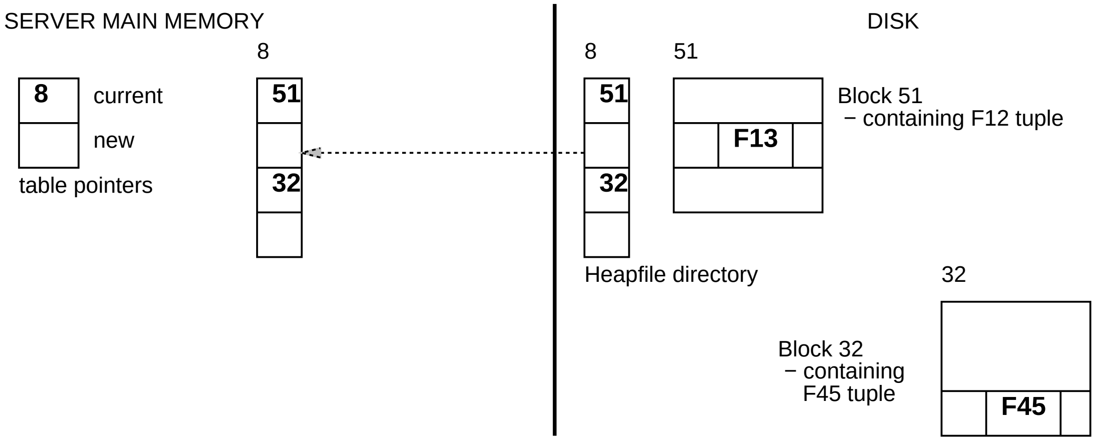
diskread(x)
memread(x)
memwrite(x)
diskread(y)
memread(y)
memwrite(y)
diskwrite(x)
diskwrite(y)

– Assume that FLIGHT is stored as a heapfile and that we have the block numbers for the two tuples of interest.

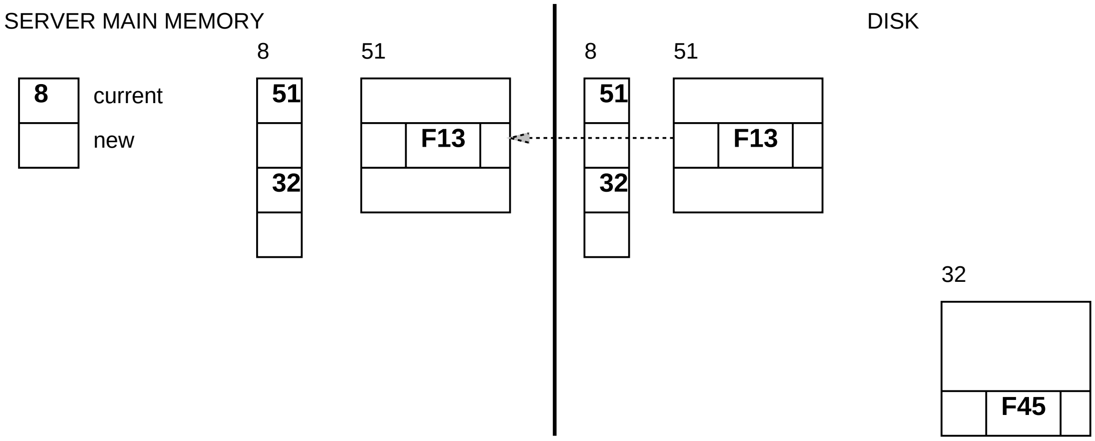
– We will not show client memory in this example.

– Let us now trace through the changes made using shadow paging:

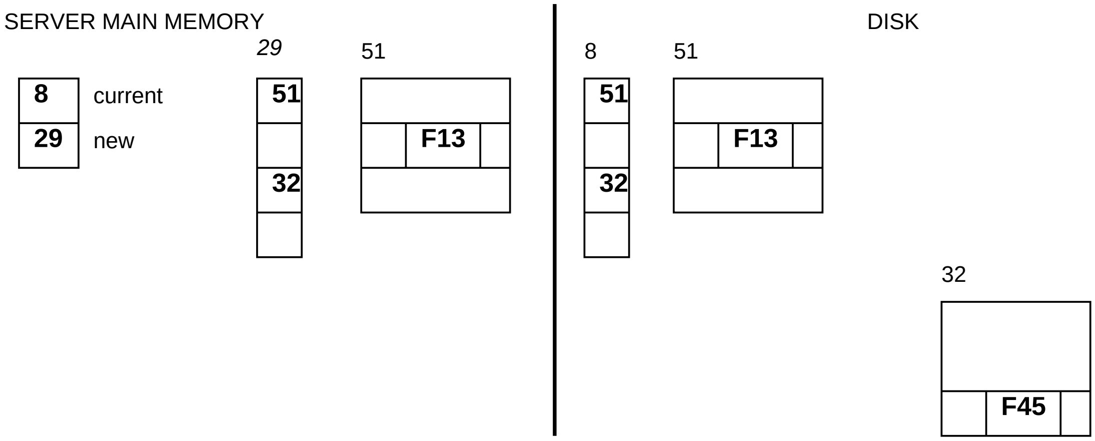
1. Initially, the blocks are on disk. The directory is read into memory.



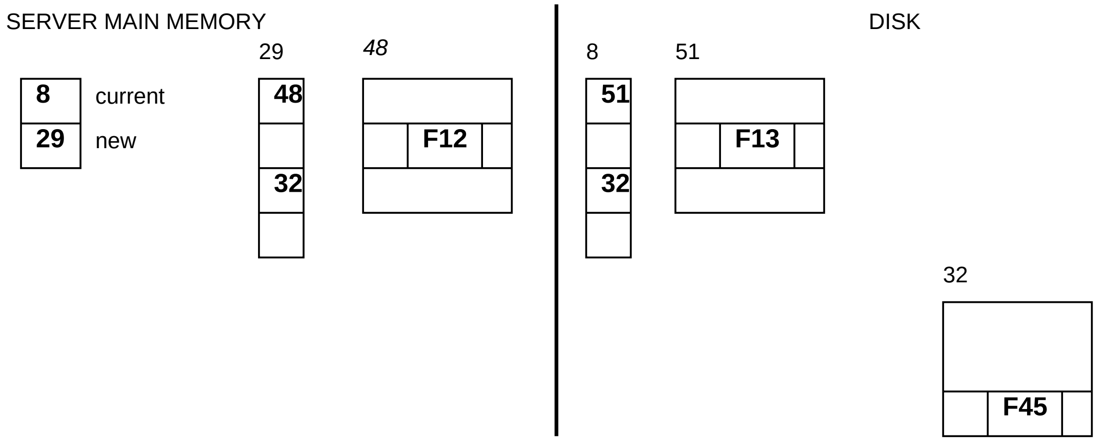
2. Bring block 51 into memory for an update:



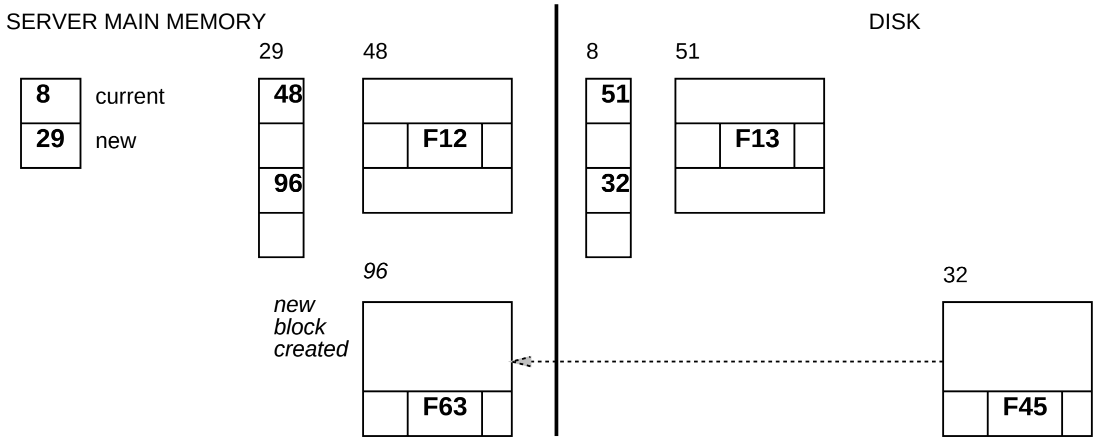
3. Copy existing directory into new directory



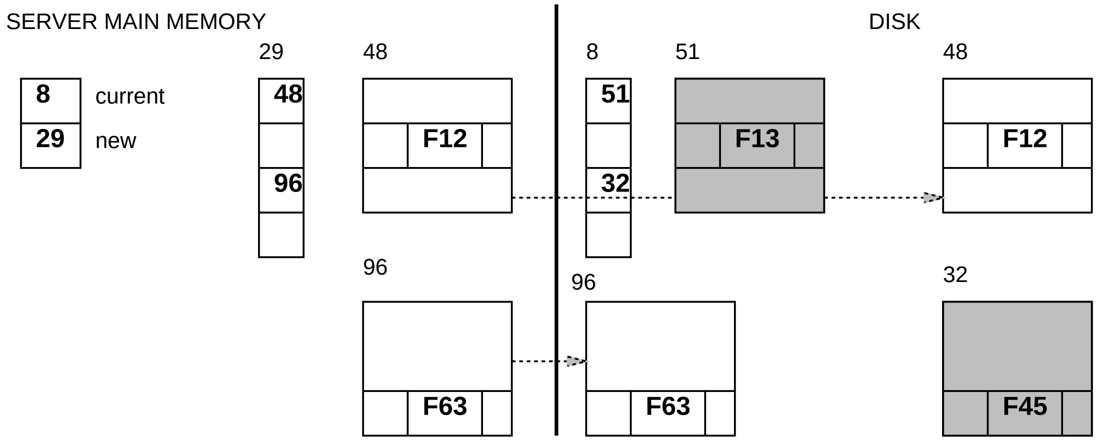
4. Create new block, copy old block data, and modify data:



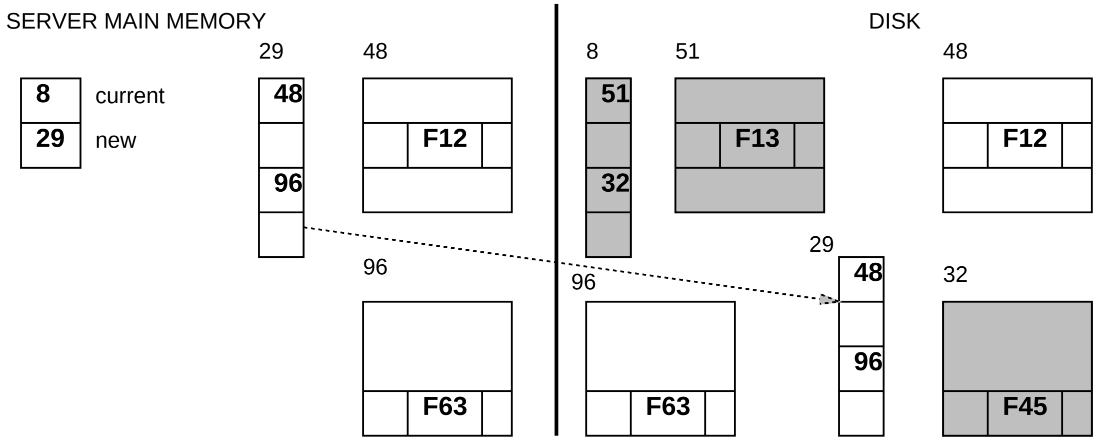
5. Similarly, read in block 32 (for the F45-tuple) and create new data block for second modification:



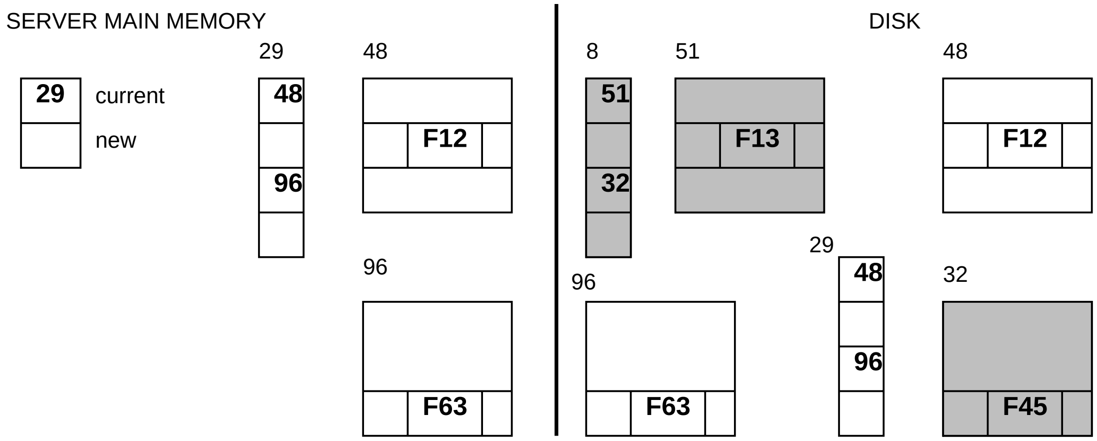
6. Next, write modifications to disk, while maintaining old blocks (shadows):



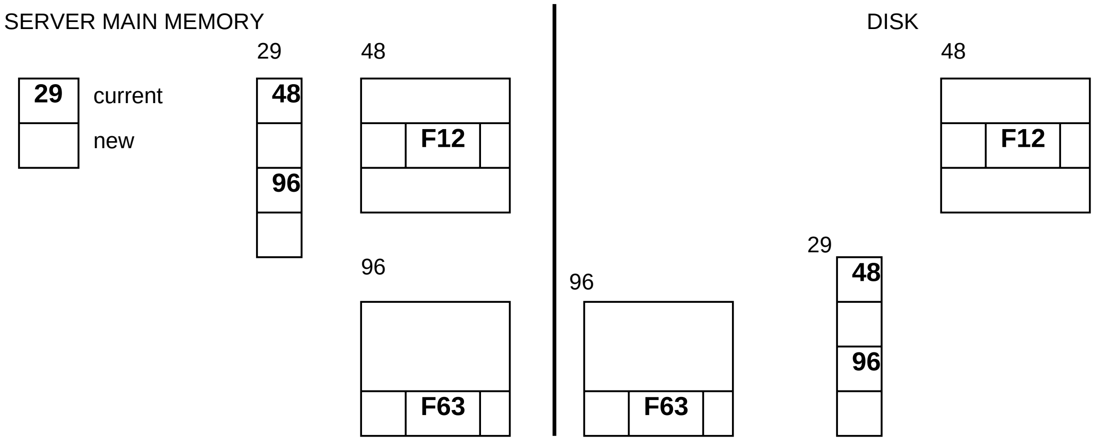
7. Write new directory to disk:



8. Set pointer to new directory and commit transaction:



9. Delete old blocks:

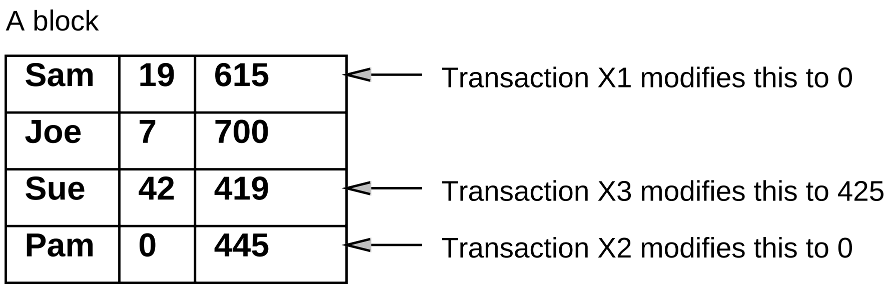


– What happens in case of a failure?

1. CASE 1: Failure occurs before any **diskwrite**'s take place
   - ∗ Thus, failure occurs before transaction commits
     ⇒ must not make changes to database.
   - ∗ Failure occurs before diskwrite's
     ⇒ no changes made
     ⇒ atomicity condition satisfied.
2. CASE 2: Failure occurs during **diskwrite**'s.
   - ∗ E.g., failure occurs just after block 48 is written but before block 96 is written.
   - ∗ Failure occurs before transaction commits
     ⇒ must preserve old data (no changes to database).
   - ∗ On reboot or recovery, old page table is retrieved
     ⇒ old version of data is preserved.
3. CASE 3: Failure occurs after blocks are written but before or during writing of directory.
   - ∗ Failure occurs before commit
     ⇒ must preserve old data.
   - ∗ Since current still points to old data
     ⇒ old directory table is used on reboot.
4. CASE 4: Failure occurs after commit.

- Since failure occurs after commit
  ⇒ must preserve changes to database.
- Upon recovery, new directory will be read
  ⇒ changes will be preserved.
- Note: deletion of old blocks can be done after recovery from failure (garbage collection).

- Problems with the shadow paging method:
  - It forces page-level concurrency:
    - Consider multiple transactions accessing the same block.
    - It's possible that several different transactions want to modify the same block, e.g.,



A block

Sam	**19**	**615**
Joe	**7**	**700**
Sue	**42**	**419**
Pam	**0**	**445**

Transaction X1 modifies this to 0

Transaction X3 modifies this to 425

Transaction X2 modifies this to 0

    - If the transactions run concurrently, but only some commit and others rollback, then one shadow and one new version is not enough.
    - Only option: allow only one transaction to modify a block
      ⇒ limits concurrency if others have to wait.
  - Shadow paging creates problems for indices like B-trees:
    - Recall: B-trees have datapointers into the heapfile.
    - If we're changing the heapfile
      ⇒ must modify corresponding datapointers in B-tree
      ⇒ must keep shadow of index
      ⇒ can be expensive to implement (lots of disk accesses).
  - Shadow paging can be inefficient:

    - Forcing all the **diskwrite**'s to occur together causes an I/O bottleneck.
    - Does not overlap CPU and I/O work.
    - If several transactions modify a block, page-level concurrency forces serialization
      ⇒ block is read and written once for each transaction's shadow paging.

- Shadow paging is rarely used; it serves as a benchmark for comparison and to better illustrate other methods.

7.6 Log-Based Recovery

- Recall the key idea in shadow paging: "We're doing OK as long as there's enough information on disk to recover from failure before starting **diskwrite**'s."
  - In shadow paging, the "information" was the fully-preserved old version itself.
  - In log-based recovery, the "information" is a log: a list of modifications made to the database.
- The meaning of "log" here is the same as in "Captain's Log."
- Key ideas in log-based recovery:
  - When a transaction does a **memwrite**(x), it puts the old value of x and the new value of x in a *log file* – the *log*.
  - The log is written to disk *before* the operation **diskwrite**(x) occurs.
  - If there's a failure during **diskwrite**(x):
    - ∗ We can **undo** changes to the database by recovering old values from the log.
    - ∗ We can **redo** the write by writing the new value from the log.
  - We make sure a transaction's log records are written to disk before it is allowed to commit.
- The log is usually itself a relation with several attributes, e.g.,

  LOG (TYPE, TRANS_ID, DATA_ITEM, OLD, NEW)

  where

  - TYPE is either "start", "data", "commit" or "rollback."
  - TRANS_ID indicates which transaction is logging the information.

- DATA_ITEM is the datapointer (blocknumber, tuple and offset) of the data tuple being modified.
- OLD is the old value that we're writing over.
- NEW is the new value to be written.

Thus, log entries are really tuples in the log relation.

- We will add operations to the *read-write model* to indicate log operations:

1. **memwrite_log**(<type>,<t_id>, <data_item>, <old-value>, <new-value>) – this function adds a record (log entry) to the log with the values indicated in the parameters.
   For example, the call

   memwrite_log('data', 117, x, 'F13', 'F12')

   causes the tuple

   <data, 117, x, F13, F12>

   to be written to the log.
2. **diskwrite_log** – this function causes the log to be written to disk.

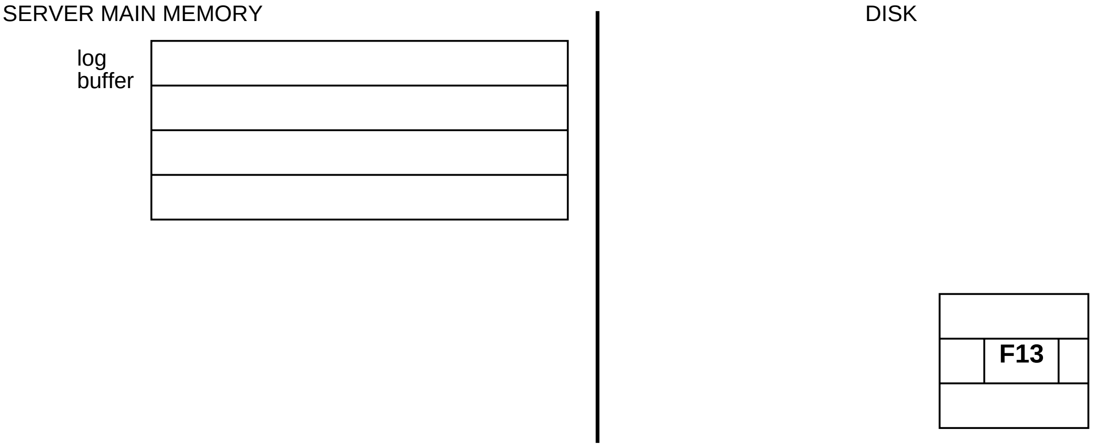
- The key ideas in the log-based recovery can now be explained:
  - Consider a transaction with TRANS_ID=117 that writes to the data item x and changes its value from 'F13' to 'F12'.
  - In the read-write model:

    1. **memwrite_log**('start', 117)
    2. **diskread**(x)
    3. **memread**(x)
    4. **memwrite**(x)
    5. **memwrite_log**('data',117,x,'F13','F12')
    6. **diskwrite_log**
    7. **diskwrite**(x)
    8. **memwrite_log**('commit',117)
    9. **diskwrite_log**

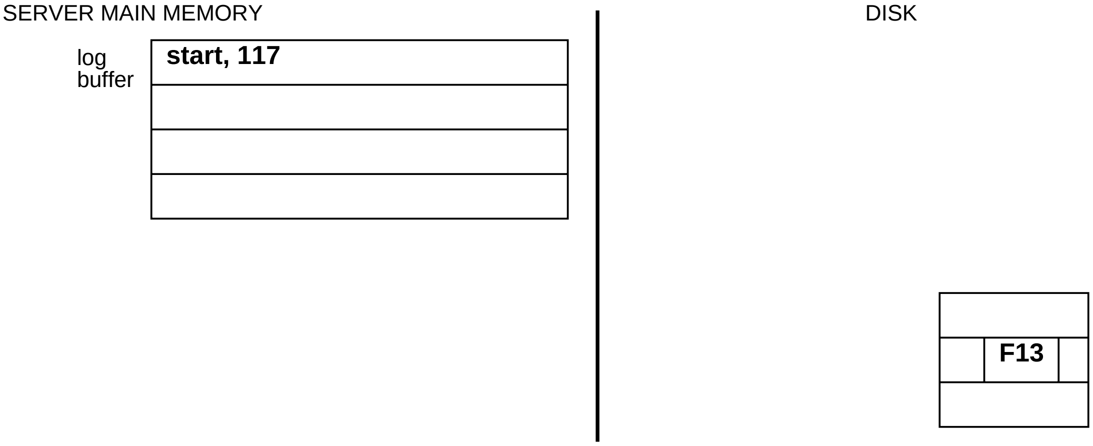
10. **commit**

– Let's illustrate the steps:

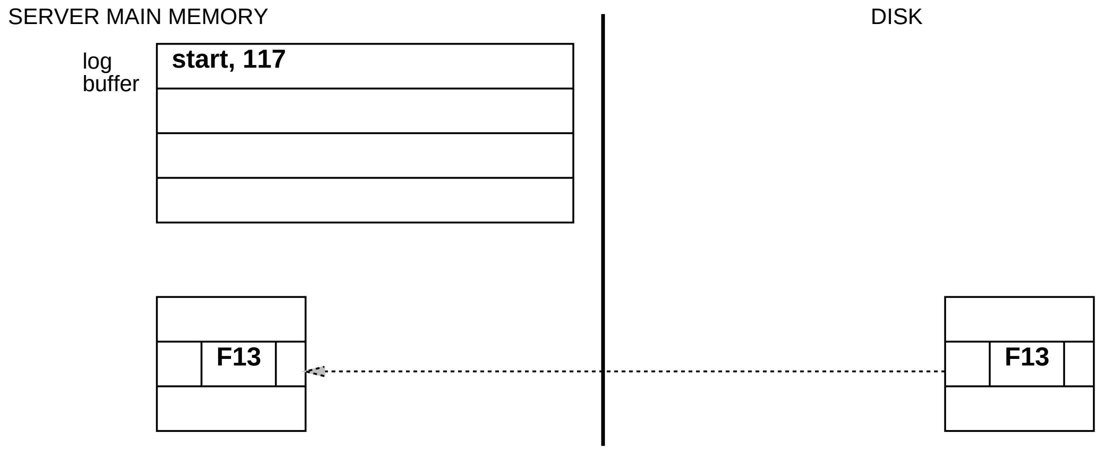
1. Initially:



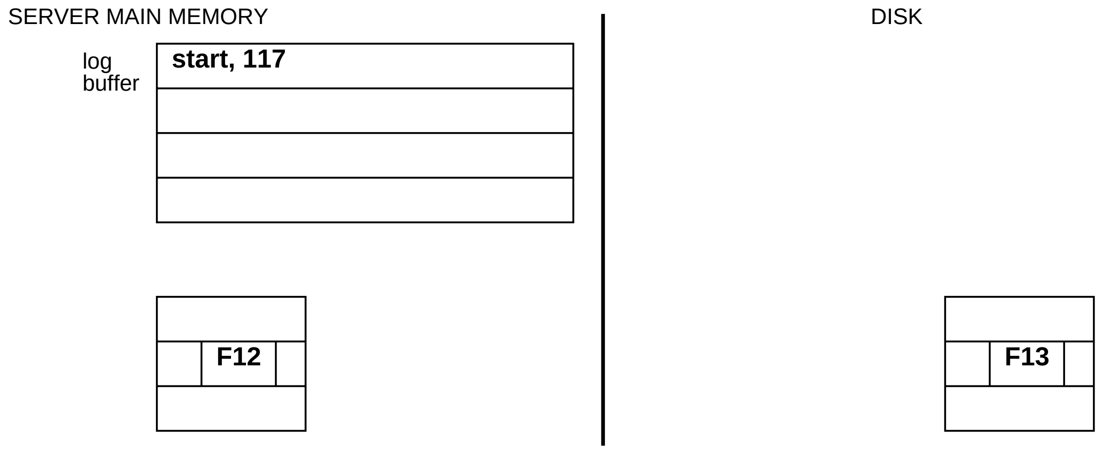
2. After **memwrite_log**('start',117):



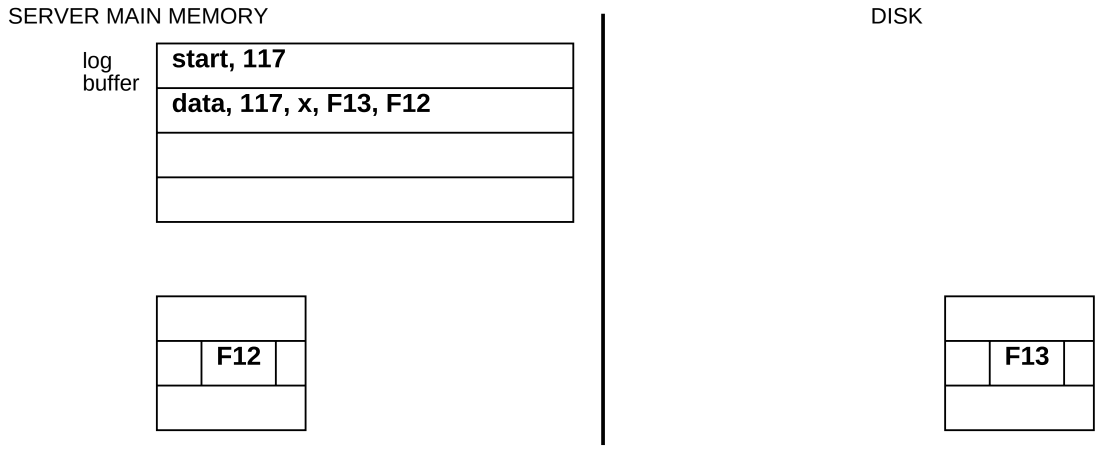
3. After **diskread**(x) and **memread**(x):



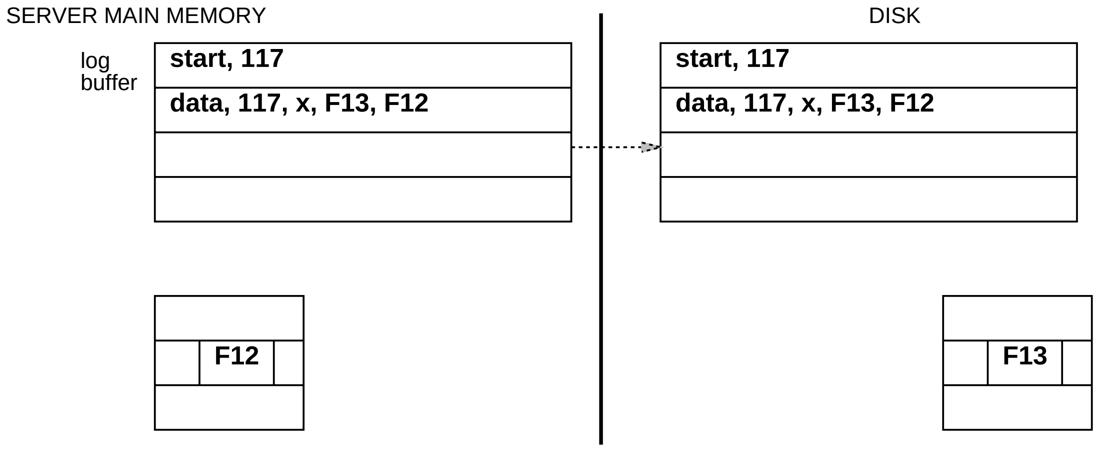
4. After **memwrite**(x):



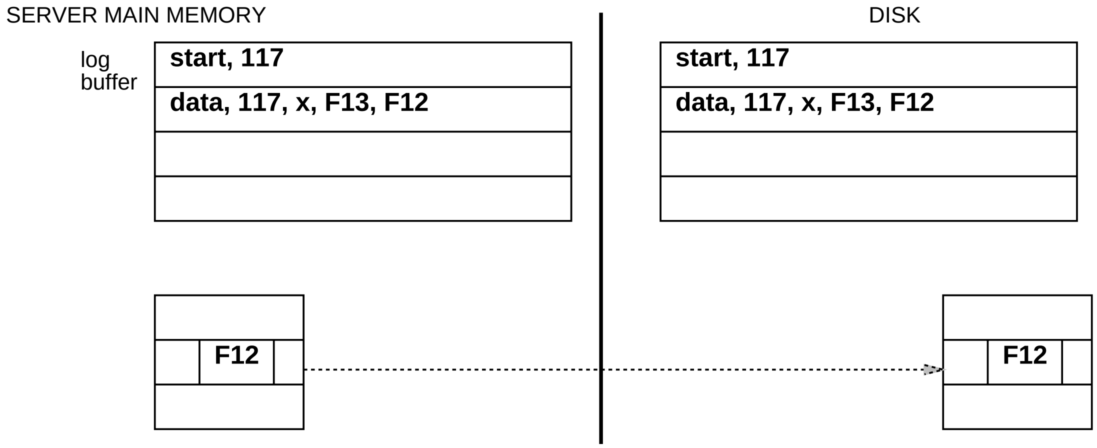
5. After **memwrite_log**(’data’,117,x,’F13’,’F12’):



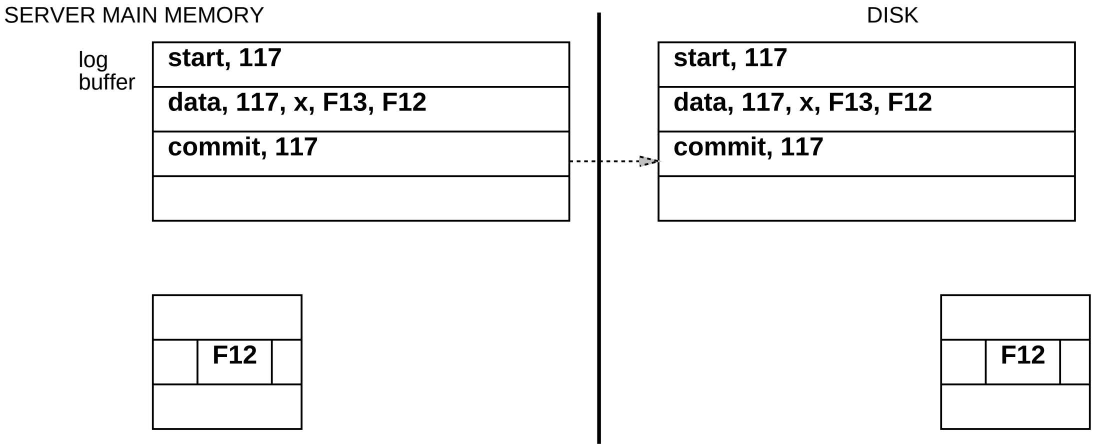
6. After **diskwrite_log** (which must precede **diskwrite**(x)):



7. After **diskwrite**(x):



8. After **memwrite_log**('commit') and **diskwrite_log**:



9. Finally, the transaction is allowed to commit.

– A failure can occur at any time during the execution of transaction 117:

1. Failure occurs during the execution of lines 1-5:
   ∗ No changes are made to the database
   ⇒ failure can be ignored (leave it to user to restart transaction).
2. Failure occurs after line 6.
   ∗ Log contains the records

   <'start', 117>
   <'data', 117, x, 'F13', 'F12'>

   ∗ On recovery, we know the intention was to write 'F12' over 'F13'.
   ∗ This could have happened during Line 7.
   ∗ However, transaction did not commit
   ⇒ must ensure old value remains in database
   ⇒ go to location x in data file and write 'F13' (old value).
3. Failure occurs after Line 9.
   ∗ Log contains the records:

   <'start', 117>
   <'data', 117, x, 'F13', 'F12'>
   <'commit', 117>

        - ∗ This could have happened only after data was written
          ⇒ changes already made
          ⇒ nothing to be done.

- Note: when a failure occurs, we won't know at which line of code it occurs
  - We will only be able to look at the log.
  - action to be taken depends on the state of the log:
    1. Log is empty
       ⇒ Failure could have occurred in lines 1-6
       ⇒ Do nothing.
    2. Log only contains

       <'start', 117>
       <'data', 117, x, 'F13', 'F12'>

       ⇒ Failure could have occurred in lines 7-9
       ⇒ undo changes made.
    3. Log contains

       <'start', 117>
       <'data', 117, x, 'F13', 'F12'>
       <'commit', 117>

       ⇒ Transaction commits.

7.7 Log-based Recovery: Writing Options

- Recall: a **diskwrite**(x) always occurs after **memwrite**(x).
- There are three ways in which **diskwrite**'s are usually handled:

  1. The *Immediate-Write* Method:
     - do a **diskwrite** right after the corresponding **memwrite**.
     - Example: consider a transaction that writes to items x and y

       ```
       diskread(x)          // Get 'F13' block
       memread(x)
       memwrite(x)          // Write 'F12'
       diskwrite(x)         // Write block to disk
       diskread(y)          // Get 'F45' block
       memread(y)
       memwrite(y)          // Write 'F63'
       diskwrite(y)         // Write block to disk
       commit               // Commit after all write's
       ```

     - Note: all **diskwrite**'s occur before a transaction commits.
  2. The *Deferred-Write* Method:
     - Always postpone all **diskwrite**'s for a transaction to after the transaction commits.
     - Example:

       ```
       diskread(x)          // Get 'F13' block
       memread(x)
       memwrite(x)          // Write 'F12'
       diskread(y)          // Get 'F45' block
       memread(y)
       memwrite(y)          // Write 'F63'
       commit               // Commit before write's
       ```

```
diskwrite(x)              // Write 'F12' block
diskwrite(y)              // Write 'F63' block
```

3. *Flexible-Write* Method:
   - Here, **diskwrite**'s may occur both before and after a transaction commits.
   - This method is the most flexible (it encompasses both Immediate-Write and Deferred-Write).
   - Example:

     ```
     diskread(x)
     memread(x)
     memwrite(x)
     diskread(y)
     memread(y)
     diskwrite(x)              // One write before commit
     memwrite(y)
     commit
     diskwrite(y)              // Another write after commit
     ```

   - Flexible-write's allow for better management of concurrency in I/O.

- **Rule**: A log entry for item x must written to disk *before* its corresponding **diskwrite** starts.

- There are three ways of permitting a transaction to commit with respect to when **diskwrite**'s are done:

  1. **R-NU**: a transaction is required to commit before its first **diskwrite** occurs.
  2. **U-NR**: a transaction may commit only after all its **diskwrite**'s are complete.
  3. **R-U**: a transaction may commit at any time, provided its log entries are first written to disk.

Note that some commit-methods may be used with some of the writing methods but not necessarily all of them:

	Commit Method		
Write Method	**R-NU**	**U-NR**	**R-U**
Immediate-Write	No	Yes	Yes
Deferred-Write	Yes	No	Yes
Flexible-Write	No	No	Yes

Why this is so will become clear when we consider recovery.

- Let's now consider the three commit-methods in some detail:

  1. **R-NU** [**Redo - No Undo**]

     – A transaction's **diskwrite**'s begin only after it commits
     ⇒ a Deferred-Write takes place.

     – Example:

     ```
     memwrite_log('start',117)
     diskread($x$)
     memread($x$)
     memwrite($x$)
     memwrite_log('data',117,x,'F13','F12')
     memwrite_log('commit',117)
     diskwrite_log // before commit
     commit
     diskwrite($x$) // deferred write
     ```

     – Upon recovery after failure, the log can be in one of three states:

     (a) Log is empty
     ⇒ nothing to be done.

     (b) Log contains

     <'start', 117>
     <'data', 117, x, 'F13', 'F12'>
     <'commit', 117>

⇒ transaction committed, but changes may not have been made

⇒ **redo** transaction.

- Note: no **undo** was needed.
- Note: if a transaction is long and many, many records are written to the log buffer in memory, it may be necessary to write parts of the log to disk (to make space)

  ⇒ Log entries may contain entries for uncommitted transactions

  ⇒ ignore these transactions since **diskwrite**'s did not begin.

2. **U-NR** [**Undo - No Redo**]

- In this case, a transaction's **diskwrite**'s must be done before it commits.
- Example:

```
memwrite_log('start')
diskread(x)
memread(x)
memwrite(x)
memwrite_log('data',117,x,'F13','F12')
diskwrite_log // before commit
diskwrite(x)
memwrite_log('commit',117)
diskwrite_log // before commit
commit
```

- Upon recovery after failure, the log can be in one of four states:

  (a) Log is empty

  ⇒ nothing to be done.

  (b) Log contains

  <'start', 117>

  ⇒ No **diskwrite**'s occurred

  ⇒ No action taken.

(c) Log contains

```
<'start', 117>
<'data', 117, x, 'F13', 'F12'>
```

⇒ A **diskwrite** may have occurred, but the transaction did not commit

⇒ must **undo** transaction

⇒ write old values back to data items (using log entries).

(d) Log contains

```
<'start', 117>
<'data', 117, x, 'F13', 'F12'>
<'commit', 117>
```

⇒ transaction committed

⇒ data got written (since commit occurs later)

⇒ nothing to be done.

– Note: no **redo** is needed.

3. **R-U** [**Redo-Undo**]

– A transaction may commit at any time, provided its log is written to disk first.

– Example: consider a transaction that modifies items x and y

```
memwrite_log('start', 118)
diskread(x)
memread(x)
memwrite(x)
memwrite_log('data', 118, x, 'F13', 'F12')
diskread(y)
memread(y)
diskwrite_log // Log for x must be written first
diskwrite(x) // x is written before commit
memwrite(y)
memwrite_log('data', 118, y, 'F45', 'F63')
memwrite_log('commit', 118)
diskwrite_log
commit
```

diskwrite(y) // y is written after commit

- Upon recovery from failure, the log can be in one of five states:
  (a) Log is empty
  ⇒ no action.
  (b) Log contains

  ```
  <'start', 118>
  <'data', 118, x, 'F13', 'F12'>
  ```

  ⇒ A write (of x) may have occurred but transaction didn't commit
  ⇒ **undo** effect by writing old value ('F13') to x.
  (c) Log contains

  ```
  <'start', 118>
  <'data', 118, x, 'F13', 'F12'>
  <'data', 118, y, 'F45', 'F63'>
  ```

  ⇒ Write's to x or y may have occurred without a commit
  ⇒ must **undo** effect by writing old values.
  (d) Log contains

  ```
  <'start', 118>
  <'data', 118, x, 'F13', 'F12'>
  <'data', 118, y, 'F45', 'F63'>
  <'commit', 118>
  ```

  ⇒ transaction committed, but all write's may not have completed
  ⇒ **redo** transaction.
  (e) Note: in the last case, we don't know which write's were successful and which weren't
  ⇒ must **redo** entire transaction.

- **Rollback**'s: sometimes a transaction issues a **rollback**
  ⇒ a log entry of the type <'rollback', 118> is written.

  Example: consider the earlier example of transferring balance from BILLING to CORP_ACCOUNT.

– Consider the read-write version of the program.

– We will add log-based recovery to this code.

– We will use the **redo - no undo** scheme.

– Suppose transaction id is 243.

```
memwrite_log(’start’, 243)
diskread(x)                    // Get Joe’s balance from disk
memread (x)                    // Read from memory
if (condition 1) {             // if cid>0
    memwrite(x)                    // Zero out Joe’s balance in memory
    memwrite_log(’data’, 243, x, ...)
    diskread(y)                // Get GM tuple
    memread(y)
    if (condition 2) {         // Check total balance
        memwrite(y)            // Write new GM balance in memory
        memwrite_log(’data’, 243, y, ...)
        memwrite_log(’commit’, 243)
        diskwrite_log
        commit
    }
    else {
        memwrite_log(’rollback’, 243)
        diskwrite_log
        rollback
    }
diskwrite(x)                   // Write Joe’s block to disk
diskwrite(y)                   // Write GM’s block to disk
}
```

Thus, for example, upon recovery from failure the log may contain

```
<’start’, 243>
<’data’, 243, x, ...>
<’rollback’, 243>
```

In this case, no action is needed since write's occur after commit and here, the transaction didn't commit.

7.8 Log-Based Recovery: Summary and Odds & Ends

- A summary of the rules used in recovery:
  - The log entry <'data',...,x,...> must be written to the log buffer when **memwrite**(x) occurs.
  - All log entries up to the most recent one for x are written to disk *before* a **diskwrite**(x) occurs.
  - All log entries pertaining to a transaction are written to disk *before* it commits.
  - Three ways of organizing the logging/disk-writing process:
    1. **Redo - No Undo**:
       - Make a transaction's **diskwrite**'s occur only after it commits.
       - Upon recovery:
         - **Redo** committed transactions.
         - Ignore uncommitted or rolledback transactions.
    2. **Undo - No Redo**:
       - Allow a transaction to commit only after its **diskwrite**'s are completed.
       - Upon recovery:
         - **Undo** uncommitted or rolledback transactions.
         - Ignore committed transactions.
    3. **Redo - Undo**:
       - Allow **diskwrite**'s to occur before and after commit.
       - Upon recovery:
         - **Redo** committed transactions.
         - **Undo** uncommitted or rolledback transactions.

- **Redo** followed by **Undo**:
  - In some systems that use **Redo-Undo**, the following is done upon recovery:
    - First, **redo** all transactions.
    - Then, **undo** uncommitted or rolledback transactions.
  - Why? If *all* transactions are **redo**ne, then the system is brought to the same state it was in before the crash.

    ⇒ if software problems caused the crash, the crash could be made to repeat

    ⇒ useful in learning what caused the crash.

- Failures during recovery:
  - What if there's another failure right in the middle of of the recovery process?
  - Recovery must be designed so that it can be re-started at will.
  - Since both the old and new values are stored for each modified data item, we can always both **redo** and **undo** after multiple failures during recovery.
- In practice:
  - Most systems use **Redo-Undo** since it is the most flexible scheme.
  - Writing of data can be done independently

    ⇒ CPU and I/O overlap can be arranged.
  - Most systems have separate processes for writing data and logging

    ⇒ these processes communicate to synchronize.
- Checkpointing:
  - Suppose 10,000 transactions have executed when we get a failure and suppose, of these, 9807 transactions committed.

    ⇒ **redo** required for 9807 transactions and **undo** required for 193 transactions.

  - Recovery can take longer if number of transactions is higher (e.g., 10^9).
  - To reduce the work in recovery, *checkpointing* is used.
  - Key ideas in checkpointing:
    - Old transactions which were successfully run long ago shouldn't have to be **redo**ne.
    - Introduce checkpoints at regular intervals.
    - For each checkpoint:
      - Hold off starting the execution of new transactions.
      - Let current transactions run to completion.
      - Write all log entries to disk.
      - Write all data blocks in memory to disk.
      - Write a <'checkpoint'> entry to the log and write that entry to disk.
      - Resume normal operation.
    - Then, upon recovery, we don't have to worry about log entries prior to the most recent checkpoint.
    - Example:

      ```
      <'start', 117>
      <'data', 117, ...>
      <'commit', 117>
      <'checkpoint'>
      <'start', 243>
      <'data', 243, ...>
      ```

      Here, only transactions after the checkpoint are examined for **redo** or **undo**.

- Safety in numbers:
  - To increase reliability, many systems maintain a copy of the log.
  - While one copy is being updated, the other is archived to tape or some other device.

7.9 Concurrency Control: Introduction

- What do we mean by "concurrency"?
  - We mean: executing transactions concurrently.
  - Think of it as concurrently running several embedded SQL programs (perhaps from different users).
  - Concurrent execution can mean
    - ∗ executing programs on different processors of a multiprocessor, or
    - ∗ interleaving the execution of different programs on a single processor.
  - In either case, shared data is accessed and access to shared data is *interleaved*.
  - Example: recall the relations CORP_ACCOUNTS and BILLING:

CORP_ACCOUNTS	CORP_ID	CNAME	BALANCE
	3	IBM	649,314
	19	Intel	213,617
	7	GM	65,973
	42	DuPont	143,112

BILLING	PNAME	CORP_ID	BALANCE
	Sam	19	615
	Joe	7	700
	Sue	42	419
	Pam	0	445

  BILLING contains the outstanding balance for individual passengers. CORP_ACCOUNTS contains the amount owed by some corporate clients.

  - Consider the following interleaving of transactions N_A and N_B (only SQL shown):

N_A	N_B

	// Read in Joe’s balance, corp.id
	SELECT BALANCE, CORPID
	INTO :amount, :cid
	FROM BILLING
	WHERE PNAME = ’Joe’;

// Read GM’s balance
SELECT BALANCE
INTO :cbalance
FROM CORP_ACCOUNTS
WHERE CNAME = ’GM’;	

	// Add Joe’s balance to GM’s
	UPDATE CORP_ACCOUNTS
	SET BALANCE = BALANCE + :amount
	WHERE CORP_ID = :cid;

// Find out which companies have
// as much or less balance
SELECT COUNT(*)
INTO :num
FROM CORP_ACCOUNTS
WHERE BALANCE ≤ :cbalance;	

- In practice, interleaving occurs at the machine-instruction level.

- Why permit concurrency?
  - If all transactions are of short duration (e.g., 1 ms to 10^3 ms), forcing serialization is feasible.
  - Consider serializing long and short transactions, e.g.,

N_A	N_B

// Obtain average mileage
SELECT AVG(MILES)
INTO :avg
FROM PASSENGER;	

	SELECT MILES
	INTO :miles
	FROM PASSENGER
	WHERE NAME = ’Joe’ ;

In the serial order $N_A \rightarrow N_B$, N_B waits a long time for N_A to complete.

  - Both can be interleaved, allowing N_B to complete early, with little noticeable increase in the time to complete N_A.

- What about guaranteeing *Isolation* (the **I** in the "ACID" properties)?

  - A transaction must be given the illusion that, once it starts execution, it alone reads or modifies the data it accesses until it's done.
  - Thus, for example, given a concurrent execution of transactions N_A and N_B, then after the concurrent execution, the data should be in the form it would be in if either serial execution $N_A \rightarrow N_B$ or $N_B \rightarrow N_A$ took place.

- An *execution history* of a concurrent execution is the particular interleaving of instructions that occurs for a particular concurrent execution of a set of transactions.

- A concurrent execution history satisfies *isolation* if it results in changes to the data equivalent to changes wrought by some serial execution of the transactions involved.

- NOTE: we could define *isolation* to mean "same order as order of arrival" but that would severely limit concurrency (see above example).

- Clearly, for a group of transactions that don't modify the data, *any* execution history will do the job (satisfies isolation).

- Also, transactions that modify disparate data can run concurrently in any way, e.g.,

N_A	N_B

`// 10% discount for passengers`
`UPDATE BILLINGS`
`SET BALANCE = BALANCE * 0.9`	

	`// 1% discount for corp's`
	`UPDATE CORP_ACCOUNTS`
	`SET BALANCE = BALANCE * 0.99`

- Consider the following interleaving:

N_A	N_B
Initially, say, Joe has $700	

`// Count passengers in BILLINGS`
`SELECT COUNT(*)`
`INTO :num`
`FROM BILLINGS`	

	`// 10% discount for passengers`
	`UPDATE BILLINGS`
	`SET BALANCE = BALANCE * 0.9`

`// Find Joe's balance`
`SELECT BALANCE`
`INTO :amount`
`WHERE PNAME ='Joe'`	

Output by N_A: 700*0.9=630.	

Is this execution history equivalent to some serial history?

Yes. It is equivalent to:

N_A	N_B
Initially, Joe has $700	

	// 10% discount for passengers
	UPDATE BILLINGS
	SET BALANCE = BALANCE * 0.9

// Count passengers in BILLINGS
SELECT COUNT(*)
INTO :num
FROM BILLINGS
 	
// Find Joe's balance
SELECT BALANCE
INTO :amount
WHERE PNAME ='Joe'	

Output by N_A: 700*0.9=630.	

It is NOT equivalent to:

N_A	N_B
Initially, Joe has $700	

// Count passengers in BILLINGS
SELECT COUNT(*)
INTO :num
FROM BILLINGS
 	
// Find Joe's balance
SELECT BALANCE
INTO :amount
WHERE PNAME ='Joe'	

	// 10% discount for passengers
	UPDATE BILLINGS
	SET BALANCE = BALANCE * 0.9

Output for N_A: $700	

Note: isolation property is satisfied since it is equivalent to at least one serial ordering.

- Q: Is it possible that every concurrent execution history is equivalent to some serial history?
  Ans: No.

- Before we describe problems with concurrency, let's revisit the Read-Write model for purposes of analysing concurrency problems:
  - From the point of view of concurrency, we don't care whether data is modified in memory or disk.
  - If data is modified in memory, the *recovery system* will make sure it will be modified on disk.
    ⇒ we don't distinguish between **diskwrite**(x) and **memwrite**(x)
    ⇒ simply use **write**(x) to denote both.
  - Similarly, we will use **read**(x) to denote "reading some item x."
  - In the code examples that follow, the read-write commands will be indicated as comments.

- Problems with un-controlled concurrent execution:

  Arbitrarily executing transactions concurrently can create strange behavior.

  1. The Case of the Unrepeatable Read:

N_A	N_B

	`// Read in Joe's balance, corp.id`
	`SELECT BALANCE, CORP_ID`
	`INTO :amount, :cid`
	`FROM BILLING`
	`WHERE PNAME = 'Joe';`

`// Read GM's balance`
`SELECT BALANCE //` **read**(x)
`INTO :cbalance`
`FROM CORP_ACCOUNTS`
`WHERE CNAME = 'GM';`	

	`// Add Joe's to GM's balance`
	`UPDATE CORP_ACCOUNTS //` **write**(x)
	`SET BALANCE = BALANCE + :amount`
	`WHERE CORP_ID = :cid;`

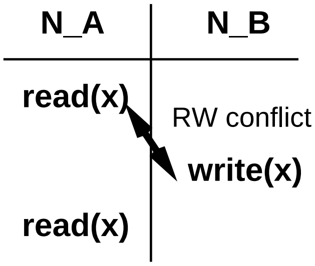
`// Companies having ≤ GM's balance`
`SELECT COUNT(*) //` **read**(x)
`INTO :num`
`FROM CORP_ACCOUNTS`
`WHERE BALANCE ≤ :cbalance;`	

N_A's output doesn't include GM!	

     - In either serial execution ($N_A \rightarrow N_B$ or $N_B \rightarrow N_A$), GM will be counted among corporations that have balance less than or equal to GM's balance.
     - However, in the above interleaving, it won't be counted
       ⇒ execution history is not serializable.
     - In the read-write model:

N_A	N_B
`// Read GM’s balance` **read**(x)	
	`// Write into GM’s balance` **write**(x)
`// Read GM’s balance again` **read**(x)	
Second read is not the same as the first!	

- Thus, the history does not satisfy the isolation property for N_A.
- From N_A’s point of view: “I’m re-reading the same data and it’s different!”
- The problem of *unrepeatable reads* is sometimes called a *Read-Write conflict*:



2. The Case of the Dirty Read.

N_A	N_B

// Read Joe's balance, corp.id
SELECT BALANCE, CORP_ID
INTO :amount, :cid
FROM BILLING
WHERE PNAME = 'Joe'
// Set Joe's balance to 0
UPDATE BILLING // **write**(x)
SET BALANCE = 0
WHERE PNAME = 'Joe';	

	// Count # people with 0-balance
	SELECT COUNT(*) // **read**(x)
	FROM BILLING
	WHERE BALANCE = 0;

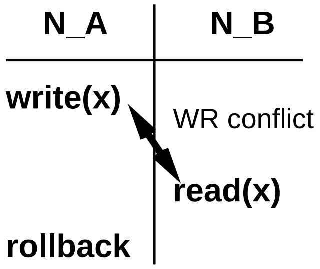
// Transfer balance, if allowed
SELECT BALANCE
INTO :cbalance
FROM CORP_ACCOUNTS
WHERE CORP_ID = :cid;
if (:amount + :cbalance $\leq$ 100,000)
{
   UPDATE CORP_ACCOUNTS
   SET BALANCE = BALANCE + :amount
   WHERE CORP_ID = :cid;
   COMMIT;
}
else {
   ROLLBACK // **rollback**
}	

	Joe gets counted

– The problem is easily seen in the read-write model:

N_A	N_B
// Set Joe's balance to 0 **write**(x)	
	// Read Joe's balance as 0 **read**(x)
// Did not commit **rollback**	

- N_B reads a value written by an uncommitted transaction.
- In either serialization, this would not happen
  ⇒ does not satisfy isolation property.
- The problem of reading an uncommitted write, the *Dirty Read* problem, is also called a *Write-Read conflict*:



3. The Case of the Lost Update:

N_A	N_B

// Read Joe's balance, corp.id
SELECT BALANCE, CORP_ID
INTO :amount, :cid
WHERE PNAME = 'Joe';	

	// 10% discount to all passengers
	UPDATE BILLING // **write**(x)
	SET BALANCE = BALANCE * 0.9

// Set Joe's balance to 0
UPDATE BILLING // **write**(x)
SET BALANCE = 0
WHERE PNAME = 'Joe';	

// Transfer balance, if allowed
SELECT BALANCE
INTO :cbalance
FROM CORP_ACCOUNTS
WHERE CORP_ID = :cid;
if (:amount + :cbalance ≤ 100,000)
{
   UPDATE CORP_ACCOUNTS
   SET BALANCE = BALANCE + :amount
   WHERE CORP_ID = :cid;
   COMMIT;
}
else {
   ROLLBACK // **rollback**
}	

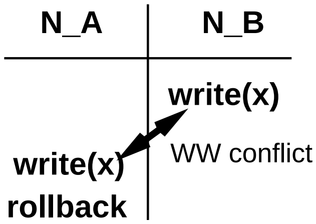
– The problem is easily seen in the read-write model:

N_A	N_B
	`// Joe gets 10% discount` **write**(x)
`// Joe's balance is set to 0` **write**(x)	
rollback	

– If N_A rolls back, N_B's update is lost.

– In either serial execution N_B's update would have worked
⇒ isolation property not satisfied.

– The lost update problem is sometimes called a *Write-Write conflict*:



- To summarize:
  - There are problems when one transaction writes into a data item (**write**(x)) and another reads (**read**(x)) or writes (another **write**(x)) into the same item.
  - Three types of conflicts: **RW** (Read-Write), **WR** (Write-Read) and **WW** (Write-Write).

    Note: **RR** (Read-Read) causes no problem.
  - The three problems we saw (Unrepeatable Read, Uncommitted-Write Read, Lost Update) are examples what can go wrong when these conflicts occur.
  - These are the only types of conflicts we need to analyze for the purposes of concurrency.

7.10 Concurrency Control: Conflict Serializability

- Note:
  - In our examples, we have interleaved at a high-level (alternating SQL statements).
  - In practice, interleaving occurs at the machine instruction level.
  - However, from a data item's point of view, access (**read** or **write**) is strictly interleaved.
    ⇒ we will consider the read-write model in the sequel.
- Sometimes, an execution history is called a *schedule*.
  Usually,
  - *execution history* is used in the past tense (as in comparing histories of transactions that already executed concurrently).
  - *schedule* is used to discuss construction of a future interleaving of a group of transactions.
- **Definition**: Two particular instances of **read** or **write** operations are *conflicting* if
  1. they belong to different transactions;
  2. they operate on the same data item; and,
  3. at least one of them is a **write**.
- **Definition**: The execution order of two steps of a concurrent schedule can be interchanged or *swapped* if
  1. the steps are consecutive in the schedule;
  2. each step involves a different transaction; and,
  3. the two operations (in the two steps) are *non-conflicting*.

- Example:

	N_A	N_B	N_C
1.			**read**(x)
2.	**read**(x)		
3.		**read**(y)	
4.		**write**(y)	
5.	**read**(z)		
6.			**write**(z)
7.	**write**(z)		

  – Steps 1 and 2 can be swapped (both are reads).

  – Steps 4 and 5 can be swapped (they read or write different items).

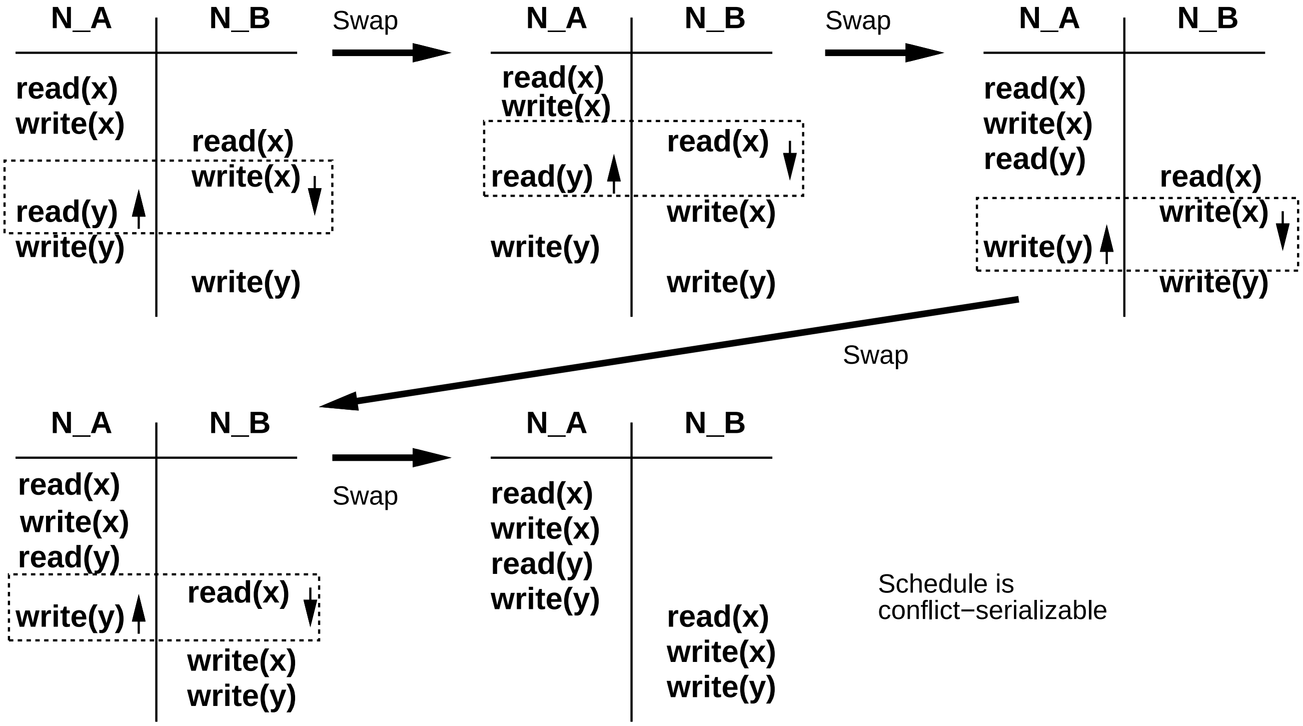
  – Steps 3 and 4 cannot be swapped (same transaction).

  – Steps 5 and 6 cannot be swapped (operations conflict).

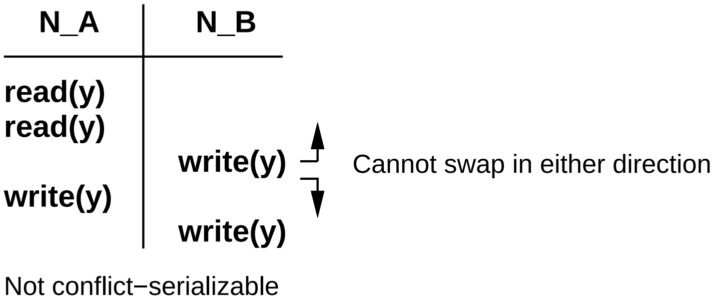
- **Definition**: A schedule (or history) is *conflict-serializable* if it can be transformed by a series of swaps into a serial schedule.

- Key idea: swaps don't affect the outcome
  ⇒ outcome is same as a serial schedule
  ⇒ isolation property will be satisfied.

- Example (conflict-serializable schedule):



- Example (non-conflict-serializable schedule):



Note, however, that the last **write** overwrites all previous y values.
$\Rightarrow$ the schedule is equivalent to

N_A	N_B
read(y)	
read(y)	
write(y)	
	write(y)
	write(y) ← Overwrites previous values

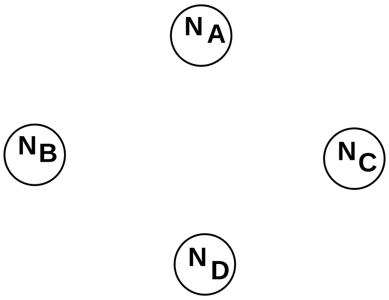
Thus, a non-conflict-serializable schedule may still be serializable.

Nonetheless, conflict-serializability at least guarantees isolation (if the transactions commit).

- Testing for conflict-serializability by trying all possible swaps is too inefficient,
  e.g. with 100 transactions
  ⇒ too many serial orders to consider.
- The Precedence Graph Method:
  - Step 1: Given transactions $N_1, N_2, \ldots, N_k$, and a schedule, construct the following graph:
    - ∗ Step 1A: Represent each transaction by a vertex.
      Example:

	N_A	N_B	N_C	N_D
1.	**read**(x)			
2.			**write**(x)	
3.		**write**(x)		
4.		**write**(y)		
5.			**read**(z)	
6.		**read**(z)		
7.				**write**(z)

Then, the vertices are:

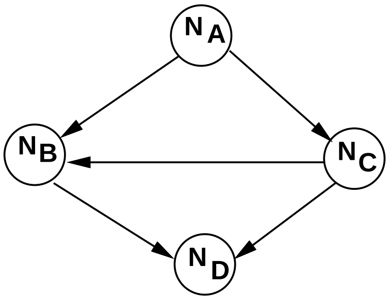


- Step 1B: Draw an edge from N_i to N_j if any of the following is true:

  1. N_i executes a **write**(x) before N_j executes a **read**(x) (for some x).
  2. N_i executes a **read**(x) before N_j executes a **write**(x) (for some x).
  3. N_i executes a **write**(x) before N_j executes a **write**(x) (for some x).

  Example:

  - *Add edges from N_A to N_B and N_C because the* **read**(x) *in line 1 conflicts with the* **write***'s in lines 2 and 3.*
  - *Add an edge from N_C to N_B because the* **write** *in line 2 (N_C) conflicts with the* **write** *in line 3 (N_B).*
  - *Add an edge from N_B to N_D because the* **read** *in line 6 (N_B) conflicts with the* **write** *in line 7 (N_D).*
  - *Similarly, add an edge from N_C to N_D because the the* **read** *in line 5 conflicts with the* **write** *in line 7.*



– Step 2: Check to see if the graph has a cycle (a traversal along edges in which a vertex is repeated).

Example: *No cycles.*

- Step 3: If it has no cycles, it is conflict-serializable; otherwise it is not.
  Example: *The schedule is conflict-serializable.*

Why does this work? Here's the key idea:

- An edge from N_i to N_j indicates that at least one of N_j's instructions (**read** or **write**) cannot be swapped to before at least one of N_j's instructions.
  $\Rightarrow N_j$ must appear after N_i in a serialization created by swaps.

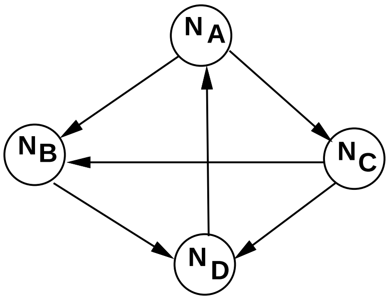
- Consider a cycle like $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_1$.
- This says "N_1 must appear after itself".
  $\Rightarrow$ a contradiction!
- Thus, cycle $\Rightarrow$ not conflict-serializable.
- Finally, if there are no cycles, a serialization is possible, using a *topological sort*.

- Example of a schedule that's not conflict-serializable:

	N_A	N_B	N_C	N_D
1.	**read**(x)			
2.			**write**(x)	
3.		**write**(x)		
4.		**write**(y)		
5.			**read**(z)	
6.		**read**(z)		
7.				**write**(z)
8.	**read**(z)			

(This example is the last one with one additional line – line 8.)

The precedence graph now has an edge from N_D to N_A:



There is a cycle: $N_A \to N_B \to N_D \to N_A$
$\Rightarrow$ not conflict-serializable.

- Recall: we considered precedence graphs because trying swaps was too inefficient.
  - Is the precedence graph method efficient?
  - If the graph has E edges, it can be built in time $O(E)$.
  - Cycle-testing can be done in time $O(E)$.
- Problems with testing for conflict-serializability:
  - In practice, we cannot obtain schedules in advance (since scheduling is usually done by the operating system).
  - Another problem: not all conflicts are covered, e.g.,

N_A	N_B
	write(x)
read(x)	
commit	
	rollback

    * Here, N_A reads a value of x written by a transaction that later fails to commit
      $\Rightarrow$ the problem of reading an uncommitted-write.
    * Yet, this schedule is conflict-serializable (can be swapped to $N_B \to N_A$).
    * Conflict-serializability is only guaranteed to work if the transactions commit
      $\Rightarrow$ not practical, since we can't predict commits.

Nonetheless, conflict-serializability is a useful theoretical tool that is used in proving that other (more practical) schemes work.

- Note: in the above example, if N_B rolls back, we really need to roll back N_A (because it read N_B's writes)
  ⇒ a cascading rollback.

- **Definition**: A schedule is *recoverable* if at the time of a transaction's commit, the transactions that wrote into data it has read have committed.

N_A	N_B
write(x)	
	read(x)
commit	
	commit

Recoverable

N_A	N_B
write(x)	
	read(x)
	commit
commit	

Not recoverable

  - Thus, for example, if N_A writes into N_B's data and N_B is about to commit, then if N_A has not committed the schedule is *not* recoverable.
  - If N_A rolls back, N_B will have to be rolled back as well.

- **Definition**: A schedule is *strict* if at the time of a transaction's read or write, any other transaction that wrote into the data item has already committed.

N_A	N_B
write(x)	
	read(x)
commit	
	commit

Not strict

N_A	N_B
write(x)	
commit	
	read(x)
	commit

Strict

  - For example, if N_B reads data written into by N_A, N_A has already committed
    ⇒ the data written was safe.

- Thus, a strict schedule is recoverable but not vice versa.
- However, a recoverable schedule can allow for greater concurrency.
- Note: strict schedules don't incur cascading rollback's.
- In practice, strict schedules are used (even though they are less flexible).

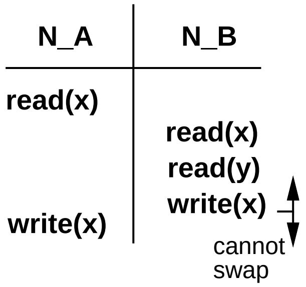
7.11 Concurrency Control Via Locking

- In practice, transaction schedules cannot be constructed ahead of execution time to ensure serializability.
  - Transactions are executed as *processes* or *threads*.
  - Operating System (OS) does the interleaving
    ⇒ OS might create a non-serializable history.
- A different (and practical) approach to serializability:
  - Allow OS to perform scheduling.
  - Introduce safeguards so that OS schedule is always serializable.
- Observe:
  - Lack of serializability occurs because of Read-Write, Write-Read or Write-Write conflicts.
  - Thus, we can try to prevent these conflicts.
- Key ideas:
  - Use *locks* on data items.
  - A transaction that wants to use item x must lock x
    ⇒ No one else can touch x until the transaction *unlocks* x.
  - Transactions that encounter a locked item simply wait until item is unlocked.

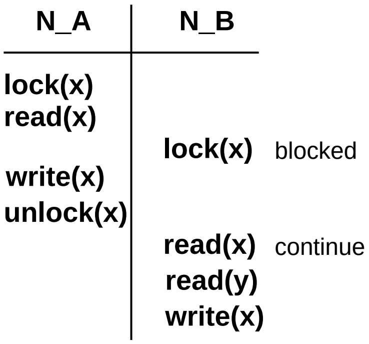
  Thus, locking attempts to prevent conflicts.
- In the read-write model, we introduce the following commands:
  - **lock**(x) – put a lock on data item x.

- – **unlock**(x) – unlock data item x (provided a lock on it was acquired earlier).

- Example:

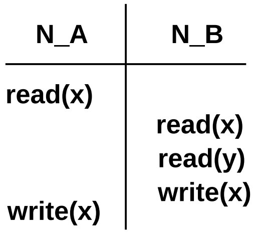


Not serializable

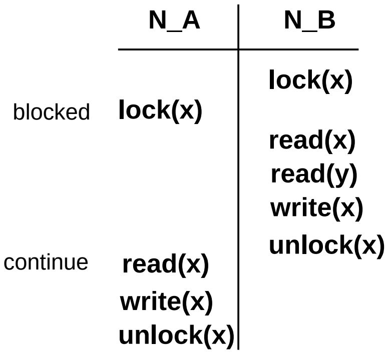


Serialization via locking

Here, N_A managed to acquire the lock on x first. If N_B acquired it first:



Not serializable



Serialization via locking

- Read and write locks:
  - – As currently defined, locks also serialize reads.

	N_A	N_B
		lock(x)
blocked	lock(x)	

		read(x)
		unlock(x)

continue	read(x)
	unlock(x)	

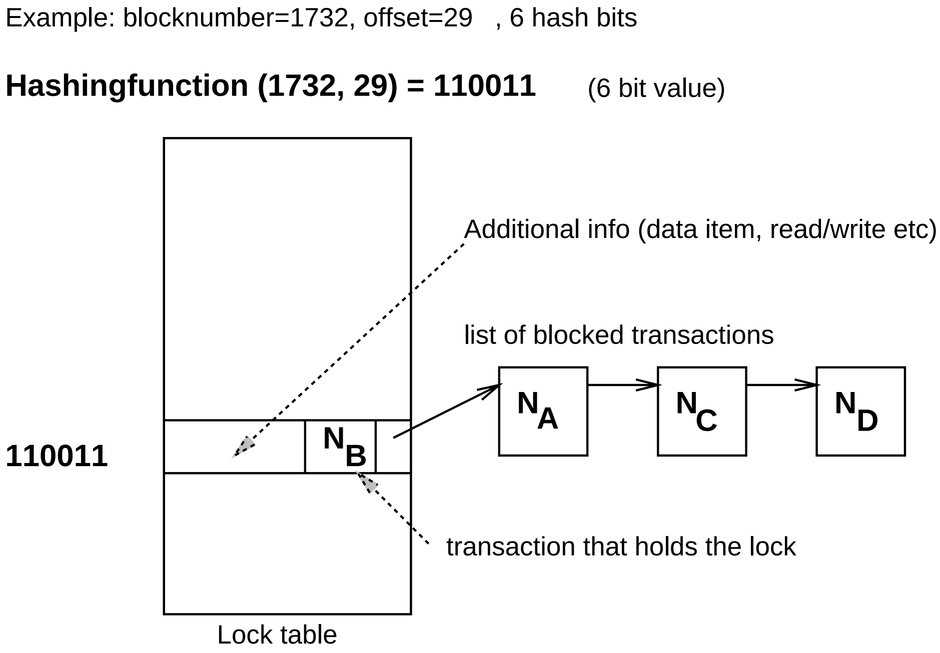
- It's better to use separate locks for reading and writing:
  - **readlock**(x) – lock for reading.
  - **writelock**(x) – lock for writing.
- Associate a counter with each locked item.
- When a transaction N_A does a **readlock**(x):
  - If any other transaction has a **writelock** on x, make N_A wait.
  - If the only locks on x are **readlock**'s, increment the counter associated with x and allow N_A to proceeed.
- When a transaction N_A does a **writelock**(x):
  - If anyone else has a lock on x, make N_A wait.
  - Otherwise, grant N_A the **writelock** and allow it to proceed.
- When a transaction N_A executes **unlock**(x):
  - If N_A had a **writelock** on x, release the lock.
  - If N_A had a **readlock** on x, decrement the counter. If the counter is zero, there is no lock on x.

- Implementing locks:
  - If a separate lock is used for each data item
    ⇒ lots of locks needed.
  - E.g., suppose a database has

20 relations
100,000 tuples per relation (average)
5 attributes per relation (average)

⇒ 10^7 possible locks!

- Of course, not all locks will be active at any given moment.
- A common approach is to decide apriori how many distinct locks to support, e.g. 1024 locks
  ⇒ use a 10-bit hash value for each data item.
- Recall: a data item is an address: blocknumber, offset and size.
- From these, create a 10-bit hash value and index into a hash table (lock table).


Example: blocknumber=1732, offset=29 , 6 hash bits
Hashingfunction (1732, 29) = 110011 (6 bit value)
Additional info (data item, read/write etc)
list of blocked transactions
110011
N_B
N_A
N_C
N_D
transaction that holds the lock
Lock table

- Note: the actions **readlock**(x), **writelock**(x) and **unlock**(x) must be atomic
  ⇒ usually implemented via semaphores.

- Other details about locks:
  - A transaction must not be allowed to block on itself:

writelock(x)
write(x)
writelock(x)

  - The system must make sure every transaction releases its locks after a commit or rollback.
  - A transaction should only be able to **unlock** something it's **lock**ed before.

- Deadlock:
  - Sometimes, transactions can deadlock.

N_A	N_B
readlock(x)	
read(x)	
	writelock(y)
	write(y)
readlock(y) // blocked	
	writelock(x) // blocked

  We will consider deadlocks later.

7.12 2-Phase Locking

- Unfortunately, locking by itself does not guarantee serializability, e.g.,

N_A	N_B
readlock(x) **read**(x) // Conflicts with N_B's **write** **unlock**(x)	
	writelock(x) **write**(x) // Conflicts with N_A's **read** **unlock**(x)
readlock(x) **read**(x) // Unrepeatable read **unlock**(x)	

  - By a series of swaps, we cannot serialize this schedule
    ⇒ it has the RW-conflict (unrepeatable read problem).
  - The problem is: N_A **unlock**ed x and allowed N_B to **write** x.
  - In 2-Phase Locking, this temporary unlocking is not allowed.

- Key ideas in 2PL (2-Phase Locking):
  - A transaction is never granted a lock after it executes its first **unlock**.
  - Thus, a transaction has two successive phases:
    - ∗ *Expanding phase*: acquire locks, but do not unlock anything.
    - ∗ *Shrinking phase*: unlock items, but do not request any locks.
  - If 2PL were used in the previous example:

N_A	N_B
readlock(x)	
read(x) // Do not release lock	
	writelock(x) // blocked
read(x)	
unlock(x)	
	write(x) // continued
	unlock(x)

⇒ isolation satisfied.

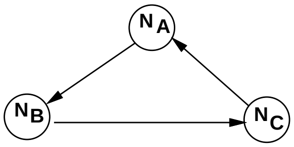
- 2PL and deadlock:
  - A 2PL schedule can still deadlock, e.g.,

N_A	N_B
readlock(x)	
	writelock(y)
readlock(y) // blocks on y-lock	
	writelock(x) // blocks on x-lock

- **Fact**: a deadlock-free 2PL schedule is always conflict-serializable.

  Proof sketch:

  - For simplicity, consider 3 transactions N_A, N_B, N_C and assume only **write** conflicts.
  - Suppose we have a deadlock-free 2PL schedule that is *not* conflict-serializable.
  - Since it's not conflict-serializable, the precedence graph has a cycle, e.g.,



  - The $N_A \rightarrow N_B$ edge indicates: N_A has a **write**(x) prior to N_B's **write**(x) (for some x).

    ⇒ This means N_A must have **unlock**ed x before N_B's **write**(x).

    ⇒ After this point, N_A cannot have locks on anything
    ⇒ It has no conflicts after this (since, if N_A has a conflict, it *must* lock).

  - Similarly, the $N_B \to N_C$ edge indicates: N_B has a **write**(y) before N_C's **write**(y) (for some y).
  - Finally, the $N_C \to N_A$ edge indicates: N_C has a **write**(z) before N_A's **write**(z) (for some z).
  - But, N_A must place a lock before its **write**(z)
    ⇒ this lock comes after the **unlock**(x)
    ⇒ contradiction!

- Deadlock-free 2PL:
  - A simple modification is sufficient to remove deadlock:

    A transaction that is blocked must release *all* the locks it has acquired.

  - Thus, a transaction either gets all its locks or none at all.
  - Example:

N_A	N_B
readlock(x)	
	writelock(y) // Acquired
readlock(y) // blocked	
release all locks	
	write(y)
	writelock(x) // Acquired
	write(x)
	unlock(x)
	unlock(y)
readlock(x) // Try again	
readlock(y)	
read(x)	
read(y)	
unlock(x)	
unlock(y)	

- Strict 2PL:
  - Recall: a *strict* schedule is one in which a transaction N_A reads a value written by N_B only if N_B has committed at the time of reading.
  - To enforce "strictness" in 2PL:
    Allow locks to be released by a transaction only after it either commits or rolls back.
  - Example:

N_A	N_B
	writelock(x)
readlock(x) // blocked	
	write(x)
	commit
	unlock(x)
read(x)	
unlock(x)	

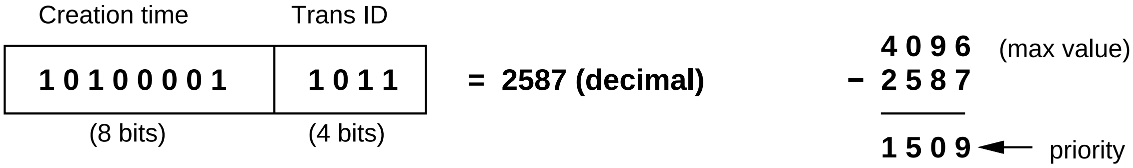
Here, N_A reads only committed data
$\Rightarrow$ the problem of reading an uncommitted-write can't occur.

- In practice, one of two methods is used:
  1. Strict deadlock-free 2PL, or
  2. Strict 2PL with a deadlock detection and resolution method.

- Note:
  - It is possible for a schedule to be deadlock-free but not strict (if locks are released before commit).
  - It is possible for a strict schedule to deadlock (blocking can occur well before commit).

7.13 Deadlocks

- Two approaches to the deadlock problem:
  1. Deadlock prevention:
     – Make sure it can't happen.
  2. Deadlock detection and resolution:
     – Allow deadlock to occur.
     – Find a way to detect it when it occurs.
     – Break the deadlock when found.
- Deadlock prevention using unique priorities:
  – Key ideas:
    ∗ Each transaction is given a unique priority number (integer).
    ∗ Typically, a transaction's priority is related to its time of creation.



Here, a transaction's creation time (8-bit) is concatenated with its ID (4-bit) to create a 12-bit string. This is subtracted from the maximum value of the 12-bit string (4096) to get the priority.

∗ The older the transaction (i.e., smaller the creation time), the higher (i.e., larger) the priority.

∗ Since priorities are unique, for any two transactions N_A and N_B, either

1. **priority**(N_A) < **priority**(N_B), or
2. **priority**(N_A) > **priority**(N_B)

but not both (**priority**(N_A) = **priority**(N_B) is not possible).

∗ Note that to prevent deadlock, we need to prevent circular wait

- e.g., "N_A is waiting for N_B which is waiting for N_C which is waiting for N_A" is a circular wait.

∗ The solution to circular wait: rollback one of the waiting transactions.

∗ Which transaction to rollback?

- Use priorities to determine whether a transaction should be rolled back or whether it should be allowed to wait.

∗ Since decisions are made at lock-request time, we are usually faced with the following scenario:

· N_B tries to acquire a lock already held by N_A.

· Should N_B be allowed to wait?

· Or should it be rolled back?

∗ Two methods are used:

1. **Wait-die** method:

```
If priority($N_B$) > priority($N_A$)
    Allow $N_B$ to wait
Else
    Roll back $N_B$, releasing its locks
    Restart $N_B$ later (with its old priority)
Endif
```

2. **Wound-wait** method:

```
If priority($N_B$) > priority($N_A$)
    Roll back $N_A$, releasing its locks
    Restart $N_A$ later (with its old priority)
Else
    Allow $N_B$ to wait
```

Endif

- Examples:

1. **Wait-die**:

   (a) $\mathbf{priority}(N_B) < \mathbf{priority}(N_A)$

No deadlock prevention

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y)	
	writelock(x) Deadlock

Wait−die method

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y)	
Allowed to wait	**writelock(x)** Not allowed to wait Roll back
N_A continues	Release all locks

   (b) $\mathbf{priority}(N_B) > \mathbf{priority}(N_A)$

No deadlock prevention

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y)	
	writelock (x) Deadlock

Wait−die method

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y)	
Not allowed to wait Roll back Release all locks	**writelock(x)**

2. **Wound-wait**:

   (a) $\mathbf{priority}(N_B) < \mathbf{priority}(N_A)$

No deadlock prevention

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y)	
	writelock (x) Deadlock

Wound−wait method

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y) Forces N_B to roll back	Roll back Release all locks

(b) **priority**$(N_B) >$ **priority**(N_A)

No deadlock prevention

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y)	
	writelock (x) Deadlock

Wait−die method

N_A	N_B
readlock (x)	
	writelock (y)
readlock (y)	
Allowed to wait	**writelock(x)**
Roll back Release all locks	Forces N_A to roll back

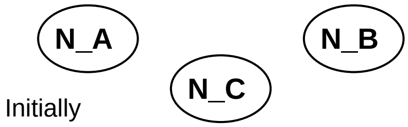
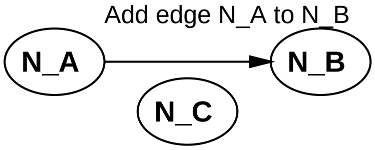
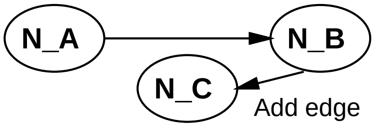
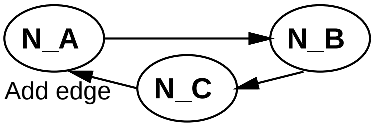
- Note: no deadlock is possible in either scheme (wait-die or wound-wait).

  Proof sketch (for wound-wait): Let N' be the oldest (highest-priority) transaction in system. Then, N' never waits and instead preempts any contending transaction

  $\Rightarrow N'$ will eventually complete. Then, second oldest will complete...then, third oldest...and so on.

- Deadlock detection:
  - Key ideas:
    - ∗ The dbase lock manager maintains a *wait-for* graph:
      - · One vertex for each transaction.
      - · If N_i blocks while waiting, for a lock held by N_j, then place an edge from vertex N_i to vertex N_j.
    - ∗ Every time a transaction is blocked, update the wait-for graph.

  - ∗ Check whether wait-for graph has a cycle.
  - ∗ If it has a cycle ⇒ deadlock has occured.
  - ∗ Select a transaction in the cycle to roll back and roll it back.
- – Example:

N_A	**N_B**	**N_C**	**Wait−for graph**
			
writelock (x)			Lock granted
	writelock (y)		Lock granted
		writelock (z)	Lock granted
readlock (y) Blocked − waiting for N_B			
	readlock (z) Blocked − waiting for N_C		
		readlock (x) Blocked − waiting for N_A	

- – Which one to roll back? Various options:
  - ∗ Roll back youngest transaction.
  - ∗ Roll back transaction with fewest writes.
  - ∗ Roll back shortest-running transacation.

- Deadlock suspicion:
  - – A simple alternative to *deadlock prevention* or *deadlock detection and resolution.*
  - – Every time a transaction blocks, start a timer.

– If timer goes off
  ⇒ transaction has waited too long
  ⇒ suspect deadlock
  ⇒ roll back transaction.

– Plus: Timeout's are simple to understand and easy to implement.

– Minus: The method is too conservative (long-waiting transactions are unnecessarily rolled back).

7.14 The Phantom Problem and Other Issues

- Recall how locking works:
  - To read or write to a data item:
    - Obtain address of item (blocknumber, offset).
    - Compute hash-value.
    - Look up hash-value in lock table (using hash-value as index).
    - Obtain lock on hash-value entry.
  - Example: consider the SQL statement

```
select  MIN (BALANCE)
from    CORP_ACCOUNTS
where   BALANCE > 100000;
```

  - At a very low level, this is typically implemented as:

```
for each tuple t in CORP_ACCOUNTS
    (blk, offset) := GET-BLOCK-AND-OFFSET (t.BALANCE);
    readlock (blk, offset, t.BALANCE);
    read (blk, offset, t.BALANCE);
    if it's less than MIN then MIN := t.BALANCE;
endfor
Release all locks // 2nd phase of 2PL
```

  - Note: locking could be done at file-level, but this would reduce concurrency.
- The Phantom problem:
  - Consider the execution of transactions N_A and N_B:

N_A	N_B

// Find lowest balance larger
// than 100,000
SELECT MIN (BALANCE)
INTO :lowest
FROM CORP_ACCOUNTS
WHERE BALANCE > 100000;	

	INSERT INTO CORP_ACCOUNTS
	VALUES (39, 'SONY', 101000);
	// Lock is granted on new tuple

// Find lowest balance larger
// than 100,000 (again)
SELECT MIN (BALANCE)
INTO :lowest
FROM CORP_ACCOUNTS
WHERE BALANCE > 100000;
// Unrepeatable read!	

- We get the *unrepeatable read* problem even if 2PL is used.
- Why?
  - The locking that occurs for N_B's INSERT does not interfere with N_B's **readlock**'s.
  - At a low level, N_B really does the rough equivalent of:

    ```
    b = Get-Last-Block (CORP_ACCOUNTS);
    t = Get-First-Empty-Space (CORP_ACCOUNTS);
    writelock (t);
    Write new values;
    unlock (t);
    ```

    The hash-value computed for the new tuple need not conflict with any of the hash-values computed by N_A's **readlock**'s
    ⇒ N_B does not wait
    ⇒ repeatable-read problem.
- Thus, an insert can cause strange effects even with 2PL
  ⇒ the Phantom problem.

- One solution: perform locking at the file-level.
  - However, this reduces concurrency.

- Predicate Locking:
  - Predicate locking allows sets of tuples to be locked.
  - Each set is defined by a predicate.
  - Example:
    - Consider the predicate

      (BALANCE > 100000).
    - Lock all tuples which satisfy this condition, e.g.,

      LOCK CORP_ACCOUNTS (BALANCE > 100000);
    - Now, inserts (or reads or updates) can be checked against predicates.
    - Maintain a table of predicates and allow table entries to be locked.
  - Predicate locking is general and useful but very expensive to implement.
    - It's hard to compare long predicates to determine if they clash (if one should lock out the other), e.g.,

      P1 = (BALANCE > 100000) **and** (MILES < 1000) **or**
      ((NAME < 'G') **and** (FLTNO > 17))
      P2 = (NAME < 'G') **or** (BALANCE >100000) **and**
      (MILES < 1000) **and** (FLTNO > 17)
  - Predicate locking is not used in practice, although the concept has led to variations that are used.

- Precision locks:
  - A variation of predicate locks.
  - Predicates are created as in predicate locks.
  - Predicates are not compared.

  - When a **write** occurs, the relevant predicates are computed using the given data (old and new values) to see if there's a conflict.
  - Inserts are treated similar to writes.

- Key-range locking:
  - Another variation of predicate locking.
  - Performed on queries, updates or inserts involving keys.
  - A file is divided into groups based on pre-selected ranges of the key, e.g.,

$$\begin{array}{ll} \text{group 1:} & 0 < \text{FLT\_ID} \leq 10 \\ \text{group 2:} & 11 < \text{FLT\_ID} \leq 20 \\ \vdots & \vdots \end{array}$$

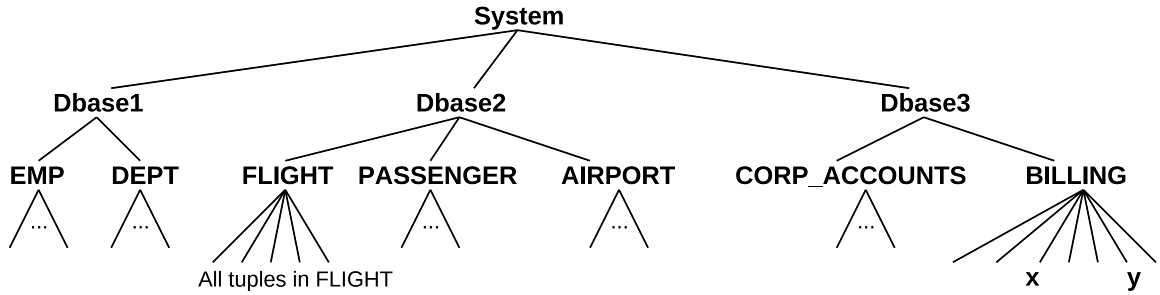
  - This is really the same as defining the predicates:

$$\begin{array}{lcl} \text{P1} & = & (0 < \text{FLT\_ID} \leq 10) \\ \text{P2} & = & (11 < \text{FLT\_ID} \leq 20) \\ \vdots & & \vdots \end{array}$$

  - The rest is the same as predicate locking, except that numeric ranges are much easier to deal with
    ⇒ key-range locking can be implemented efficiently.
  - Inserts beyond the ends are handled by defining special ranges:

$$\begin{array}{l} (-\infty < \text{FLT\_ID} \leq 0) \\ (\text{current maximum} < \text{FLT\_ID} \leq \infty) \end{array}$$

- Multiple-Granularity Locking:
  - A compromise between locking whole files or sets of files and tuple-level locking.
  - It uses the "containment property" for the hierarchy of objects we find in any database system:



- Suppose we want to lock FLIGHT for an UPDATE:

  Should the whole system be locked? Should all of Dbase2 be locked?

- Suppose we want to access x and y in Dbase3:

  Should all of Dbase3 or BILLING be locked?

- Define new locks:
  - **readintent** (x) – declare an intent to read something in the subtree of object x, e.g., **readintent** (Dbase3).
  - **writeintent** (x) – declare an intent to write something in the subtree of object x, e.g., **writeintent** (Dbase3).

- Also, allow **readlock** and **writelock** to apply to higher-level objects (files, databases).

- When a transaction wishes to lock something, it must lock top-down, stating its intentions along the way (using **readintent** and **writeintent**).

- Example:

N_A	N_B	N_C
writeintent(Dbase3)		
writeintent(BILLING)		
writelock(x)		
write (x)		
	writeintent(Dbase3)	
	writelock(BILLING)	
	// Blocked	
		writeintent(Dbase3)
		writeintent(BILLING)
		writelock(y)
		// Not blocked
		write(y)

- Permitted combinations that can be held on an object:

	readintent	**read**	**writeintent**	**write**
readintent	yes	yes	yes	no
read	yes	yes	no	no
writeintent	yes	no	yes	no
write	no	no	no	no

Example: if N_A has a **writeintent** on object x, then if N_B does a **readlock** on x, it will block.

7.15 Transactions in SQL

- Generally speaking,
  - the more careful we are about locking, the less concurrency we permit;
  - the less we're fussy about locking, the more the potential concurrency;
- SQL allows the user to decide what level of protection (and therefore, concurrency) a transaction wants.
- SQL allows the user to define one of four levels of protection:

```
set transaction isolation level
    [   read uncommitted |
        read committed |
        repeatable read |
        serializable ]
```

  - An isolation level determines the protection a transaction receives.
  - In particular:

Isolation Level	Lost Update	Dirty Read	Unrepeatable Read	Phantom Problem
read uncommitted	yes	yes	yes	yes
read committed	no	no	yes	yes
repeatable read	no	no	no	yes
serializable	no	no	no	no

  - Thus, a transaction that has set the isolation level to **read committed** will not experience the lost update problem, but might experience an unrepeatable read.

  - **Serializable** is the strictest (but allows for the least concurrency).

- Guidelines in choosing the isolation level:
  - Use **read uncommitted** for casual browsing of data (nothing guaranteed).
  - Use **serializable** for any write's.
  - Use **read committed** for computing approximate statistics (max, count, etc).

  Note:

  - Default is **serializable**.
  - To use **read uncommitted**, a transaction must be read-only.

Chapter 8

Oracle

Course Notes on Database Systems

8.1 Oracle: An Introduction

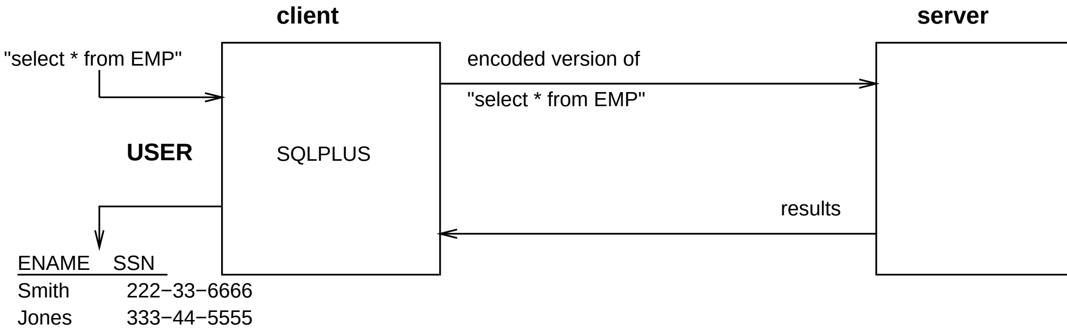
- What is Oracle?
  ⇒ Oracle is a commercial database product.

  Oracle consists of:

  1. A DBMS server:
     - Parses and executes SQL.
     - Executes "stored" PL/SQL code.
     - Handles query processing and optimization.
     - Handles disk I/O, file system and memory management (paging).
     - Provides concurrency, recovery and security.
  2. Database tools:
     - DBA tools: svrmgrl (line mode) and svrmgrm (window mode).
       * System startup and monitoring.
       * Add/delete users, set priveleges for users.
       * Control space usage.
     - SQL interpreter: sqlplus
       * Prompt user for SQL input and connect to server.
       * Pass SQL queries to server and retrieve results.
       * Other features: formatting, transaction processing, Oracle extensions to SQL.
       * PL/SQL compiler.
     - A dbase programming language: PL/SQL
       * An (almost) full programming language.
       * Syntax and keywords based on Ada.
       * Standard libraries for process control, transaction processing, I/O.

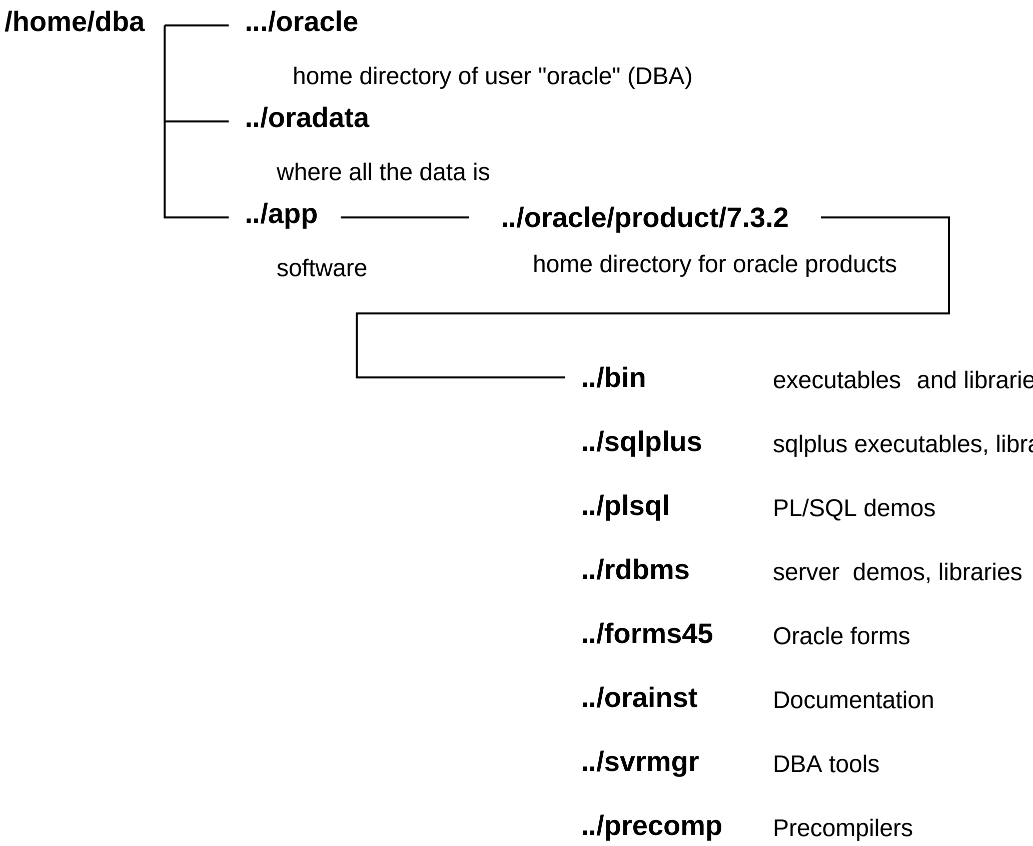
- A window-based programmable user-interface: OracleForms
    - ∗ Can set up window-based environment for end-users.
    - ∗ Customizable and programmable.
- Other tools:
    - ∗ A report-generating and formatting tool: OracleReport.
    - ∗ A tool for designing web-based interfaces: Webserver.
    - ∗ A tool for handling spatial data.
    - ∗ Tools for distributed data, parallel execution of server, graphics and multimedia.

- Oracle is designed to work in client-server mode:
  ⇒ the tools operate as server clients.



- Oracle supports all of SQL/92 (the most recent ANSI SQL standard).

- Oracle on Unix:
    - Oracle is usually under a directory called `/home/dba/`.
    - The directory structure in the standard installation looks like:



- Oracle files and tablespaces:
  - Oracle keeps data in a bunch of tablespaces:
    ⇒ a tablespace is a group of related data
    For example, all students have their data in tablespace `STUDENT97`.
  - Each tablespace consists of a number of (unix) files.
  - A relation is placed in a tablespace and may spread across several files.
  - The files are in `/home/dba/oradata/`.
  - Each user is required to have a:
    - ∗ Default tablespace: where the user's data is stored.
    - ∗ Temp tablespace: for intermediate computations (e.g., sorting).
  - Standard tablespaces: SYSTEM, TEMP, TOOLS, USER.
- Oracle processes: an Oracle server installation will have several processes running simultaneously:

  - DBWR: Database writer – for disk I/O.
  - LGWR: Log writer – for writing recovery information.
  - SMON, PMON: System monitor and process monitor – to handle connections, client-server communication.
  - Server processes.

- Oracle handles its own memory management: a huge piece of main memory (System Global Area) is kept for:
  - Data and catalogs.
  - Current and “stored” SQL and PL/SQL.
  - Memory for indexing, processing, recovery.

8.2 Oracle Datatypes

- Oracle has several datatypes, among which are:
  - CHAR(n) – Fixed-length character string.
    - n is the length.
    - Default length is 1.
    - Length must be between 1 and 255 bytes.
    - Fixed amount of space is allocated.
  - VARCHAR2(n) – variable-length character string.
    - n is the maximum length (must be specified). $1 \leq n \leq 2000$.
    - Space usage depends on length of string.
    - Use VARCHAR2 where possible.
  - VARCHAR(n) – currently same as VARCHAR2.
    - VARCHAR2 and VARCHAR correspond to the ANSI definition of VARCHAR.
    - VARCHAR may change with the next change in ANSI standards.
  - NUMBER(m, n) – fixed and floating point numbers.
    - m is the total number of digits.
    - n is the number of digits after the decimal point.
    - NUMBER($*$,n) specifies decimal accuracy but unlimited size.
    - Numbers are in the range 10^{-130} to 10^{125}.
  - INTEGER – integers.
  - DATE – to store a dates (year,month,day,time).
    - Date range: Jan 1, 4712 BC to Dec 31, 4712 AD.
    - A library of date-manipulation functions is provided.

  - LONG – to store large amounts of text data.
    - Up to 2GB of text data.
    - Only *one* LONG column is permitted per relation.
    - LONG attributes cannot be used in the **where** subclause and in certain other places.
  - RAW(n) – to store non-text data.
    - n is size of field: $1 \leq n \leq 255$.
    - RAW data is not converted into any format.
  - LONG RAW – to store images, graphics etc.

- For comparison, here are some standard datatypes in SQL/92:

  > CHAR, VARCHAR, NCHAR, NATIONAL CHAR, BIT, NUMERIC, DECIMAL, DEC, INTEGER, INT, SMALLINT, FLOAT, REAL, DOUBLE PRECISION, DATE, TIME, TIMESTAMP.

8.3 Oracle and SQL

- Oracle provides the tool sqlplus to execute SQL statements:
  - You can type SQL statements at the Sqlplus the command line directly:

    ```
    % sqlplus <username>/<password>
    SQL> select * from EMP;
    ```

  - Alternatively, you can place SQL statements in a file and execute them by specifying the file name, e.g.,
    - Suppose the file test1.sql contains the the string "select * from EMP".
    - To execute the SQL:

      ```
      SQL> @test1.sql
      ```

- Sqlplus has number of Oracle-specific commands and features:
  - Use help to obtain a complete list of commands.
  - The command describe tells you the structure of a table, e.g., to find out the definition of relation EMP:

    ```
    SQL> desc EMP
    ```

  - Output can be directed to a file using spool, e.g.,

    ```
    SQL> spool test1.output
    SQL> @test1.sql
    SQL> spool off
    ```

  - Sqlplus supports the definition and use of variables, e.g.,

```
define name_var = ’Smith’
select E.SSN
from EMP E
where E.NAME = ’&name_var’;
```

If a variable name is used without definition, e.g.,

```
select E.SSN
from EMP E
where E.NAME = ’&name_var’;
```

then Sqlplus prompts the user for the value.

- Sqlplus has features for rudimentary screen I/O, e.g.,

```
clear screen
accept name_var prompt Enter an employee name:

select E.SSN
from EMP E
where E.NAME = ’&name_var’;
```

- Oracle SQL differs from standard SQL in some ways, e.g., Oracle uses **minus** instead of **except** for set difference.

- Some additional useful facts:
  - Use ‘`;`’ (semi-colon) to see the last command executed.
    ⇒ command is in server’s SQL buffer
    ⇒ already parsed and ready for execution.
  - Use ‘`\`’ (backslash) to execute last command.
  - Use ‘`!`’ (exclamation) before Unix command to execute a Unix command.
  - To see a list of tables that you have created:
    `select table_name from user_tables;`

- System tables that may be useful to you: `all_tables`, `all_users`, `dba_free_space`, `dba_tablepsaces`, `dba_users`.

8.4 PL/SQL: Introduction

- Consider the following relation:

TRAVEL	FLT_NO	DEST	NEAREST	AIRLINE
	139	Montreal	New York	Air Canada
	452	Brussels	London	Sabena
	339	London	New York	British Airways
	116	Macau	Hong Kong	Macau Air
	322	Hong Kong	Tokyo	Japan Air Lines
	43	Tokyo	San Francisco	USAir
	31	San Francisco	New York	McValue

  – Suppose we are given a destination and we want to produce a list of flights starting from New York to that destination.

  – We can't do this in SQL.
    ⇒ cannot compute transitive closure in SQL
    ⇒ alternative programming language needed.

- Most DBMS's extend SQL with their own programming language.

- In Oracle: PL/SQL (Procedural Language extensions to SQL):

  – PL/SQL has (almost) the full power of many programming languages (like C).

  – PL/SQL's syntax and keywords are modeled on Ada.

  – PL/SQL has dbase-specific constructs such as *cursors* and *anchored declarations.*

  – PL/SQL has pre-defined libraries for I/O, process control and interprocess communication.

- "Hello World" in PL/SQL:

  – There is no separate tool for PL/SQL.

  - PL/SQL can be entered at the command line of Sqlplus or run as a file.
  - Suppose the file `test1.pl` contains:

    ```
    begin
      dbms_output.enable;
      dbms_output.put_line('Hello World');
    end;
    /
    ```

  - Then, the Sqlplus commands

    ```
    SQL> set serveroutput on;
    SQL> @test1.pl
     Hello World
    ```

    cause the file to be executed.

- In general a PL/SQL program will have the following structure:

  ```
  declare
    /* variables, if any, are declared here */
  begin
    /* code */
  exception
    /* exception handling code -- optional */
  end; /* Note the semi-colon */
  /    /* Note the forward-slash */
  ```

- Datatypes in PL/SQL: all of Oracle's SQL datatypes are available in PL/SQL, and with the same names, e.g.,

  char, varchar2, number, date, boolean, long,
  raw

Example: the following are declarations in PL/SQL:

```
declare
  i integer;
  x number(10,3);
  ename varchar2(50);
begin
  null;
end;
```

- Anchored declarations:
  - Suppose I have the following declaration:

    ```
    emp_name varchar2(50);
    ```

  - Later, we want to make comparisons with a variable called `mgr_name`.
  - One option:

    ```
    emp_name varchar2(50);
    mgr_name varchar2(50);
    ```

  - Instead, it's better to let the type of `mgr_name` depend on `emp_name` using an *anchored declaration*:

    ```
    emp_name varchar2(50);
    mgr_name emp_name%TYPE;
    ```

  - Anchored declarations are even more useful in creating variables to hold tuples.
  - For example, suppose we have created (in SQL) a table called `Travel`:

```
create table Travel (  Flt_no   integer
                       Dest     varchar(50)
                       Nearest  varchar(50)
                       Airline  varchar(50) );
```

To declare variables of the same types as above in PL/SQL:

```
fltnum Travel.Flt\_no%TYPE;
which_airline Travel.Airline%TYPE;
```

– It is even possible to define a variable to hold a complete tuple:

```
declare
  travel_tuple Travel%ROWTYPE;
begin
  ...
  dbms_output.put_line (travel_tuple.Airline);
  ...
end;
```

8.5 PL/SQL: Control Statements

- Like any standard high-level programming language, PL/SQL has conditional constucts:

```
if <cond> then
  /* statements */
end if;  /* Note the space between end and if */

if <cond> then
  /* blah-blah */
else
  /* rah-rah */
end if;

if <cond> then
  /* s1 */
elsif <cond>
  /* s2 */
elsif <cond>
  /* s3 */
else
  /* s4 */
end if;
```

- Loops:
  - The simplest loop in PL/SQL is of this form:

```
loop
  ...
```

```
 if <cond> then exit;
 ...
end loop;
```

- To simulate a C-while loop, for example

  ```
  i = 1;
  while (i <= n) {
    ...
  }
  ```

  use the following PL/SQL loop:

  ```
  i := 1;
  loop
    if (i > n) then exit;
    ...
  end loop;
  ```

- Similarly, for example, the following C-forloop

  ```
  for (i=1; i<=n; i++) {
    ...
  }
  ```

  can be simulated in PL/SQL as:

  ```
  i := 1;
  loop
    if (i > n) then
      exit;
    else
      i := 1 + 1;
    end if;
    ...
  ```

```
          end loop;
```

- Note: several variants of loops exist in PL/SQL (see the manual).

8.6 PL/SQL: Cursors

- SQL by itself provides no for-loop like mechanism to iterate through the tuples of a relation.
- Yet, it is very useful to be able to retrieve tuples one-by-one for processing.
- Most DBMS vendors provide such mechanisms – often called cursors.
- Oracle provides cursors in PL/SQL as well as in the C interfaces.
- What exactly is a cursor?
  - To understand cursors, first consider a list of items in C++, e.g.,

```
struct list_item {
  ...
} item_type;

class list {
  ...
  // Access functions
  void start ();         // Initialize walk through list
  item item_next ();     // Return current item
  boolean list_empty (); // Is the list empty?
  boolean at_end ();     // Are we at the end?
};
```

  - This `list` class would be used as:

```
list A;
item_type item;
...
```

```
if (! A.list_empty () ) {
  A.start ();
  while (! A.at_end () ) {
    item = A.next ();
    ...
  }
}
```

  - Note that the class `list` is a *generalization* of a simple for-loop variable.
  - A cursor is similar:
    - Cursors provide ways to determine if a relation is empty, to start a scan and determine if the scan has ended.
    - A cursor is an "object" that scans a relation.
    - Several cursors can be defined for a single relation.
    - Each cursor applies to only one relation.
  - In PL/SQL, you can:
    - Declare a cursor.
    - Open a cursor.
    - Use a cursor repeatedly.
    - Close a cursor.

- Example: consider the relation

  travel (Flt_no, Dest, Nearest, Airline).

  Let's write a PL/SQL program to "curse" through the relation:

```
DECLARE
  CURSOR travel_cursor IS   /* Declaration of cursor */
    SELECT *
    FROM Travel;
  travel_tuple Travel%ROWTYPE;
BEGIN

  DBMS_OUTPUT.ENABLE;

  OPEN travel_cursor;

  /* Fetch first tuple from travel relation */
  FETCH travel_cursor INTO travel_tuple;

  IF travel_cursor%FOUND THEN
    DBMS_OUTPUT.PUT ('First FLT\_NO found is: ');
    DBMS_OUTPUT.PUT_LINE (travel_tuple.Flt\_no);
  ELSE
    DBMS_OUTPUT.PUT_LINE ('Empty relation');
  END IF;

  CLOSE travel_cursor;

END;
```

Note:

- If the above code is in a file called `test2.pl`, it can be run from Sqlplus:

  ```
  SQL> set serveroutput on;
  SQL> @test2.pl
  ```

- The program doesn't do much: it only fetches one tuple.
- PL/SQL keywords are capitalized for reading convenience.

- Let's now write a more interesting program: to print out all the tuples in Travel.

```
DECLARE
  CURSOR travel_cursor IS   /* Declaration of cursor */
    SELECT *
    FROM Travel;

  travel_tuple Travel%ROWTYPE;

  i INTEGER;

BEGIN

  DBMS_OUTPUT.ENABLE;

  IF travel_cursor%ISOPEN THEN
    DBMS_OUTPUT.PUT_LINE ('Error: cursor already open');
  ELSE
    OPEN travel_cursor;
  END IF;

  /* Now curse through Travel */
  i := 1;
  LOOP
    FETCH travel_cursor INTO travel_tuple;
    IF travel_cursor%NOTFOUND THEN
      EXIT;
    END IF;
    DBMS_OUTPUT.PUT ('Tuple #');
    DBMS_OUTPUT.PUT (i);
    DBMS_OUTPUT.PUT (': ');
    DBMS_OUTPUT.PUT (travel_tuple.Flt\_no);
    DBMS_OUTPUT.PUT (travel_tuple.Dest);
    DBMS_OUTPUT.PUT (travel_tuple.Nearest);
    DBMS_OUTPUT.PUT_LINE (travel_tuple.Airline);
    i := i + 1
  END LOOP;

CLOSE travel_cursor;

END;
```

- Thus, the key steps in using a cursor are:

  1. Declaring a cursor via an SQL **select**, e.g.,

     ```
     CURSOR travel_cursor IS   /* Declaration of cursor */
       SELECT *
       FROM Travel;
     ```

  2. Opening a cursor, e.g.,

     ```
     IF travel_cursor%ISOPEN THEN
       DBMS_OUTPUT.PUT_LINE ('Error: cursor already open');
     ELSE
       OPEN travel_cursor;
     END IF;
     ```

     Note: it's always wise to check whether another cursor has been opened on the same relation.

  3. Using FETCH to obtain successive tuples.
  4. CLOSE-ing a cursor.

- Note that the **select** in a cursor declaration can be quite powerful, e.g.,

  ```
  CURSOR travel_cursor IS   /* Declaration of cursor */
    SELECT *
    FROM Travel
    WHERE Travel.Airline = 'McValue'
    ORDER BY Travel.Dest;
  ```

This cursor will only retrieve those tuples satisfying the **where** condition.

Similarly, a cursor can be limited to a few attributes (via a projection):

```
DECLARE
  CURSOR travel_cursor IS   /* Declaration of cursor */
    SELECT Dest, Nearest
    FROM Travel
    WHERE Travel.Airline = 'McValue';

  travel_tuple travel_cursor%ROWTYPE
  /* travel_tuple only has 2 fields */
```

- The following special attributes are automatically available with any cursor, e.g., for `travel_cursor`:

travel_cursor%FOUND	Check if a tuple was retrieved
travel_cursor%NOTFOUND	See if nothing was retrieved
travel_cursor%ISOPEN	Is the cursor already open?
travel_cursor%ROWCOUNT	How many tuples are returned?

The last attribute counts the number of tuples that matched the defining **select** in the cursor definition.

8.7 PL/SQL: Procedures and Functions

- PL/SQL supports the definition and use of procedures and functions.
- Example:

```
PROCEDURE print_tuple_num (i IN INTEGER) IS
  /* Local variables are declared here */
BEGIN
     DBMS_OUTPUT.PUT ('Tuple #');
     DBMS_OUTPUT.PUT (i);
     DBMS_OUTPUT.PUT (': ');
END;

DECLARE
  ...
BEGIN
  ...
  print_tuple_num (i);
  ...
END;
```

Note:

- The general definition of a procedure has this structure:

```
PROCEDURE <name> ( <parameters> )
  <local variables>
BEGIN
  <body>
EXCEPTION
  <exception code>
```

```
        END;
```

- Each parameter, apart from having a data type, also has an input type: either IN, OUT or IN OUT.
- Example:

```
PROCEDURE proc1 (x IN INTEGER, y OUT INTEGER, z IN OUT INTEGER)
```

- Functions can be defined which return a value. The general form is

```
FUNCTION <name> ( <parameters> )
  RETURN <return type>
IS
  <local variables>
BEGIN
  <body>
EXCEPTION
  <exception code>
END;
```

  For example:

```
FUNCTION circle_area (radius IN REAL)
  RETURN REAL
IS
  area REAL;
BEGIN
  area := 3.141 * radius * radius;
  RETURN area;
END;
```

- Packages:

- A collection of functions and procedures can be encapsulated in a **package** that other developers can use.
- PL/SQL's `package` is weaker than `C++`'s `class`.
  ⇒ encapsulation, but no dynamic creation.

8.8 Oracle and C

- Oracle (as does almost every dbase vendor) provides two mechanisms for programming in C and accessing the dbase:
  1. Using the Oracle Call Interface:
     - A library of C functions.
     - Functions for executing SQL, directing output and taking input from user-defined C variables.
     - Currently, Oracle supports only some loaders.
       ⇒ Use the loader `ld` in `/opt/SUNWspro/bin`.
  2. Using *precompilers*:
     - SQL-statements can be inserted in a C program.
     - The C precompiler parses the SQL blocks.
     - Precompilation allows for some compile-time optimizations.
- Oracle also has language support for Ada, Fortran and Cobol.
- First, we'll focus on the OCI SQL library.
- To create your C program that interfaces with Oracle, you will need the following includes:

```
#include <stdio.h>      /* Standard C library */
#include <ctype.h>      /* Standard C library */
#include <string.h>     /* Standard C library */

#include <oratypes.h>   /* Definitions of Oracle datatypes, constan
#include <ocidfn.h>     /* Variable types for local data */
#include <ociapr.h>     /* Definitions of function calls */
#include <ocidem.h>     /* Constants used in demo programs. Useful
```

8.9 Oracle OCI: Setting up a Connection

- The first thing your C program will do is to set up a connection with the Oracle server.
  ⇒ username and password need to be specified
- The library function `olog(...)` is used to set up a connection.
- The file `ORACLE_HOME/rdbms/demo/ociapr.h` contains the definition:

  ```
  sword  olog     (struct cda_def *lda, ub1* hda,
                   text *uid, sword uidl,
                   text *pswd, sword pswdl,
                   text *conn, sword connl,
                   ub4 mode);
  ```

  which makes very little sense.
- Oracle defines its own data types in place of standard C data types:
  - Example: the data type `text *` is really `char *`, a character string.
  - Example: the data types `sword` and `uword` correspond to `signed int` and `unsigned int` in C.
  - Here are some of the more important ones:

ORACLE	C
text *	char *
sb1,ub1	char
sb2,ub2	signed and unsigned 2-byte int's
sb4,ub4	signed and unsigned 4-byte int's
sword, uword	signed and unsigned int's

- Note: most often you will use `sword` instead of `int`.
- Use these declarations for variables used in OCI calls. For example, to pass the integer `num_rows` to a call that requires `ub4`, do a cast:

  ```
  int my_mode = 5;
  ub4 mode_in_call;
  mode_in_call = (ub4) my_mode;
  /* Now mode_in_call can be used in the function call */
  ```

- These weird type names have been defined so that the definitions can vary from system to system (in the file `oratypes.h`) while keeping function definitions the same (in `ocidfn.h`).

• Every OCI program *must* have certain global variables:

```
/* Global variables */
Lda_Def lda;                          /* Logon data area */
Cda_Def cda;                          /* Cursor data area */
ub4 hda[HDA_SIZE/sizeof(ub4)];        /* Host data area */
```

- The *logon data area* is for Oracle (the OCI client) to place your connection ID, error codes relating to function calls etc.
- The *cursor data area* is for Oracle to keep track of current locations of cursors.
  Note: a cursor is a programming technique unique to dbases.
- The *host data area* is also used by Oracle for information related to your program.
- How does Oracle know the addresses of these variables?
  ⇒ you are going to pass these on to Oracle very early.
- Note: multiple lda's and cda's may be used for multiple connections. [See the documentation].

- A typical `olog` call looks like:

```
text * username = (text *) "Smith";
text * password = (text *) "S93JfiIE";
if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
         (text *) 0, -1, OCI_LM_DEF) ) {
   /* olog does not return 0 */
   printf ("Function call failed\n");
   exit (0);
}
/* Else, olog returned 0 => successful connection */
```

- Let's take a closer look:

```
if ( olog           /* Returns 0 if successful */
     (&lda,         /* Pass address of lda */
     (ub1 *)hda,    /* Pass address of hda */
     username,      /* Give username */
     -1,            /* If username is not null-terminated,
                       give length */
     password,      /* Give username */
     -1,            /* Length, if not null-terminated */
     (text *) 0,    /* For use in a networking application */
     -1,            /* Length for above */
     OCI_LM_DEF)    /* Blocking or non-blocking mode */
     ) {
```

Thus, the only "real" information is: username and password.

- Note: the C code contained the password. A more intelligent way to handle this problem:

```
text username[20];
text password[20];
printf ("username: ");   gets( (char*) username);
printf ("password: ");   gets( (char*) password);
if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
           (text *) 0, -1, OCI_LM_DEF) ) {
   printf ("Connection failed. Exiting...\n");
   exit (1);
}
```

- To close a connection, use `ologof (...)`:

```
  if (ologof (&lda) ) {
    printf ("Error during log out\n");
    exit (1);
  }
```

8.10 Oracle OCI: Executing SQL from C

- An SQL statement in a C program is executed with three steps:

  1. *Cursor definition*: every SQL query is associated with a cursor. Thus, a cursor is first associated with a connection using the `oopen` call:

     ```
     oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
     ```

  2. *Parsing*: the function `oparse` is used to parse an SQL string:

     ```
     text * sqlquery = "select NAME, SSN from EMP";
     oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
     oparse (&cda, sqlquery, -1, FALSE, VERSION_7);
     ```

  3. *Execution*: the function `oexec` is then used to execute the query:

     ```
     text * sqlquery = "select NAME, SSN from EMP";
     oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
     oparse (&cda, sqlquery, -1, FALSE, VERSION_7);
     oexec (&cda);
     ```

- A closer look at these functions:

  1. The `oopen` function:

     ```
     oopen (&cda,       /* The cursor global variable we've defined */
            &lda,       /* The current connection identifier */
            (text *) 0, /* Remaining parameters can be ignored. */
            -1,         /* They are for compatibility with earlier */
            -1,         /* Oracle versions */
            (text *) 0,
     ```

```
          -1);
```

Thus, the `oopen` call really tells Oracle “Here’s a cursor variable that I want you to associate with my current connection”.

2. The `oparse` function:

```
    oparse( &cda,        /* Ptr to CDA specified by OOPEN */
            sqlquery,    /* String containing sql query*/
            (sb4) -1,    /* Length of SQL statement, if not
                            NULL terminated */
            FALSE,       /* Value 0 => parse NOW
                            Value 1 => parse later */
            VERSION_7)   /* Value 2: version info */
```

   NOTE:

   - This function causes Oracle to parse the SQL query in the string `sqlquery`.
   - It is possible to request *deferred parsing* for efficiency reasons.
   - The deferred parsing will force parsing at the time of execution.
   - The SQL statement can be a Data Definition statement (e.g., a **create table** statement) in which case it is executed if you did not defer parsing.

3. The `oexec` statement:

```
   oexec (&cda);   /* Simply specify the cursor associated
                      with the query */
```

   NOTE:

   - The `oexec` function can be used for *queries*, *updates* and *inserts.*
   - Oracle recommends using other functions for *queries* only.

- What about output from the query?

- When `oexec` is called, the query is executed and the results are placed in a temporary buffer.
- To actually obtain the data, there are two steps:
  1. Use the `odefin` call to indicate where you want the result placed (in your C variables).
  2. Use the function `ofetch` repeatedly to get one tuple at a time.

- Getting results into C variables using `odefin`:
  - Consider the following SQL statement:

select	NAME, FLT_ID
from	PASSENGER

  - Suppose that NAME is VARCHAR(50) and FLT_ID is an INTEGER.
  - In the C program, we would create the following string:

    ```
    text * sqlquery = "select NAME, FLT_NO from PASSENGER"
    ```

  - This SQL statement would be parsed using `oparse`:

    ```
    text * sqlquery = "select NAME, FLT_NO from PASSENGER";
    oopen (&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
    oparse (&cda, sqlquery, -1, FALSE, VERSION_7);
    ```

  - Now, we want the output (NAME and FLT_NO) to be placed in C variables of our choice.
  - Accordingly, define the C variables:

    ```
    #define PNAME_LEN 50
    text pname[PNAME_LEN];
    sword fltnum;
    ```

  - Unfortunately, Oracle requires that *three additional variables* be defined for each such attribute:

```
text pname[PNAME_LEN];     /* Passenger name */
  sb2 ind_pname;      /* An indicator code for function calls */
  ub2 retc_pname;     /* A return code - to indicate errors */
  ub2 retl_pname;     /* The length of the value returned */
sword fltnum;       /*Flight number */
  sb2 ind_fltnum;
  ub2 retc_fltnum;
  ub2 retl_fltnum;
```

Thus, for any variable x, you need ind_x, retc_x and retl_x defined.

- Next, use the odefin call to tell Oracle your intentions for each variable:

```
odefin (&cda, 1, (ub1*) pname, (sword) PNAME_LEN, SQLT_STR,
        -1, &ind_pname, (text *) 0, -1, -1,
        &retl_pname, &retc_pname);

odefin (&cda, 2, (ub1*) &fltnum, (sword) sizeof(int), SQLT_INT,
        -1, &ind_fltnum, (text *) 0, -1, -1,
        &retl_fltnum, &retc_fltnum);
```

- Let's look at one of these in a little more detail:

```
odefin (
 &cda,              /* The cursor associated with the query */
 1,                 /* Position of NAME in select query:
                       NAME is the *first* attribute in
                         select NAME, FLT_NO from ...
                       Use 2 for FLT_NO */
 (ub1*) pname,      /* Address of the C variable
                       - cast to ub1* */
 (sword) PNAME_LEN, /* Size of space allocated for result */
 SQLT_STR,          /* It's a string type: SQLT_STR is a
```

```
                               pre-defined constant in ocidfn.h */
     -1,                 /* Not used */
     &ind_pname,         /* Did a row-fetch work? */
     (text *) 0,         /* Not used */
     -1,                 /* Not used */
     -1,                 /* Not used */
     &retl_pname,        /* Length of data returned to be put here */
     &retc_pname)        /* A return code */
```

– After the addresses of your C variables have been handed over to Oracle, tuples can be fetched one at a time using `ofetch`:

```
ofetch (&cda);    /* Provide the cursor identification */
```

- A complete example: consider the relations:

ACCOUNTS (SSN, ACC_CODE, ACCNUM, BRANCHNUM, AMOUNT)
CUSTOMER (SSN, NAME, ADDR, CITY, STATE)

Goal: print each customer's name and balance.

– First, `include`'s and global variables:

```
#include <stdio.h>      /* Standard C library */
#include <ctype.h>      /* Standard C library */
#include <string.h>     /* Standard C library */

#include <oratypes.h>   /* Definitions of Oracle datatypes */
#include <ocidfn.h>     /* Variable types for local data */
#include <ociapr.h>     /* Definitions of function calls */
#include <ocidem.h>     /* Constants used in demo programs */

Lda_Def lda;                      /* Logon data area */
Cda_Def cda;                      /* Cursor data area */
ub4 hda[HDA_SIZE/sizeof(ub4)];    /* Host data area */
```

– Second, let's define a function called `connect` that handles username and password:

```
void connect () {
  text username[20];
  text password[20];
  printf ("username: ");   gets( (char*) username);
  printf ("password: ");   gets( (char*) password);
  if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
           (text *) 0, -1, OCI_LM_DEF) ) {
    printf ("Connection failed. Exiting...\n");
    exit (1);
}
```

– Define the SQL statement that produces the desired result:

```
text * sqlquery
  = "select C.NAME, A.AMOUNT from ACCOUNTS A, CUSTOMER C
      where A.SSN=C.SSN";
```

– Next, define the variables in which we want the results:

```
#define CNAME_LEN 50
text cname[CNAME_LEN];     /* Customer name */
  sb2 ind_cname; ub2 retc_pname; ub2 retl_pname;
sword amount               /* Amount */
  sb2 ind_amount;  ub2 retc_amount;  ub2 retl_amount;
```

– Next, connect, define a cursor, parse the SQL statement and alias the C variables.

```
connect ();

oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
```

```
    oparse (&cda, sqlquery, -1, FALSE, VERSION_7);

    odefin (&cda, 1, (ub1*) cname, (sword) CNAME_LEN, SQLT_STR,
            -1, &ind_cname, (text *) 0, -1, -1,
            &retl_cname, &retc_cname);

    odefin (&cda, 2, (ub1*) &amount, (sword) sizeof(int), SQLT_INT,
            -1, &ind_amount, (text *) 0, -1, -1,
            &retl_amount, &retc_amount);
```

- Now execute and fetch resulting tuples one by one:

  ```
  oexec (&cda);

  do {
    ofetch (&cda);
    /* Check whether anything was returned
       -- look at the .rc field of cda */
    if (cda.rc == NO_DATA_FOUND) break;
    printf ("%s %d \n", cname, amount);
  } while (1);
  ```

- Close cursors and log off:

  ```
    oclose (&cda);
    ologoff (&lda);
  ```

The complete code is:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include "oratypes.h"   /* Definitions of Oracle datatypes, constants
```

```
#include "ocidfn.h"     /* Variable types for local data */
#include "ociapr.h"     /* Definitions of function calls */
#include "ocidem.h"     /* Constants used in demo programs. Useful */

#define VERSION_7 2

Lda_Def lda;                      /* Logon data area */
Cda_Def cda;                      /* Cursor data area */
ub4 hda[HDA_SIZE/sizeof(ub4)];    /* Host data area */

void connect ()
{
  text username[20];
  text password[20];

  printf ("username: ");   gets( (char*) username);
  printf ("password: ");   gets( (char*) password);
  if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
             (text *) 0, -1, OCI_LM_DEF) ) {
    printf ("Connection failed. Exiting...\n");
    exit (1);
  }
}

text * sqlquery = (text *) "select C.NAME, A.AMOUNT from simha.ACCOUN

#define CNAME_LEN 50
text cname[CNAME_LEN];     /* Customer name */
  sb2 ind_cname; ub2 retc_cname; ub2 retl_cname;
sword amount;               /* Amount */
  sb2 ind_amount;  ub2 retc_amount;  ub2 retl_amount;

int n;
```

```
void main ()
{
  connect ();

  if (oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1) ) {
    printf ("oopen failed \n"); exit (0);
  }

  if (oparse (&cda, sqlquery, -1, FALSE, VERSION_7)) {
    printf ("oparse failed \n"); exit (0);
  }

  if (odefin (&cda, 1, (ub1*) cname, (sword) CNAME_LEN, SQLT_STR,
          -1, &ind_cname, (text *) 0, -1, -1,
      &retl_cname, &retc_cname)) {
    printf ("first odefin failed \n"); exit (0);
  }


  if (odefin (&cda, 2, (ub1*) &amount, (sword) sizeof(int), SQLT_INT,
             -1, &ind_amount, (text *) 0, -1, -1,
             &retl_amount, &retc_amount)) {
    printf ("second odefin failed \n"); exit (0);
  }

  if (oexec (&cda)) {
    printf ("oexec failed \n"); exit (0);
  }

  n = 0;
  do {
    ofetch (&cda);
    /* Check whether anything was returned
       -- look at the .rc field of cda */
    if ( (n > 100) || (cda.rc == NO_DATA_FOUND) ) break;
    cname[retl_cname + 1] = '\0';
    printf ("%s %d \n", cname, amount);
    n++;
  } while (1);

  printf ("%2d tuples found\n", n);

  oclose (&cda);
  ologof (&lda);

}
```

To obtain the complete code, see the file `~simha/321/oracle/test1.c`.

- Suppose we want to have the user input values to be used in queries:
  - Consider these relations:
    CUSTOMER (SSN, NAME, ADDR, CITY, STATE)
    ACCOUNTS (SSN, ACC_CODE, ACCNUM, BRANCHNUM, AMOUNT)
  - Suppose we want the user to input a customer name and obtain the amounts in his/her accounts.
  - In SQL, for customer `Ray` we would write this query as:

    select A.AMOUNT
    from CUSTOMER C, ACCOUNTS A
    where A.SSN=C.SSN and C.NAME = 'Ray';

  - However, in our C program we want to be able to *read in* the value 'Ray' from the keyboard:

    ```
    char * cname;
    printf ("Enter customer name: ");
    gets ( cname );
    /* We need to use cname in the SQL statement */
    ```

- Defining and using *placeholders* in Oracle OCI:
  - Oracle allows SQL statements to contain so-called *placeholders* or temporary variables.
  - These placeholders can be aliased to a C variable which can contain desired values.
  - The aliasing is done using either the obndrv or obndrn function calls.
  - For the above example:

```
text * sqlquery
  = (text *) "select A.AMOUNT \
               from ACCOUNTS A, CUSTOMER C \
               where A.SSN=C.SSN and C.NAME= :1";

#define CNAME_LEN 50
text cname[CNAME_LEN];     /* Customer name */
  sb2 ind_cname;
sword amount;               /* Amount */
  sb2 ind_amount;  ub2 retc_amount;  ub2 retl_amount;

/* Connect, open a cursor and parse the SQL statement */
connect ();
oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1)
oparse (&cda, sqlquery, -1, FALSE, VERSION_7)

/* Use obndrn to bind cname to ":1" */
obndrn (&cda, 1, (ub1*) cname, CNAME_LEN,
        SQLT_STR, -1, &ind_cname, (text*) 0, -1, -1);

/* Alias the output */
odefin (&cda, 1, (ub1*) &amount, (sword) sizeof(int), SQLT_INT,
        -1, &ind_amount, (text *) 0, -1, -1,
        &retl_amount, &retc_amount);
```

```
 /* Now read in the customer name */
 printf ("Enter customer name: ");  gets ( (char*) cname);

oexec (&cda);

/* Fetch tuples from result */
do {
  ofetch (&cda);
  if (cda.rc == NO_DATA_FOUND) break;
  printf ("%s %d \n", cname, amount);
} while (1);
```

- Note that an SQL placeholder is a string *preceded* by a colon.
- SQL placeholders are of two types:
  1. *Numeric placeholders*: these are integers (starting from 1), e.g.,

     ```
     text * sqlquery
       = (text *) "select C.NAME \
                   from CUSTOMER C \
                   where C.NAME= :1 and C.CITY = :2";
     ```

     Numeric placeholders are always *bound* using the `obndrn` call, e.g., for the above example:

     ```
     #define CNAME_LEN 50
     #define TOWN_LEN 50
     text cname[CNAME_LEN];
     text ccity[CITY_LEN];
     obndrn (&cda, 1, (ub1*) cname, CNAME_LEN,
             SQLT_STR, -1, &ind_cname, (text*) 0, -1, -1);

     obndrn (&cda, 2, (ub1*) ccity, CITY_LEN,
             SQLT_STR, -1, &ind_ccity, (text*) 0, -1, -1);
     ```

2. *String placeholders*: these are strings used for identification (and have nothing to do with C variables).

   ```
   text * sqlquery
     = (text *) "select C.NAME \
                 from CUSTOMER C \
                 where C.NAME= :cname and C.CITY = :town";
   ```

   String placeholders are always bound using `obndrv`:

   ```
   #define CNAME_LEN 50
   #define TOWN_LEN 50
   text cname[CNAME_LEN];
   text ccity[CITY_LEN];
   obndrv (&cda, (ub1*) ":cname", -1, (ub1*) cname, CNAME_LEN,
            SQLT_STR, -1, &ind_cname, (text*) 0, -1, -1);

   obndrv (&cda, (ub1*) ":town", -1, (ub1*) ccity, CITY_LEN,
            SQLT_STR, -1, &ind_ccity, (text*) 0, -1, -1);
   ```

– Let's take a look at `obndrv`:

```
obndrv (
  &cda,                /* The cursor reference */
  (ub1*) ":city",      /* Placeholder string */
  -1,                  /* If not null-delimited */
  (ub1*) ccity,        /* Address of C variable */
  CITY_LEN,            /* Space allocated */
  SQLT_STR,            /* Variable type */
  -1,                  /* Not used */
  &ind_ccity,          /* For return code */
  (text*) 0, -1, -1);  /* Not used */
```

– An example of using placeholders can be found in `~simha/321/oracle/test2.c`.

- OCI summary:

- Oracle OCI is a library of C functions that let you pass in SQL statements for execution.
- Use `olog, ologof` to set up a connection.
- Use `oopen` and `oclose` to open and close cursors.
- Use `odefin` to SQL results into C variables.
- Use `obndrv` and `obndrn` to use C variables as part of SQL statements.
- Use `oparse` and `oexec` to parse and execute a query.
- Use `ofetch` to retrieve one tuple at a time from the results.
- Other functions are available – see manual.

8.11 Embedded SQL: Using the Oracle C Precompiler

- An alternative to using an SQL library is to use a *precompiler* and *embed* SQL in a C program.
- The precompiler is a *translator* provided by Oracle that converts a C program with embedded SQL into a C program with function calls.
- Embedded SQL makes it easy to declare C variables for input and output.
- Embedded SQL allows SQL's datatypes to be declared like C variables, e.g.,

```
varchar cname[50];   /* An SQL type */
char custname[50];   /* Standard C type */
```

- Embedded SQL makes it easy to declare and fetch from cursors, e.g.,

```
EXEC SQL DECLARE cust_cursor CURSOR FOR
  SELECT C.NAME, A.AMOUNT
  FROM ACCOUNTS A, CUSTOMER C
  WHERE A.SSN=C.SSN and C.NAME = :custname;

EXEC SQL OPEN cust_cursor;
```

- Let's examine a complete program:

  This program reads in a customer's name and prints out the amounts owned by this customer.

```
#include <stdio.h>
#include <string.h>
#include "sqlca.h"   /* You need this include */

#define     UNAME_LEN      20
#define     PWD_LEN        20
varchar username[UNAME_LEN];   /* Note the varchar data type */
varchar password[PWD_LEN];
char usern[UNAME_LEN];
char passw[PWD_LEN];

/* We will obtain these attributes from a query */
varchar cname[50];   /* Corresponds to SQL's VARCHAR */
int amount;          /* Corresponds to SQL's INTEGER */

char custname[50];

/* Must declare error handling function. */
void sql_error();

main()
{

    printf ("username: ");  gets( (char*) usern);
    printf ("password: ");  gets( (char*) passw);
    /* Connect to ORACLE--
     * Copy the username into the VARCHAR.
     */
    strncpy((char *) username.arr, usern, UNAME_LEN);

    /* Set the length component of the VARCHAR. */
    username.len = strlen((char *) username.arr);
```

```
/* NOTE the structure of varchar: fields .arr and .len */

/* Copy the password. */
strncpy((char *) password.arr, passw, PWD_LEN);
password.len = strlen((char *) password.arr);

/* Register sql_error() as the error handler. */
EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--\n");

/* Connect to ORACLE.  Program will call sql_error()
 * if an error occurs when connecting to the default database.
 */
EXEC SQL CONNECT :username IDENTIFIED BY :password;

printf("\nConnected to ORACLE as user: %s\n", username.arr);

printf("\nEnter customer name: ");
gets(custname);
printf ("Customer name entered: %s\n", custname);

/* NOTE the use of a C variable as placeholder */
EXEC SQL DECLARE cust_cursor CURSOR FOR
  SELECT C.NAME, A.AMOUNT
  FROM simha.ACCOUNTS A, simha.CUSTOMER C
  WHERE A.SSN=C.SSN and C.NAME = :custname;

EXEC SQL OPEN cust_cursor;

printf ("Amounts owned by Customer %s: \n", custname);

for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH cust_cursor
    INTO :cname, :amount;
```

```
        /* NOTE the availability of Oracle datatypes for use in C */
        printf ("%s %4d\n", cname.arr, amount);
      }

      EXEC SQL CLOSE cust_cursor;

      EXEC SQL COMMIT WORK RELEASE;
      exit(0);
  }


  void sql_error (char *msg)
  {
      char err_msg[128];
      size_t buf_len, msg_len;

      EXEC SQL WHENEVER SQLERROR CONTINUE;

      printf("\n%s\n", msg);
      buf_len = sizeof (err_msg);
      sqlglm(err_msg, &buf_len, &msg_len);
      printf("%.*s\n", msg_len, err_msg);

      EXEC SQL ROLLBACK RELEASE;
      exit(1);
  }
```

- NOTE:
  - While not always necessary, it's best to enclose those variables that store parts of tuples in an embedded SQL block:

```
    EXEC SQL BEGIN DECLARE SECTION;
      varchar cname[50];
```

```
  int amount;
EXEC SQL END DECLARE SECTION;
```

– Pointers to character arrays can be used instead of declaring a fixed size of type varchar:

```
EXEC SQL BEGIN DECLARE SECTION;
  char *cname;
  int amount;
EXEC SQL END DECLARE SECTION;

char *custname;
...
custname = (char *) malloc (50*sizeof(char));
...
EXEC SQL DECLARE cust_cursor CURSOR FOR
  SELECT C.NAME, A.AMOUNT
  FROM simha.ACCOUNTS A, simha.CUSTOMER C
  WHERE A.SSN=C.SSN and C.NAME = :custname;
  EXEC SQL OPEN cust_cursor;

for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH cust_cursor
    INTO :cname, :amount;
  printf ("%4d\n", amount);
}
...
```

However, it's best to use varchar where possible.

– It is often inefficient to fetch only a tuple at a time.
  - Consider the following relation:

    EMP (NAME, SSN, AGE)

    with millions of tuples.

- Each tuple fetch may occur across a network (client-server mode)
  ⇒ very time-consuming.
- It's better to retrieve *batches* of tuples.
- First, declare an array (of your desired batch size) of variables to receive the output of the SQL statement:

```
EXEC SQL BEGIN DECLARE SECTION;
  varchar ename[20][50];  /* Batch size 20, ename size is 50
  varchar ssn[20][12];
  int age[20];
EXEC SQL END DECLARE SECTION;
```

  NOTE: arrays of pointers and multidimensional arrays are not allowed.
- Next, declare the cursor:

```
EXEC SQL DECLARE emp_cursor FOR
  SELECT E.NAME, E.SSN, E.AGE
  FROM EMP E;
```

- Each fetch from the cursor will retrieve a batch:

```
for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH emp_cursor;
}
```

- It is possible that fewer than 20 items are retrieved
  ⇒ could be near end of table.
- The precompiler-defined variable `sqlca.sqlerrd[2]` keeps track of the *number of tuples returned so far*.
- Thus, the following method tells you how many rows were returned:

```
n = 0;
for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
```

```
    EXEC SQL FETCH emp_cursor;
    rows_returned = sqlca.sqlerrd[2] - n;
    n = sqlca.sqlerrd[2];
  }
```

  - Note: batches can be used for *insert*'s, *update*'s and *delete*'s as well.

- Compilation and execution:
  - The `C/C++` precompiler in Oracle is called `proc` (for Pro-C).
  - The suffix used for embedded SQL (in C) programs is `.pc`, for example, `testpc1.pc`.
  - Now, as part of compilation, a C file will be generated. Thus, if you have a C file called `test1.c` then DO NOT name your Pro-C file test1.pc
    ⇒ compilation will overwrite your test1.c file.
  - Consider the file `testpc1.pc`.
  - You can compile this file as follows:

    ```
    % proc testpc1.pc
    ```

    However, you are likely to get linking errors.
  - Oracle recommends that you use their pre-defined *makefile* for compilation.
  - Copy over their `proc.mk` makefile and use it as:

    ```
    % make -f proc.mk EXE=testpc1 OBJ=testpc1.o
    ```

  - Observe that several files are produced as the result of compilation:
    * `testpc1.c` – this is a plain C file which has the embedded SQL replaced by incomprehsible C code.
    * `testpc1.o` – an object file.
    * `testpc1` – an executable.

  - Unfortunately, you don't get much help with errors.

    Two kinds of compilation errors are produced:

    1. Pre-compiler errors: these are likely errors in the embedded SQL.
    2. C compiler errors: errors found in the "intermediate" C file `testpc1.c`.

- What does the mysterious "intermediate" file look like?

  For example, the declaration

```
   varchar cname[50];
```

  becomes

```
  /* varchar cname[50]; */
  struct { unsigned short len; unsigned char arr[50]; } cname;
```

  Similarly, the statement:

```
   EXEC SQL OPEN cust_cursor
```

  becomes

```
    struct sqlexd sqlstm;
    sqlstm.sqlvsn = 8;
    sqlstm.arrsiz = 3;
    sqlstm.stmt = sq0002;
    sqlstm.iters = (unsigned int  )1;
    sqlstm.offset = (unsigned int  )28;
    ...
    sqlcxt((void **)0, &sqlctx, &sqlstm, &sqlfpn);
    if (sqlca.sqlcode < 0) sql_error("ORACLE error--\n");
```

Thus, it has been translated to the function call `sqlcxt`.

- Embedded SQL in C: a summary
  - SQL statements can be embedded in a C program.
  - C variables can be used to determine values in SQL statements.
  - C variables can be used to receive the results of SQL queries.
  - A precompiler translates the source into a regular C program, which is then compiled by a regular C compiler.

8.12 Useful SQL Functions in Oracle

- Oracle provides a number of useful functions to manipulate various data types.
- Type `help functions` in `sqlplus` for descriptions.
- *Numeric functions*:

  Consider the following relation:

  ```
  create table PATIENT ( NAME      VARCHAR(50),
                         AGE       INTEGER,
                         WEIGHT    FLOAT,
                         HEIGHT    FLOAT,
                         BALANCE   NUMBER(8,2) );
  ```

  The following numeric functions can be used: `ceil, floor, mod, power, round,`

  – Example:

  ```
  select  round (P.WEIGHT, 2)
  from    PATIENT P;
  ```

  – Example:

  ```
  select  P.NAME, sign (P.BALANCE)
  from    PATIENT P;
  ```

  – Example:

  ```
  select  P.NAME, P.WEIGHT, P.HEIGHT
  from    PATIENT P
  where   floor (P.WEIGHT) > 10;
  ```

- *Character functions*:

  The following functions operate on strings and return strings:

- `initcap`: capitalize first letter of string, e.g.,

```
select  initcap (E.NAME)
from    EMP E;
```

- `replace(strA, strB, strC)`: Replace every occurence of string `strB` with `strC` in the string `strA`, e.g.,

```
select  replace (E.POSITION,'man','person')
from    EMP E
where   E.POSITION = 'Chairman';
```

- `soundex`: for phonetic searches:

```
select  E.NAME
from    EMP E
where   soundex (E.NAME) = soundex ('Meyer');
```

  (Should match: Maier, Meier, Mayer, Meyer).

- Other functions of interest: `lower, substr, length`.

• *Date functions*:

Several date-related functions are useful:

- `sysdate` – current date and time, e.g.,

```
select  L.NAME, L.BOOK
from    LIBRARY_BOOKS L
where   L.DUE_DATE < sysdate
```

- `last_day` – last day of the month, e.g.,

```
select  L.NAME, L.BOOK
from    LIBRARY_BOOKS L
where   last_day (L.DUE_DATE) < sysdate;
```

- `months_between` – difference in months betwen 2 dates, e.g.,

```
select  L.NAME, L.BOOK
from    LIBRARY_BOOKS L
where   months_between (L.DUE_DATE), sysdate) > 3;
```

• Conversion functions: to convert from one data type to another.

- `to_char` – converts from *any* data type to a character string, e.g.

  ```
  select  M.TITLE, M.YEAR
  from    MOVIES M
  where   M.TITLE = to_char (10)
  ```

- `to_date` – converts *character* data in the specified format to a date, e.g.,

  ```
  select  P.NAME
  from    PATIENTS P
  where   P.BIRTHDAY = to_date (24-Jan-66,'DD)
  ```

  The function **to_date** figures out the *day of the week* for 24-Jan-66 for use in the comparison.

Chapter 9

Summary

Course Notes on Database Systems

9.1 CS 321: A Summary

- What did we learn in this course?

  Concepts:

  - What is a database? How is it different from a simple file system?
    ⇒ File system does not provide recovery, concurrency, fast querying etc.
  - Data Models.
    ⇒ The Relational Model (tables), constraints.
  - Relational databases: relational algebra.
    ⇒ σ, Π, $\bowtie$ etc.
  - Relational databases: SQL.
    ⇒ **select** statement, **create table**, **update** etc.
  - Example: Oracle.
    ⇒ Structure of Oracle, using SQLPlus for SQL queries.
  - Database programming in Oracle.
    ⇒ Using the C-library (OCI) and embedded SQL.
  - Physical implementation: file structures.
    ⇒ Heapfiles, hashfiles, sorted files, disk I/O.
  - B-trees and B+-trees
    ⇒ Insertion, search and deletion.
  - Hashing
    ⇒ Extendible hashing, linear hashing.
  - Sorting
    ⇒ Binary merge-sort, polyphase merge-sort, snowplow method.
  - Query processing
    ⇒ Developing a query plan, evaluating different plans.

  - Database design: normalization
    ⇒ Theory of Functional Dependencies (FD's), normal forms: 2NF, 3NF, BCNF
  - Recovery
    ⇒ Shadow paging, Log-based recovery, **redo**'s and **undo**'s.
  - Concurrency
    ⇒ Problems with concurrent execution, serializability, locking, deadlock.

- Advanced topics we have not covered:
  - Database systems (of interest to dbase system developers):
    - Details of implementing a transaction manager
      ⇒ Including code for locking, recovery, buffer management.
    - Advanced query processing
      ⇒ Query plan enumeration, transformation heuristics.
    - Disk subsystem implementation
      ⇒ Disk I/O, partitions and extents, directory management.
    - Parallel databases
      ⇒ parallel algorithms for implementing relational operators.
    - Distributed databases
      ⇒ Concurrency control and recovery in a distributed system.
  - Database theory:
    - Proofs for results on functional dependencies and normal forms.
    - Higher normal forms.
    - Detailed algorithms for decomposition and minimal covers.
    - logic databases, deductive databases.
  - Applications development:
    - Generating reports.
    - Oracle Report, combining forms and report.
    - Client-server application development, web-based interfaces.
    - ODBC and standard interfaces to other database systems.

    - JDBC and Java-related interfaces.
- Text processing and document retrieval:
    - String and pattern search algorithms
⇒ string search, approximate search, regular expression search.
    - Indices for text data, keyword-searching.
    - Organizing document databases
⇒ indices, clustering, thesaurus construction.
    - Data compression
⇒ Huffman coding, Liv-Zempel algorithm.
- Other data models:
    - Network and hierarchical models.
    - Object-oriented databases, persistent objects.
    - Object-relational databases.
- Non-traditional databases:
    - Geographic and spatial databases.
⇒ Storage and retrieval of maps, geographic queries.
    - Image and multimedia databases.
⇒ Storage and retrieval of image, audio and video data.
    - Scientific databases
⇒ Databases for CAD, astronomy, DNA and medical applications.
    - Temporal databases
⇒ history-related queries.
- New business applications:
    - Data mining.
    - Online Analytical Processing (OLAP) and multidimensional databases.

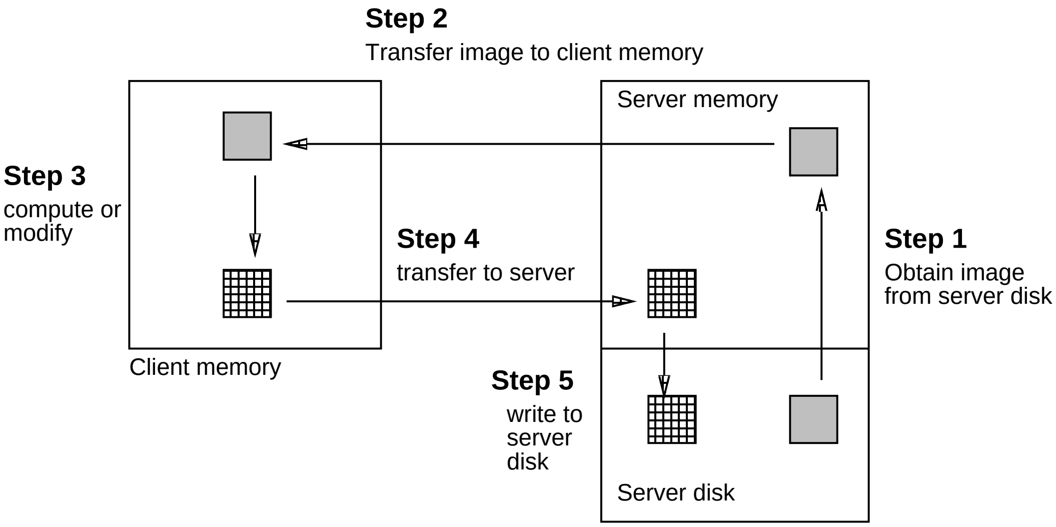
9.2 Object-Oriented and Object-Relational Databases

- The development of OO (Object-Oriented) and OR (Object-Relational) databases is motivated by:
  - Supporting new data types: image, spatial and multimedia data.
  - Providing applications developers with mechanisms to define their own data types, their own operators.
  - Allowing for efficient implementation of the above.
  - Integrating a dbase-like file system (with recovery and concurrency) into a high-level programming language (like C++).
  - Providing applications programmers with powerful OO features (inheritance, encapsulation).
- Consider supporting image data:
  - We want to allow users to store and retrieve images in a variety of formats (GIF, Postscript, Xbitmap, JPEG, etc).
  - We want to allow some kinds of querying, e.g., to determine if two images are of approximately equal intensity.
  - Currently, most commercial systems have a data type for binary (raw bits) data.
  - Thus, an image can be stored as follows:

```
create table  IMAGE_TABLE (  bit       IMAGE,
                             varchar   NAME,
                             char(3)   TYPE,
                             number    CODE);
```

  - Now, a C program can be written (as a client) to retrieve images and work with them (e.g., to compute mean intensity etc).

- Unfortunately, this "compute-at-the-client" design causes lots of data to move back and forth between client and server memory:



- Thus, a query that searches through the whole dbase of images will cause a lot of data transfer
  ⇒ better to have all the work done at the server.
- Some database systems provide SQL-like functions to support non-traditional data.
  - These are usually very limited (e.g., only very few data types).
  - Also, there is no programmer flexibility in defining new types and functions.

- Object-Oriented Databases.

  OODB vendors take the view that:

  - Object-oriented programming is here to stay (e.g., C++, Smalltalk, Java).
  - Objects are the natural way to treat all types of data.
  - If database models can't handle objects, then it's time to change database software.

- There are three schools of thought on this matter:

- *Pure OO*:
  - ∗ Forget the concerns of dbase folks and instead simply design persistence into a programming language, e.g., persistent C++.
  - ∗ What does *persistence* mean? All objects that are written into are stored in an object repository and can be retrieved later.
  - ∗ Dbase vendors then can use a persistent language to develop dbase-specific libraries for applications programmers.
  - ∗ Do not provide a database server.
- *OODB*:
  - ∗ Since pure-OO folks don't really care about dbase concerns (such as recovery and concurrency), design a dbase-specific object repository.
  - ∗ Provide an interface via an object-oriented language but provide a full-fledged database server.
  - ∗ Allow applications programmers to program assuming persistent objects.
  - ∗ Handle recovery and concurrency (and efficient memory management).
  - ∗ Provide a library for SQL programmers (parser, interpreter, query optimizer etc).
- *Object-Relational*:
  - ∗ Relational systems aren't broken – let's instead amend them to handle newer data types.
  - ∗ Allow programmers to define new data types and functions that manipulate the data types.
  - ∗ The system then loads programmer-defined functions *into* the server (incorporating it as server code) to handle the new types.

• Object-Oriented Databases (OODB):

- In a typical implemention, an application programmer will define an object using an Object Definition Language, e.g.,

```
object definition my_object {
    private data:
        int size; // For a size×size image
        real image[100][100];
    methods:
        int get_size ();
        set_size (int val);
        real** get_image ();
        set_image (real** I);
}
```

- The programmer then compiles this using a special compiler for the language.
- The compiler produces C++ output (in a `.h` file) that looks like the above:

```
class my_object {
  private:
    int size;
    float image[100][100]
  public:
    int get_size ();
    void set_size (int);
    float ** get_image ();
    void set_image (float** I);
};
```

- Then, the programmer is expected to write code for the functions in the class.
- Finally, the system compiles the code for use when the object is manipulated.
- An applications programmer then defines the object using an encapsulation (usually via templates in C++):

```
encap<my_object> A;
A.set_size (5);  // Not allowed (won't work)
A.update->set_size(5) // Works. Goes via the encapsulation.
```

The trick is to define the encapsulation so that it returns a pointer (usually a `const` pointer in C++). Then, an applications programmer cannot modify the data
⇒ must go via a member function of the encapsulation instead
⇒ server has control over modification.

- Since updates go via the OO-system, the data remains at the server, allowing for recovery and concurrency control.
- Example: The SHORE (Scalable Heterogeneous Object REpository) System, developed at the University of Wisconsin, Madison.

- The *Object-Relational* approach:
  - The programmer is allowed to create types and new functions.
  - Creating a type:
    * Suppose we want to create an image type called `image`.
    * First, create the image type in C:

```
typedef struct image {
  int size;
  float image[100][100];
} image;
```

    * Then, create two C functions that handle I/O for this new data type:

```
image* image_in (char *data)
{
  // Code to take in raw data and put it in the struct
}

char* image_out (image* I)
```

```
{
  // Code to take in an image and put out char data, e.g
}
```

and compile them.

- Now, tell the database what you've done:

```
CREATE FUNCTION image_in (char*)
  RETURNS image
  AS '/usr/mom3/joe/321/image.o'
  LANGUAGE 'C';

CREATE FUNCTION image_out (image)
  RETURNS char*
  AS '/usr/mom3/joe/321/image.o'
  LANGUAGE 'C';
```

- Next, create the type in the database:

```
 CREATE TYPE image ( size = 10002,
                     input = image_in,
                     output = image_out);
```

- Finally, create manipulation functions and declare them.
- After all of this is done, an applications programmer can use the new data type in a query:

```
SELECT name
FROM image_file
WHERE approx_equals (image_file.image, :test_image);
```

– Example: Postgres95 (University of California, Berkeley), Illustra (part of Informix), Oracle 8.

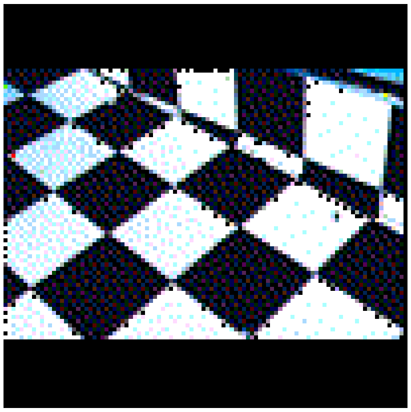
9.3 Spatial Information Systems: Introduction

- Standard relational databases store and manipulate alphanumeric data:
  - Vast majority of data consists of simple ASCII text.
  - Numbers (reals, integers) are stored using standard numeric representations.
  - Data sizes are most often known in advance.
  - Data is carefully organized:
    - At high-level: relations.
    - At low-level: file systems, indices.
- Newer applications demand the ability to store different kinds of data, e.g., Image data, Video data, Geometric data, Scientific data.
- Image data:
  - Application example: X-rays in a medical database.
  - Data type: 2D images (e.g., JPEG images).
  - Typical queries:
    - "Retrieve Smith's shoulder X-rays"
    - "Find all X-rays similar to this one"
- Video data:
  - Application example: Sports.
  - Data type: Video clips in a suitable format (e.g., MPEG).
  - Typical queries:
    - "Find all scenes in which Michael Jordan scores over 3 defenders".
    - "Find all scenes containing Olympic pole vaults".

- Geometric data:
  - Application example: CAD database.
  - Data type: points, lines, polygons.
  - Typical queries:
    - (Architecture) “Find all floor plans similar to this one”.
    - (VLSI) “Find all chip layouts which use less than 10 square microns of area”.
  - Application example: Geographic Information System (GIS).
  - Typical queries:
    - “Find the nearest hospital to this school”.
    - “Find all towns and counties covered by the hurricane path”.
- Scientific data:
  - Application example: data recorded from weather satellites and probes.
  - Data type: vectors of numbers.
  - Typical queries:
    - “Find all previous hurricanes with similar air-pressure patterns.”
    - “Find all measurements with this pressure-temperature combination”.
- Other kinds of non-standard data:
  - Financial data (stock prices, macroeconomic indicators, etc).
  - 3D data (terrain and contour maps).
  - Audio data (ultrasound, music, sonar).
  - Temporal data (time-specific data).
  - Typical query: “Find similar stock price patterns in history” (given a particular 1-week trend).

- A *spatial information system*:
  - stores and manipulates spatial data:
    - Image data: usually gray-scale or color images.
    - Geometric data: usually points, lines, polygons and polygonal maps.
  - supports queries related to spatial information:
    - distance-related (e.g., “nearest hospital”).
    - spatial relationships (e.g., “Find lakes Northwest of Richmond”).
  - also allows annotation using text and searching using text.
- Image and geometric data are quite different:
  - Image data:
    - usually larger in size (must store each pixel).
    - easy to display.
    - easy to obtain (via photography).
    - difficult to extract information from.
    - querying by annotation is easy (using text annotation), e.g., “Find Smith’s X-rays”.
    - querying by context is hard, e.g., “Does this fingerprint exist in the dbase?”
  - Geometric data:
    - usually smaller in size (only need to store endpoints of lines).
    - special rendering software needed (graphics).
    - difficult to obtain (usually requires manual intervention).
    - easier to extract information from
      ⇒ searching is easier.

- Current state-of-the-art in image databases:
  - Examples: IBM's QBIC, MIT Multimedia Lab (Photobook), Virage Corp., several medical dbases.
  - Query by annotation (easy).
  - Query by content (hard, varies with vendor).
    - Queries based on physical characteristics are a little easier, "Find all images that are 40% white and 40% black"
      ⇒ use color histograms.
      A sample from QBIC:



    - Similarity searches based on similarity metrics, e.g., "Find all images similar to this one".
    - Object recognition is hard (manual annotation is often needed).
- Current state-of-the-art in geometric databases:
  - Examples: ArcInfo, Intergraph, U.S. Census (TIGER) project.
  - Query by annotation (easy).
  - Several geometric queries are efficiently implemented:
    - queries based on distance.
    - object relationships (directional and topological).
    - range queries (using a query rectangle).
    - spatial joins.

- Note: there's a big difference between implementing a query and implementing it efficiently:
  - Example: color-percentage example above.
  - Wrong way:
    - search all images.
    - compute color percentages for each image.
    - test condition for each image.
  - Right way:
    - use an index.
    - the index results in a narrowing down the list of images to test.

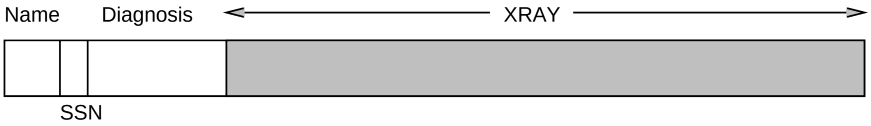
    ⇒ a key problem in spatial dbases: designing useful indices for various kinds of data.
- Issues in spatial information systems:
  - Data storage:
    - Store images in compressed or uncompressed form?
    - Store maps as collections of line segments or store individual polygons separately?
  - Display:
    - Image display tool (like `xv`) needed.
    - Map display tool needed.
  - Input/conversion:
    - conversion of raster data to geometric data and vice-versa.
    - extraction of features.
    - correlation with text data.
  - Indexing.
  - High-level issues:
    - How to integrate with relational system.
    - Query languages.

- Query optimization.

- Why relational dbases are inadequate for spatial queries:
  - Consider storing an image as an attribute, e.g, a 256×256 image:

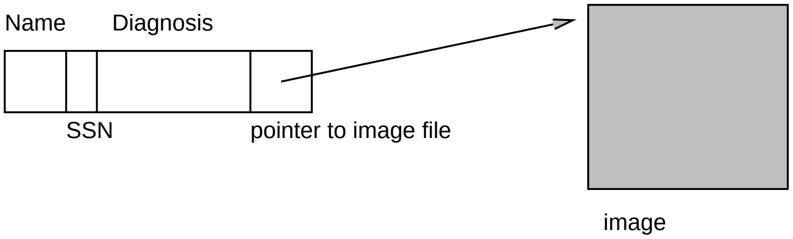
```
create table PATIENT (  NAME        char(20),
                        SSN         char(11),
                        DIAGNOSIS   char(100),
                        XRAY        image(256,256) );
```

  - One option is to store each image with each tuple:



This will make the file very large
⇒ standard relational operations will be slow (even text-only queries)

  - Another option: store images separately and use a pointer:



  - While storage issues are straightforward, the real problem is querying
    ⇒ can't use SQL for querying by content.
  - One solution: use C API to implement image-specific queries
    ⇒ a common solution (image query tool needed).
  - Another solution: modify relation system to allow for non-standard data and new functions
    ⇒ object-relational approach.
  - Discard relational approach altogether
    ⇒ object-oriented dbases.

– Another example: consider storing geometric data, e.g.,

```
create table CITY (  NAME      char(20),
                     XVALUE    real,
                     YVALUE    real );
```

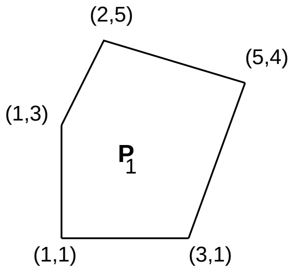
– Then, the query

```
select  NAME
from    CITY C1, CITY C2
where   C1.NAME=‘Richmond’ and
        (sqr(C1.XVALUE-C2.XVALUE)+sqr(C1.YVALUE-C2.YVALUE)) < 10000;
```

finds all cities within 100 miles of Richmond.

– However, more complicated queries are not possible in SQL (e.g., intersection and containment queries).

– Storing polygons as collections of lines (an old approach) can be inefficient, e.g.,



In a relational dbase, the polygon would typically stored as:

POLYGON	NAME	LINE_NUM	LINE_ID
	P_1	1	37
	P_1	2	43
	P_1	3	44
	P_1	4	47
	P_1	5	12

LINE	LINE_ID	STARTX	STARTY	ENDX	ENDY
			⋮		
	12	1	3	1	1
			⋮		
	37	1	1	3	1
	43	3	1	5	4
	44	5	4	2	5
	47	2	5	1	3

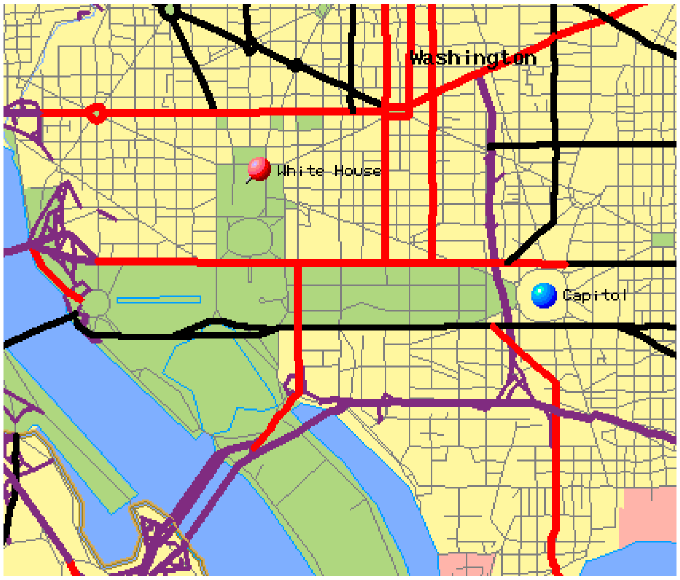
This is inefficient for many reasons:

- Duplication of points (only the vertices of the polygon are needed).
- Any processing of the polygon will have to use the LINE relation ⇒ if it's large, disk I/O needed.
- Repeated access of the POLYGON relation to retrieve the line segments itself takes up CPU time.

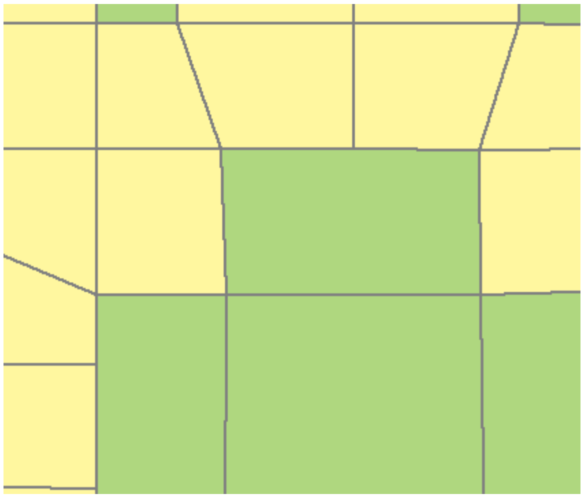
9.4 Geometric Databases: Introduction

- Geometric databases form the core of Geometric Information Systems (GIS)
  ⇒ most examples will concern GIS.
- Data types:
  - 0-D: points
    - ∗ Points are specified using a (planar) coordinate system.
    - ∗ In a GIS points are cross-referenced to latitude/longitude.
    - ∗ Points are used for landmarks, for cities/towns etc at large scales.
    - ∗ Points are used for query specification.
  - 1-D: line segments and lines:
    - ∗ Line segments are specified by end points.
    - ∗ Lines are sequences of line segments (with common end points).
    - ∗ Lines are used to represent road systems, rivers and boundaries.
  - 2-D: polygons, polygonal maps and networks:
    - ∗ A polygon is an ordered collection of line segments with exactly one point in common between successive line segments.
    - ∗ A simple polygon has intersections only between successive line segments.
    - ∗ Polygons are used to represent regions (countries, states, lakes etc).
    - ∗ A polygonal map is a collection of polygons.
    - ∗ Polygonal maps are used for complex regions, e.g., a country map showing states.
    - ∗ A network is a Euclidean graph: a collection of vertices some of which have an edge between them.

    - Networks are used to represent road maps, river maps and transportation systems.
- Other geometric objects:
    - Special polygons: e.g., triangles, rectangles.
    - Circles, ellipses.
    - Text labels.
- Example: Consider the following map of downtown Washington, DC (taken from the TIGER Census Project):



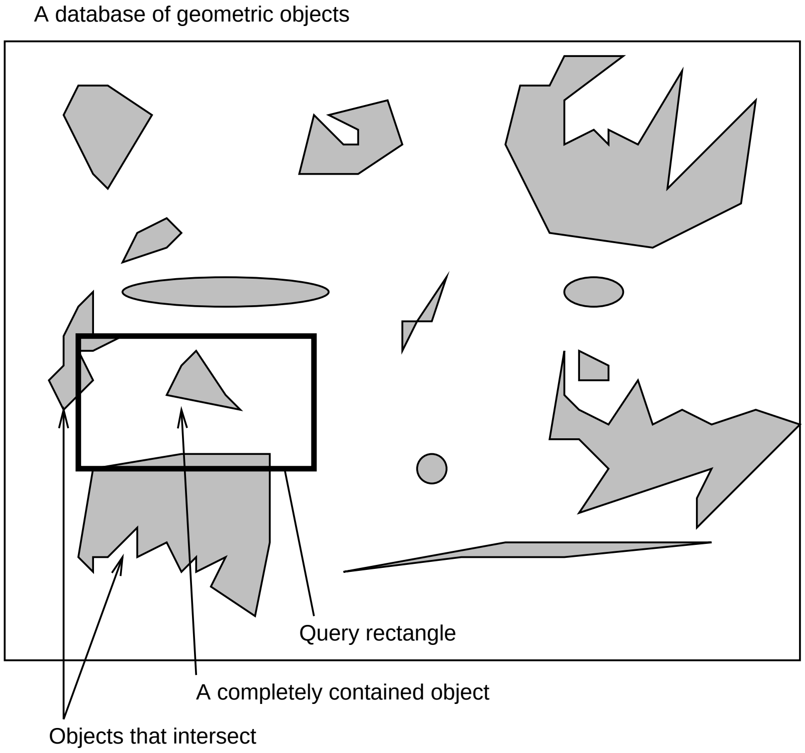
A close-up view of the area near the White House reveals:



Note the representation using line segments.

- Formal theory underlying geometric dbases:
  - (Euclidean) Geometry.
  - Computational Geometry: efficient algorithms for manipulating geometric objects.
  - Topology (to a lesser extent): formal theory of spatial properties.
- Storage issues:
  - Points and line segments have fixed-size storage.
  - For polygons, store list of vertices in order (using, say, clockwise order)
    $\Rightarrow$ one option: use heapfile with variable-size records.
  - Note: Large polygons may cross block boundaries
    $\Rightarrow$ need a robust file system.
  - Most systems today allow the storage of BLOBs (Binary Large OBjects) for images, polygons etc.
  - Some dbases create specific types, e.g., Postgres has a `polygon` type.

- Types of queries:
  - *Range queries*:
    - Example: “Find all lakes withing 100 miles of Richmond”.
    - Example: “Retrieve all objects that intersect with or are contained in a specified query rectangle.


A database of geometric objects
Query rectangle
A completely contained object
Objects that intersect

  - *Nearest neighbor queries*:
    - Example: “For each elementary school, find the nearest hospital.”
    - Distances are usually Euclidean distances, but may also be street/driving distances.
  - *Spatial joins*:
    - Example: Given two files of spatial objects, find all pairs of objects, one from each file, that intersect.

- Indices:
  - Several types of indices exist: for point data, for polygonal data.
  - Indices greatly speed up access.
  - Example: consider a file with 10,000 polygonal objects and a query rectangle:
    - Naive approach: scan through all objects and compute intersection with query rectangle.
    - Better: use a spatial index.
  - Key problem in spatial indexing: index should capture "spatialness" of data.
- An important difference between standard relational databases and spatial databases:
  - In standard relational databases, we pretty much ignored CPU costs in processing data
    $\Rightarrow$ only focused on I/O costs.
  - In spatial databases, CPU costs are also significant.
  - Example: Suppose we want to report intersection points of two polygons:
    - One approach: test each segment of first polygon against each segment of the other.
    - If both polygons have 1000 segments each
      $\Rightarrow$ 10^6 intersection line intersection tests
      $\Rightarrow$ can be very time consuming!

Chapter 10

Spatial Information Systems

Course Notes on Database Systems

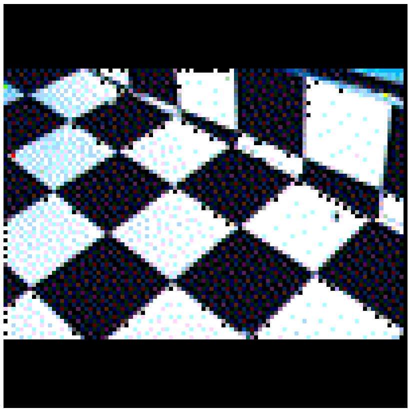
10.1 Spatial Information Systems: Introduction

- Standard relational databases store and manipulate alphanumeric data:
  - Vast majority of data consists of simple ASCII text.
  - Numbers (reals, integers) are stored using standard numeric representations.
  - Data sizes are most often known in advance.
  - Data is carefully organized:
    - At high-level: relations.
    - At low-level: file systems, indices.
- Newer applications demand the ability to store different kinds of data, e.g., Image data, Video data, Geometric data, Scientific data.
- Image data:
  - Application example: X-rays in a medical database.
  - Data type: 2D images (e.g., JPEG images).
  - Typical queries:
    - "Retrieve Smith's shoulder X-rays"
    - "Find all X-rays similar to this one"
- Video data:
  - Application example: Sports.
  - Data type: Video clips in a suitable format (e.g., MPEG).
  - Typical queries:
    - "Find all scenes in which Michael Jordan scores over 3 defenders".
    - "Find all scenes containing Olympic pole vaults".

- Geometric data:
  - Application example: CAD database.
  - Data type: points, lines, polygons.
  - Typical queries:
    - (Architecture) “Find all floor plans similar to this one”.
    - (VLSI) “Find all chip layouts which use less than 10 square microns of area”.
  - Application example: Geographic Information System (GIS).
  - Typical queries:
    - “Find the nearest hospital to this school”.
    - “Find all towns and counties covered by the hurricane path”.
- Scientific data:
  - Application example: data recorded from weather satellites and probes.
  - Data type: vectors of numbers.
  - Typical queries:
    - “Find all previous hurricanes with similar air-pressure patterns.”
    - “Find all measurements with this pressure-temperature combination”.
- Other kinds of non-standard data:
  - Financial data (stock prices, macroeconomic indicators, etc).
  - 3D data (terrain and contour maps).
  - Audio data (ultrasound, music, sonar).
  - Temporal data (time-specific data).
  - Typical query: “Find similar stock price patterns in history” (given a particular 1-week trend).

- A *spatial information system*:
  - stores and manipulates spatial data:
    - Image data: usually gray-scale or color images.
    - Geometric data: usually points, lines, polygons and polygonal maps.
  - supports queries related to spatial information:
    - distance-related (e.g., "nearest hospital").
    - spatial relationships (e.g., "Find lakes Northwest of Richmond").
  - also allows annotation using text and searching using text.
- Image and geometric data are quite different:
  - Image data:
    - usually larger in size (must store each pixel).
    - easy to display.
    - easy to obtain (via photography).
    - difficult to extract information from.
    - querying by annotation is easy (using text annotation), e.g., "Find Smith's X-rays".
    - querying by context is hard, e.g., "Does this fingerprint exist in the dbase?"
  - Geometric data:
    - usually smaller in size (only need to store endpoints of lines).
    - special rendering software needed (graphics).
    - difficult to obtain (usually requires manual intervention).
    - easier to extract information from
      ⇒ searching is easier.

- Current state-of-the-art in image databases:
  - Examples: IBM's QBIC, MIT Multimedia Lab (Photobook), Virage Corp., several medical dbases.
  - Query by annotation (easy).
  - Query by content (hard, varies with vendor).
    - Queries based on physical characteristics are a little easier, "Find all images that are 40% white and 40% black"
      ⇒ use color histograms.
      A sample from QBIC:



    - Similarity searches based on similarity metrics, e.g., "Find all images similar to this one".
    - Object recognition is hard (manual annotation is often needed).
- Current state-of-the-art in geometric databases:
  - Examples: ArcInfo, Intergraph, U.S. Census (TIGER) project.
  - Query by annotation (easy).
  - Several geometric queries are efficiently implemented:
    - queries based on distance.
    - object relationships (directional and topological).
    - range queries (using a query rectangle).
    - spatial joins.

- Note: there's a big difference between implementing a query and implementing it efficiently:
  - Example: color-percentage example above.
  - Wrong way:
    - search all images.
    - compute color percentages for each image.
    - test condition for each image.
  - Right way:
    - use an index.
    - the index results in a narrowing down the list of images to test.

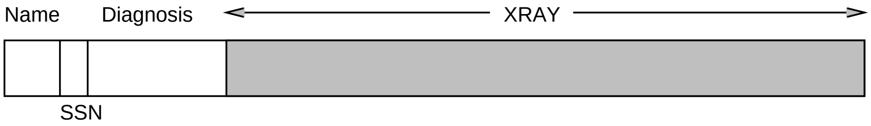
    ⇒ a key problem in spatial dbases: designing useful indices for various kinds of data.
- Issues in spatial information systems:
  - Data storage:
    - Store images in compressed or uncompressed form?
    - Store maps as collections of line segments or store individual polygons separately?
  - Display:
    - Image display tool (like `xv`) needed.
    - Map display tool needed.
  - Input/conversion:
    - conversion of raster data to geometric data and vice-versa.
    - extraction of features.
    - correlation with text data.
  - Indexing.
  - High-level issues:
    - How to integrate with relational system.
    - Query languages.

        - ∗ Query optimization.

- Why relational dbases are inadequate for spatial queries:
    - Consider storing an image as an attribute, e.g, a 256×256 image:

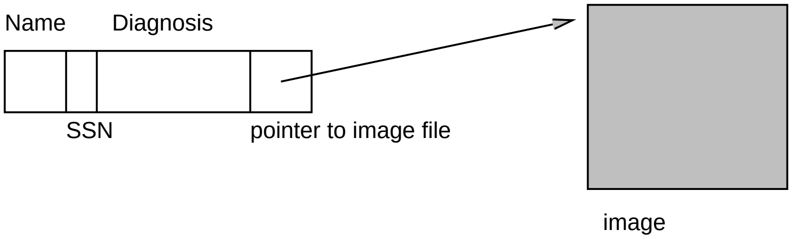
```
create table PATIENT (  NAME        char(20),
                        SSN         char(11),
                        DIAGNOSIS   char(100),
                        XRAY        image(256,256) );
```

    - One option is to store each image with each tuple:



      This will make the file very large
      ⇒ standard relational operations will be slow (even text-only queries)

    - Another option: store images separately and use a pointer:



    - While storage issues are straightforward, the real problem is querying
      ⇒ can't use SQL for querying by content.
    - One solution: use C API to implement image-specific queries
      ⇒ a common solution (image query tool needed).
    - Another solution: modify relation system to allow for non-standard data and new functions
      ⇒ object-relational approach.
    - Discard relational approach altogether
      ⇒ object-oriented dbases.

– Another example: consider storing geometric data, e.g.,

```
create table CITY (  NAME     char(20),
                     XVALUE   real,
                     YVALUE   real );
```

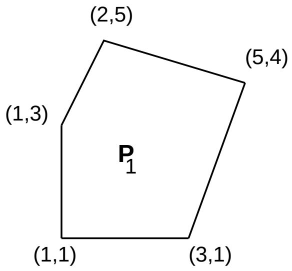
– Then, the query

```
select  NAME
from    CITY C1, CITY C2
where   C1.NAME=‘Richmond’ and
        (sqr(C1.XVALUE-C2.XVALUE)+sqr(C1.YVALUE-C2.YVALUE)) < 10000;
```

finds all cities within 100 miles of Richmond.

– However, more complicated queries are not possible in SQL (e.g., intersection and containment queries).

– Storing polygons as collections of lines (an old approach) can be inefficient, e.g.,



In a relational dbase, the polygon would typically be stored as:

POLYGON	NAME	LINE_NUM	LINE_ID
	P_1	1	37
	P_1	2	43
	P_1	3	44
	P_1	4	47
	P_1	5	12

LINE	LINE_ID	STARTX	STARTY	ENDX	ENDY
			⋮		
	12	1	3	1	1
			⋮		
	37	1	1	3	1
	43	3	1	5	4
	44	5	4	2	5
	47	2	5	1	3

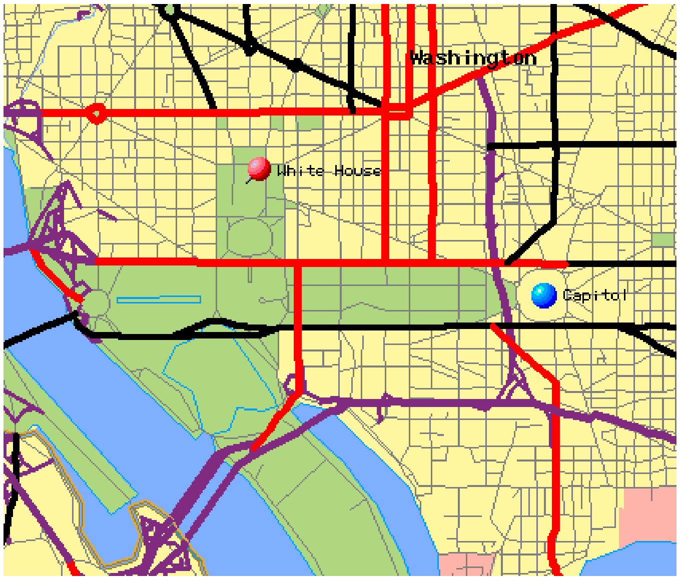
This is inefficient for many reasons:

- Duplication of points (only the vertices of the polygon are needed).
- Any processing of the polygon will have to use the LINE relation ⇒ if it's large, disk I/O needed.
- Repeated access of the POLYGON relation to retrieve the line segments itself takes up CPU time.

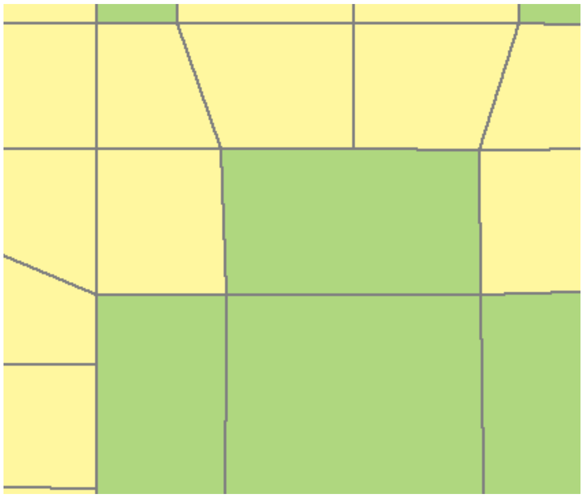
10.2 Geometric Databases: Introduction

- Geometric databases form the core of Geometric Information Systems (GIS)
  ⇒ most examples will concern GIS.
- Data types:
  - 0-D: points
    - Points are specified using a (planar) coordinate system.
    - In a GIS points are cross-referenced to latitude/longitude.
    - Points are used for landmarks, for cities/towns etc at large scales.
    - Points are used for query specification.
  - 1-D: line segments and lines:
    - Line segments are specified by end points.
    - Lines are sequences of line segments (with common end points).
    - Lines are used to represent road systems, rivers and boundaries.
  - 2-D: polygons, polygonal maps and networks:
    - A polygon is an ordered collection of line segments with exactly one point in common between successive line segments.
    - A simple polygon has intersections only between successive line segments.
    - Polygons are used to represent regions (countries, states, lakes etc).
    - A polygonal map is a collection of polygons.
    - Polygonal maps are used for complex regions, e.g., a country map showing states.
    - A network is a Euclidean graph: a collection of vertices some of which have an edge between them.

    - ∗ Networks are used to represent road maps, river maps and transportation systems.
- – Other geometric objects:
    - ∗ Special polygons: e.g., triangles, rectangles.
    - ∗ Circles, ellipses.
    - ∗ Text labels.
- – Example: Consider the following map of downtown Washington, DC (taken from the TIGER Census Project):



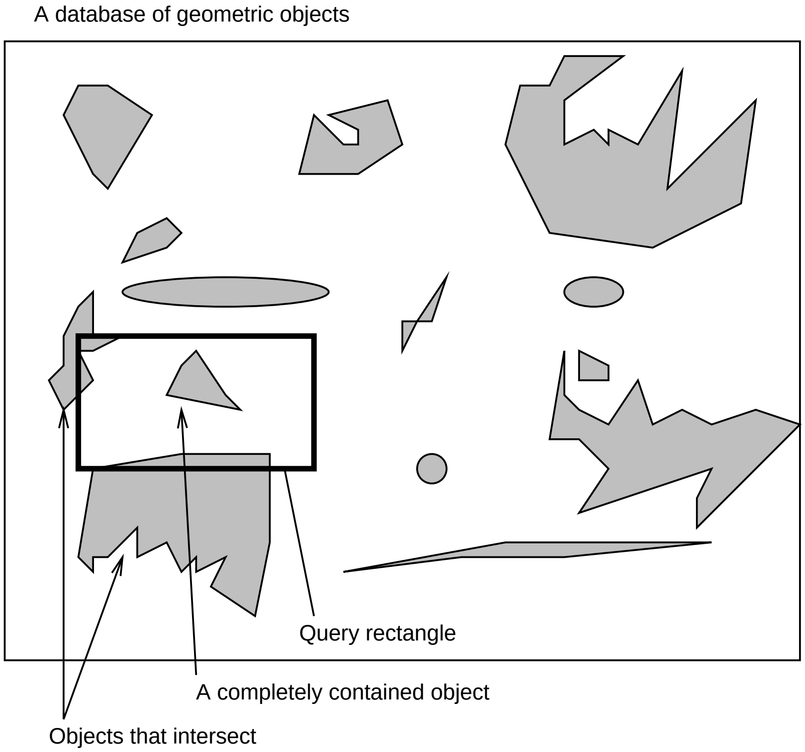
A close-up view of the area near the White House reveals:



Note the representation using line segments.

- Formal theory underlying geometric dbases:
  - (Euclidean) Geometry.
  - Computational Geometry: efficient algorithms for manipulating geometric objects.
  - Topology (to a lesser extent): formal theory of spatial properties.
- Storage issues:
  - Points and line segments have fixed-size storage.
  - For polygons, store list of vertices in order (using, say, clockwise order)
    ⇒ one option: use heapfile with variable-size records.
  - Note: Large polygons may cross block boundaries
    ⇒ need a robust file system.
  - Most systems today allow the storage of BLOBs (Binary Large OBjects) for images, polygons etc.
  - Some dbases create specific types, e.g., Postgres has a `polygon` type.

- Types of queries:
  - *Range queries*:
    - ∗ Example: “Find all lakes withing 100 miles of Richmond”.
    - ∗ Example: “Retrieve all objects that intersect with or are contained in a specified query rectangle.


A database of geometric objects
Query rectangle
A completely contained object
Objects that intersect

  - *Nearest neighbor queries*:
    - ∗ Example: “For each elementary school, find the nearest hospital.”
    - ∗ Distances are usually Euclidean distances, but may also be street/driving distances.
  - *Spatial joins*:
    - ∗ Example: Given two files of spatial objects, find all pairs of objects, one from each file, that intersect.

- Indices:
  - Several types of indices exist: for point data, for polygonal data.
  - Indices greatly speed up access.
  - Example: consider a file with 10,000 polygonal objects and a query rectangle:
    - Naive approach: scan through all objects and compute intersection with query rectangle.
    - Better: use a spatial index.
  - Key problem in spatial indexing: index should capture "spatialness" of data.

- An important difference between standard relational databases and spatial databases:
  - In standard relational databases, we pretty much ignored CPU costs in processing data
    $\Rightarrow$ only focused on I/O costs.
  - In spatial databases, CPU costs are also significant.
  - Example: Suppose we want to report intersection points of two polygons:
    - One approach: test each segment of first polygon against each segment of the other.
    - If both polygons have 1000 segments each
      $\Rightarrow$ 10^6 intersection line intersection tests
      $\Rightarrow$ can be very time consuming!

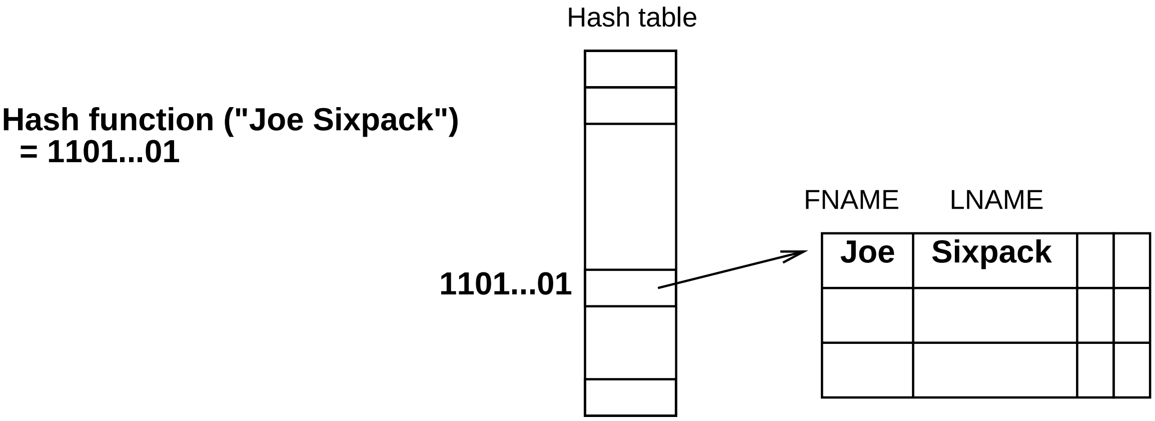
Chapter 11

Spatial Indices

Course Notes on Database Systems

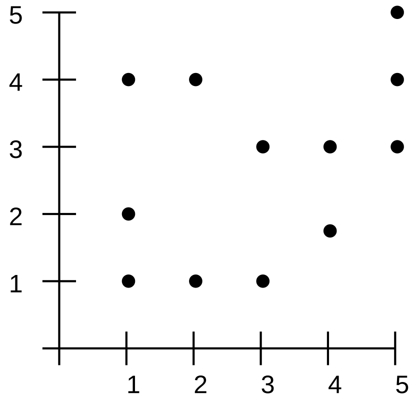
11.1 Peano Curves: Introduction

- A *Peano Curve Index* is a spatial index constructed from a unidimensional key (and often using a unidimensional index).
- Recall: unidimensional indices can actually be used for multiple attributes:
  - Example: Consider EMP (FNAME, LNAME, SSN, SALARY).
  - Suppose we want to handle queries on FNAME-LNAME values.
  - To answer equality queries, a hash table can be used on the concatenation of the two, e.g.,

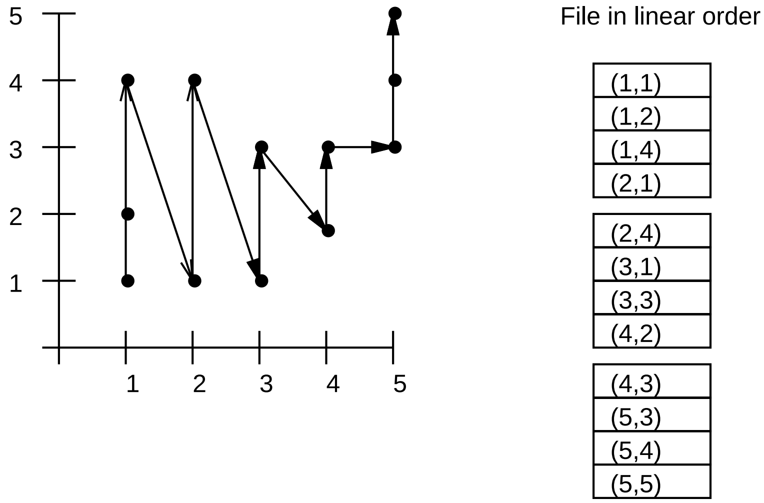


  - For range queries, a B+-tree on FNAME-LNAME can be used.
- Consider using a unidimensional index for point data:
  - Hash tables
    - ∗ Can be used for equality queries.
    - ∗ Useless for range or nearest-neighbor queries.

- What about B+-trees?
  - We'd have to pick some linear order in which to store the point data.
  - Example: consider the following point data: $(5,3)$, $(1,1)$, $(5,4)$, $(5,5)$, $(3,3)$, $(4,2)$, $(4,3)$, $(1,2)$, $(1,4)$, $(3,1)$, $(2,1)$, $(2,4)$.



  - Suppose we use *column-major* order:



File in linear order

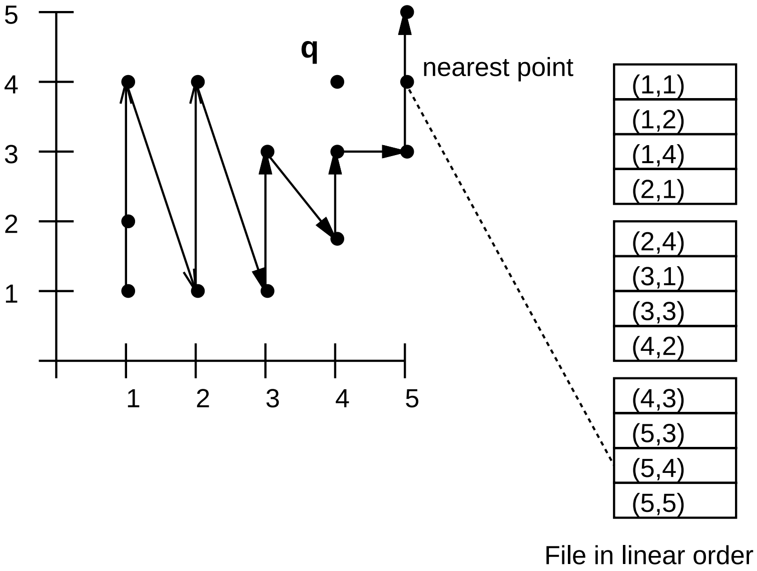
(1,1)
(1,2)
(1,4)
(2,1)

(2,4)
(3,1)
(3,3)
(4,2)

(4,3)
(5,3)
(5,4)
(5,5)

  - For a nearest-neighbor search, we would still have to search the whole file.

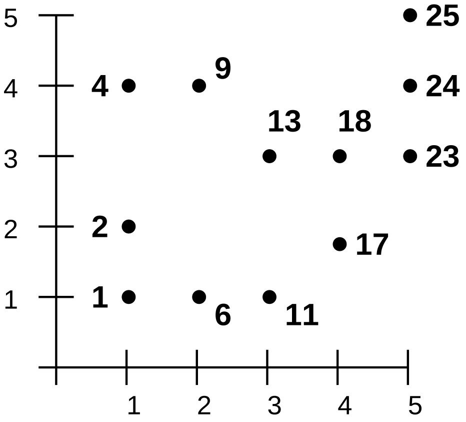
∗ Example: consider the query point $q = (4, 4)$:



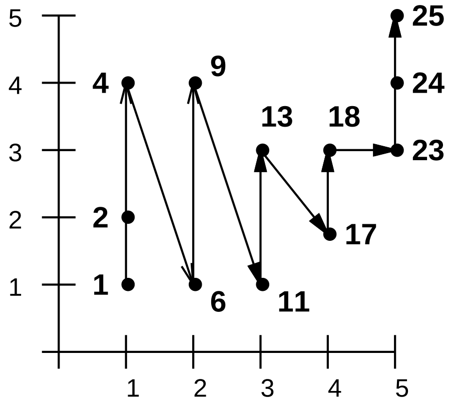
The nearest points are $(5, 4)$ and $(4, 3)$. But how do we locate, for example, $(5, 4)$ in the file quickly?

⇒ need to map $(4, 4)$ to $(5, 4)$'s location in file.

- Mapping a 2D-value to a 1D-value:
  - Consider the function $F(x, y) = 5(x - 1) + y$.
  - If we compute F for each of the points in the above example, we get:



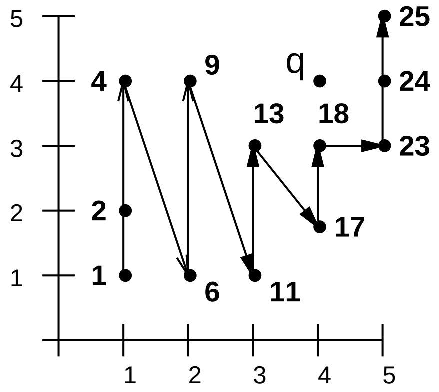
– Suppose now we store the points in order of F-value and include the F-value in each record:



point	F−value
(1,1)	1
(1,2)	2
(1,4)	4
(2,1)	6
(2,4)	9
(3,1)	11
(3,3)	13
(4,2)	17
(4,3)	18
(5,3)	23
(5,4)	24
(5,5)	25

– Notice: it's the same order as before.

– Consider the nearest-neighbor query $q = (4, 4)$:



point	F−value
(1,1)	1
(1,2)	2
(1,4)	4
(2,1)	6
(2,4)	9
(3,1)	11
(3,3)	13
(4,2)	17
(4,3)	18
(5,3)	23
(5,4)	24
(5,5)	25

sorted by F−value

F(q) = F(4,4) = 5*(4−1)+4 = 19

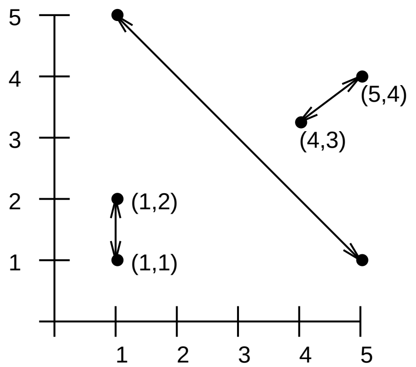
- $*$ Compute $F(q) = F(4, 4) = 19$.
- $*$ Use key=19 to search the sorted file using *binary search*
  $\Rightarrow$ search ends between keys 18 and 23.
- $*$ Use keys 18 and 23 as starting point for nearest-neighbor search.

- What other formulas (like $F(x, y) = 5(x-1) + y$) might be useful?
  - For nearest-neighbor queries, the following property would be useful: For points $a = (x_0, y_0)$ and $b = (x_1, y_1)$:

    $$x_0 \approx x_1 \text{ and } y_0 \approx y_1 \Rightarrow F(x_0, y_0) \approx F(x_1, y_1).$$

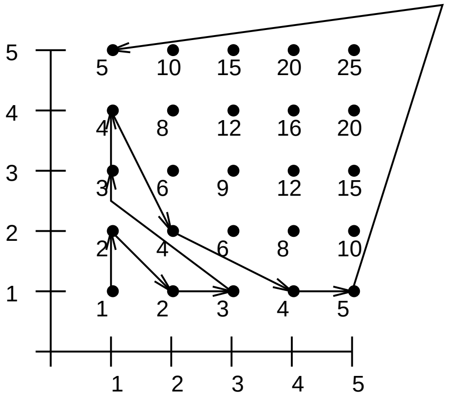
    i.e., if a "is close to" b, then $F(a)$ should be approximately equal to $F(b)$.
  - Can we design such a function?
  - Consider $F(x, y) = xy$ (product):
    - Rationale:

      $$\begin{array}{l} x_0, y_0 \text{ both large} \Rightarrow x_0y_0 \text{ large} \\ x_1, y_1 \text{ both large} \Rightarrow x_1y_1 \text{ large} \end{array}$$
    - Example: consider points (4,3) and (5,4) $\Rightarrow F(4, 3) = 12$ and $F(5, 4) = 20$.
    - Similarly for points (1,2) and (1,1): $\Rightarrow F(1, 2) = 2$ and $F(1, 1) = 1$.



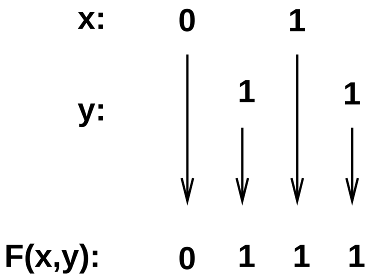
    - On the other hand, consider (5,1) and (1,5): $\Rightarrow F(5, 1) = 5 = F(1, 5)$, but they are not close.

- ∗ Let's join the dots in F-order:



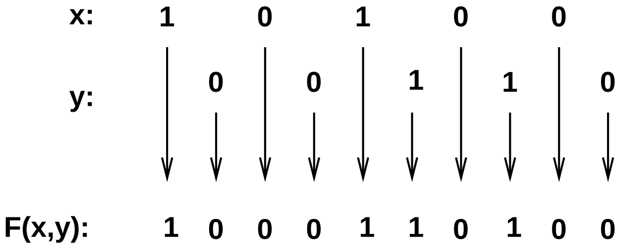
  $\Rightarrow$ the F-curve jumps all over the place.

- Consider the following unusual function:
  - This function is defined only on k-bit integers, in the range $0, \ldots, 2^k$ for some k (we can pick k as large as we want).
  - Define $F(x, y) =$ interleave x's and y's bits.
  - Example: consider 2-bit integer coordinates ($k = 2$).
    Suppose $x = 1$ and $y = 3$
    $\Rightarrow$ binary(x) = 01 and binary(y) = 11
    $\Rightarrow$ interleaved bitstring = 0111 (decimal: 7)
    $\Rightarrow$ $F(1, 3) = 7$.



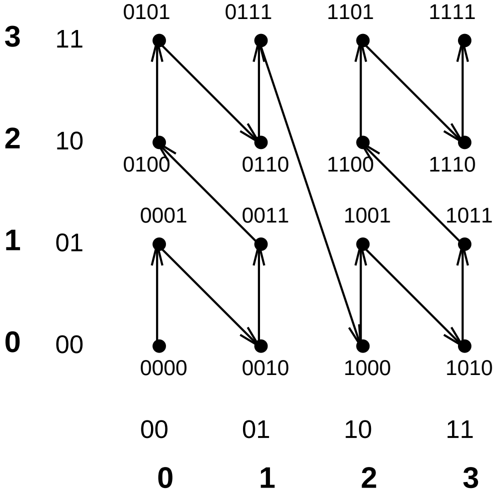
  - Example: consider 5-bit integer coordinates ($k = 5$).
    Suppose $x = 20$ and $y = 6$
    $\Rightarrow$ binary(x) = 10100 and binary(y) = 00110

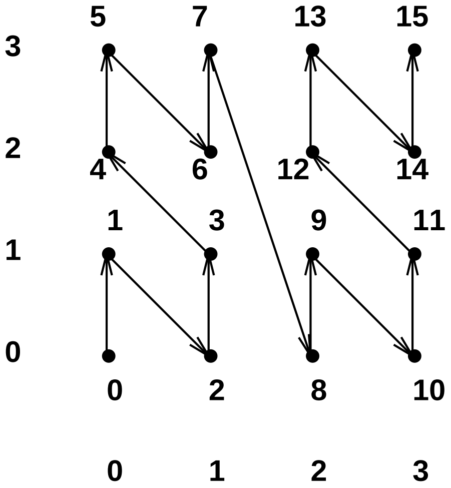
⇒ interleaved bitstring = 1000110100 (decimal: 564).
⇒ $F(20, 6) = 564$.



– Suppose we consider $k = 2$ and compute F for all possible points ⇒ 16 possible points.

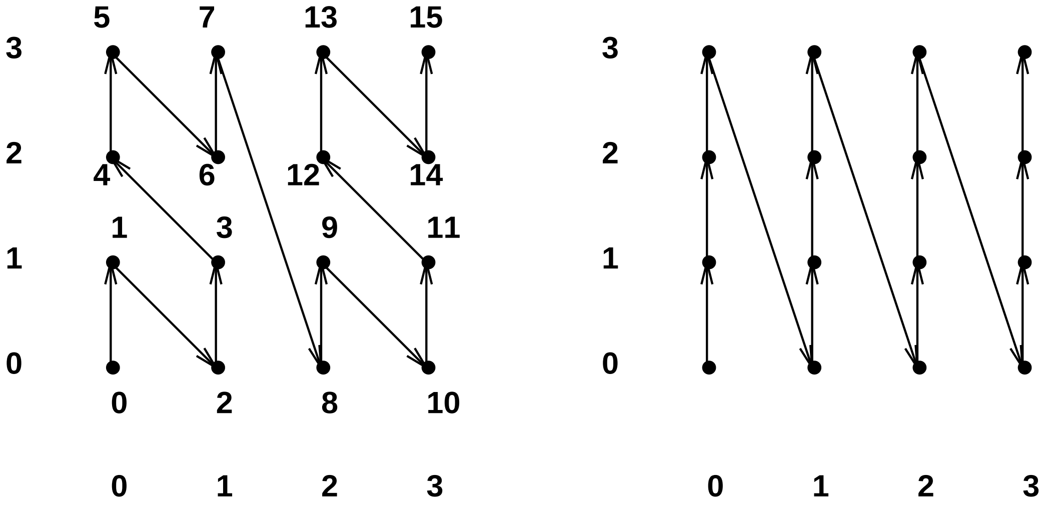
– Then, join the dots in F_Z order:





– This ordering method is the *Z-order*.

- Why should *Z-order* be better than, say, *column-major* order?



Z−order Column−major order

- Let $F_Z(x, y)$ denote the Z-value of a point (x, y) and let $F_C(x, y)$ denote the corresponding F-value using column-major ordering.
- Note: column-major ordering is: $F_C(x, y) = cx + y$ where c is the column size.
- Let's apply the following test to each order:
  - Pick a distance d, e.g., $d = 3$.
  - Set *count* := 0.
  - For each point p in the set:
    - Identify $N(p)$ = neighboring points of p.
    - For each point $q \in N(p)$, set *count* := *count* + 1, if $|F(p) - F(q)| \leq d$.
- Thus, we are counting how often a neighbor occurs within a short distance (d) along the linear ordering F.

- Here's a comparison of F_Z and F_C:

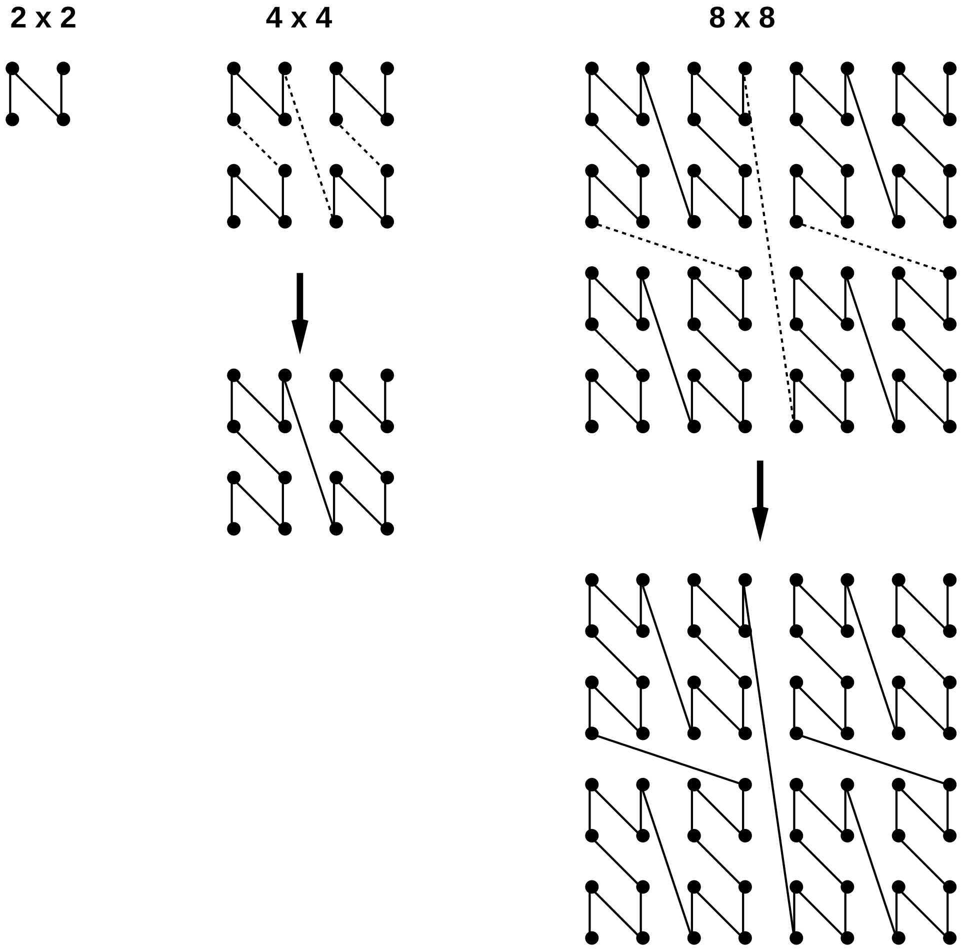
Size	d	Using F_C	Using F_Z
$k = 2$	1	24	28
	2	24	44
	3	42	62
$k = 3$	1	112	112
	2	112	176
	3	112	248

This example shows that the chances of finding spatial neighbor "close by" in the file are higher using the Z-order
⇒ point neighbors are more often clustered together in a file using F_Z than using F_C.

- Because of its self-similar nature, the Z-ordering often referred to as a *fractal* ordering. (Also called a space-filling curve – Hausdorff dimension = 2). Discovered by the mathematician Giuseppe Peano in the late 19th century.

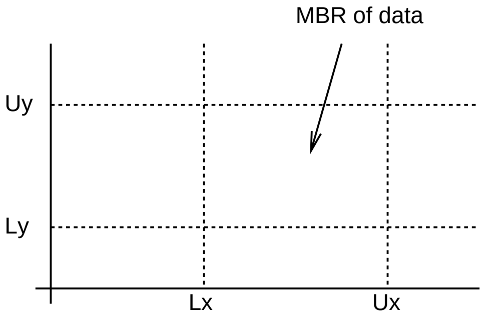
  Note: a $2^k \times 2^k$ curve is obtained from a $2^{k-1} \times 2^{k-1}$ curve by the following steps:

  - Make four copies and place them in a square.
  - Join appropriate points in the four copies (by adding three line segments).

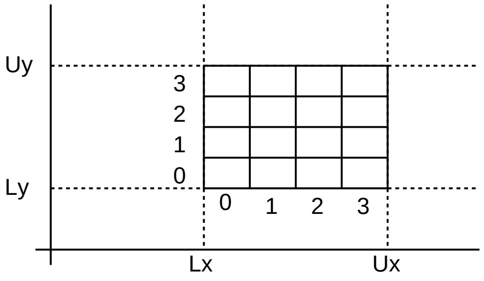

2 x 2
4 x 4
8 x 8

11.2 Peano Curve Indices

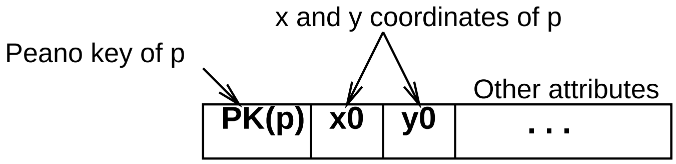
- So far, we've only dealt with integer-valued coordinates in a power-of-2 square.
- What's needed for a useful index:
  - The ability to handle arbitrary real-valued coordinates.
  - A mechanism for storing and searching the data.
  - A mechanism for nearest-neighbor queries.
- Key ideas:
  - We will assume MBR of data is known in advance:



  - Select a granularity for the Peano key
    ⇒ select a power-of-2 – 2^k.
  - Impose an imaginary grid on the MBR of size $2^k \times 2^k$, e.g., if $k = 2$:



  - Label rows and columns of grid $0, \ldots, 2^k - 1$
    $\Rightarrow$ each grid cell has an address $[i, j]$ where $0 \leq i, j \leq 2^k - 1$.
  - To compute the *Peano-key* of a point $p = (x_0, y_0)$:
    - Find the cell $[i, j]$ containing p.
    - Compute the Peano-value of $(i, j) \Rightarrow F(i, j)$.
    - Example: if Z-order is being used, interleave bits of i and j.
    - Return the Peano-value $F(i, j)$ as the Peano-key of p.
  - Note:
    - Points in the same cell have the same Peano-key
      $\Rightarrow$ we will allow duplicates
      $\Rightarrow$ it's not strictly a *key*.
    - Notation: use $PK(p)$ to denote the Peano-key of point p.
    - $PK(p)$ is an integer in the range $0, \ldots, 2^{2k} - 1$.
  - Create a tuple for point p and include p's Peano-key:



  - Insert tuple in a standard unidimensional index, such as a B+-tree using the Peano-key attribute as the search key.
  - To handle a nearest-neighbor query:
    - Compute the Peano-key of the query point q, $PK(q)$.
    - Use the search-value $PK(q)$ to search in the unidimensional index.
    - After finding closest (by PK-value) point in index, proceed with remainder of nearest-neighbor search.

- Notation: we will use
  - $PK_Z(p)$ to denote the Peano-key using Z-order.

  - $PK_C(p)$ to denote the Peano-key using column-major order.
  - $PK_H(p)$ to denote the Peano-key using *Hilbert order*.

- We distinguish between two kinds of settings:

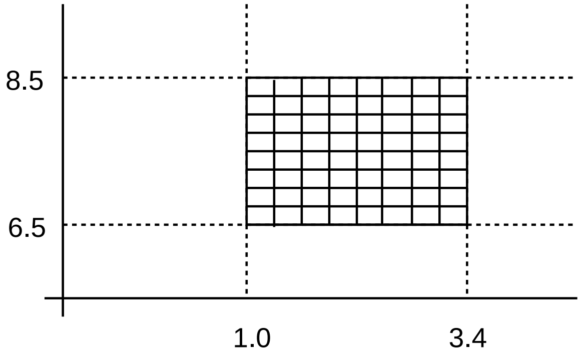
  1. Applications which require frequent insertions and deletions:
     - In this case, a B+-tree is used to store the tuples.
     - The leaf-level of the B+-tree is useful for range searches.
  2. Applications which do not require many insertions and deletions:
     - Usually, the data is a set of points that doesn't change much.
     - In this case, the tuples are placed in a sorted file
       ⇒ sorted by Peano-key.
     - Binary search is used for handling queries.

- Example:

  - Consider the following data set with MBR $L_x = 1.0, U_x = 3.4$ and $L_y = 6.5, U_y = 8.5$:

p_1	(1.4,7.9)	p_5	(2.0,8.1)	p_9	(1.4,6.6)
p_2	(1.1,6.9)	p_6	(2.7,7.3)	p_{10}	(1.7,6.8)
p_3	(2.1,7.1)	p_7	(1.8,7.6)	p_{11}	(1.7,7.7)
p_4	(1.1,7.4)	p_8	(3.3,6.8)	p_{12}	(3.2,7.6)

  - Suppose we use a $2^3 \times 2^3$ grid
    ⇒ cell addresses have 3-bit coordinates.



  - Note: size of cell is 0.3×0.25.

- We will use Z-order.

- Examples of Peano-key calculation:
  - $p_1 = (1.4, 7.9)$:
    - Compute x-coordinate of cell:

$$\left\lfloor \frac{1.4 - 1.0}{0.3} \right\rfloor = 1$$

    - Compute y-coordinate of cell:

$$\left\lfloor \frac{7.1 - 6.5}{0.25} \right\rfloor = 5$$

      $\Rightarrow$ Cell [1,5].
    - Compute Peano-key of [1,5]:
      - Binary(1) = 001 ($k = 3$ bits).
      - Binary(5) = 101.
      - Interleave: 010011 = 19.
      - Thus, $PK_Z(1.4, 7.9) = 19$.
  - $p_7 = (1.8, 7.6)$:
    - Compute x-coordinate of cell:

$$\left\lfloor \frac{1.8 - 1.0}{0.3} \right\rfloor = 2$$

    - Compute y-coordinate of cell:

$$\left\lfloor \frac{7.6 - 6.5}{0.25} \right\rfloor = 4$$

      $\Rightarrow$ Cell [2,4].
    - Compute Peano-key of [2,4]:
      - Binary(1) = 010 ($k = 3$ bits).

        - · Binary(5) = 100.
        - · Interleave: 011000 = 24.
        - · Thus, $PK_Z(1.8, 7.6) = 24$.
  - – $p_{11} = (1.7, 7.7)$:
    - ∗ Compute x-coordinate of cell:

$$\left\lfloor \frac{1.7 - 1.0}{0.3} \right\rfloor = 2$$

    - ∗ Compute y-coordinate of cell:

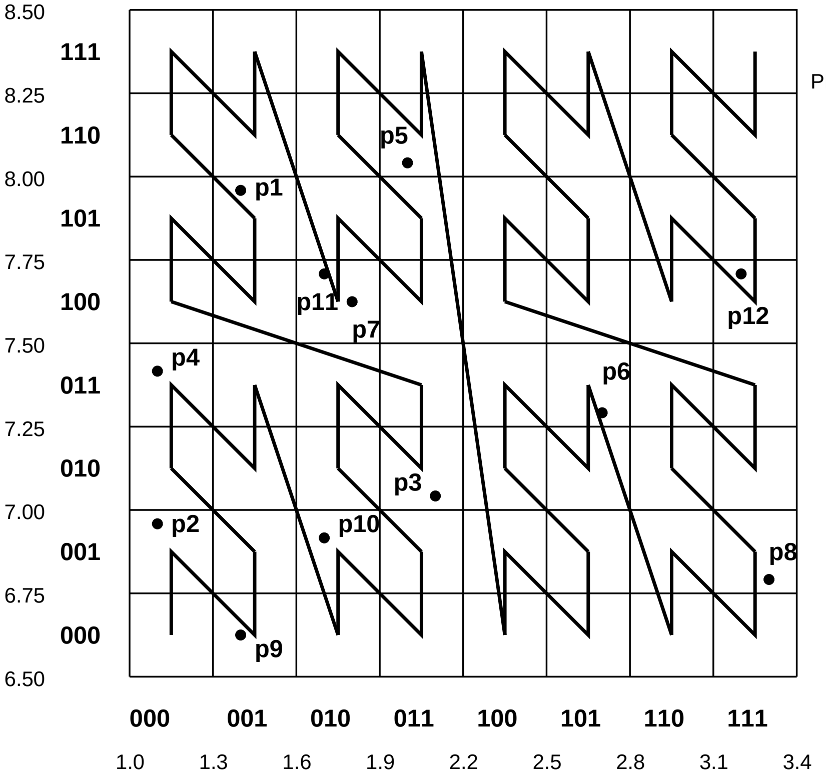
$$\left\lfloor \frac{7.7 - 6.5}{0.25} \right\rfloor = 4$$

      $\Rightarrow$ Cell [2,4].
    - ∗ Compute Peano-key of [2,4]:
        - · Binary(1) = 010 ($k = 3$ bits).
        - · Binary(5) = 100.
        - · Interleave: 011000 = 24.
        - · Thus, $PK_Z(1.7, 7, 7) = 24$.

- Similarly, we compute the Peano-key of the remaining points:

Point	(x, y)	$PK_Z(x, y)$	Point	$(x.y)$	$PK_Z(x, y)$	Point	(x, y)	$PK_Z(x, y)$
p_1	(1.4,7.9)	19	p_5	(2.0,8.1)	30	p_9	(1.4,6.6)	3
p_2	(1.1,6.9)	1	p_6	(2.7,7.3)	39	p_{10}	(1.7,6.8)	9
p_3	(2.1,7.1)	14	p_7	(1.8,7.6)	24	p_{11}	(1.7,7.7)	24
p_4	(1.1,7.4)	5	p_8	(3.3,6.8)	43	p_{12}	(3.2,7.6)	58

and create a sorted file:



PK−value	Coordinates		
1	**1.1, 6.9**		p2
3	**1.4, 6.6**		p9
5	**1.1, 7.4**		p4
9	**1.7, 6.8**		p10
14	**2.1, 7.1**		p3
19	**1.4, 7.9**		p1
24	**1.8, 7.6**		p7
24	**1.7, 7.7**		p11
30	**2.0, 8.1**		p5
39	**2.7, 7.3**		p6
43	**3.3, 6.8**		p8
58	**3.2, 7.6**		p12

Sorted file

- Nearest-neighbor query example: $q = (1.2, 8.1)$
  - Compute $PK_Z(q)$:
    - Compute x-coordinate of grid cell containing q:

$$\left\lfloor \frac{1.2 - 1.0}{0.3} \right\rfloor = 0$$

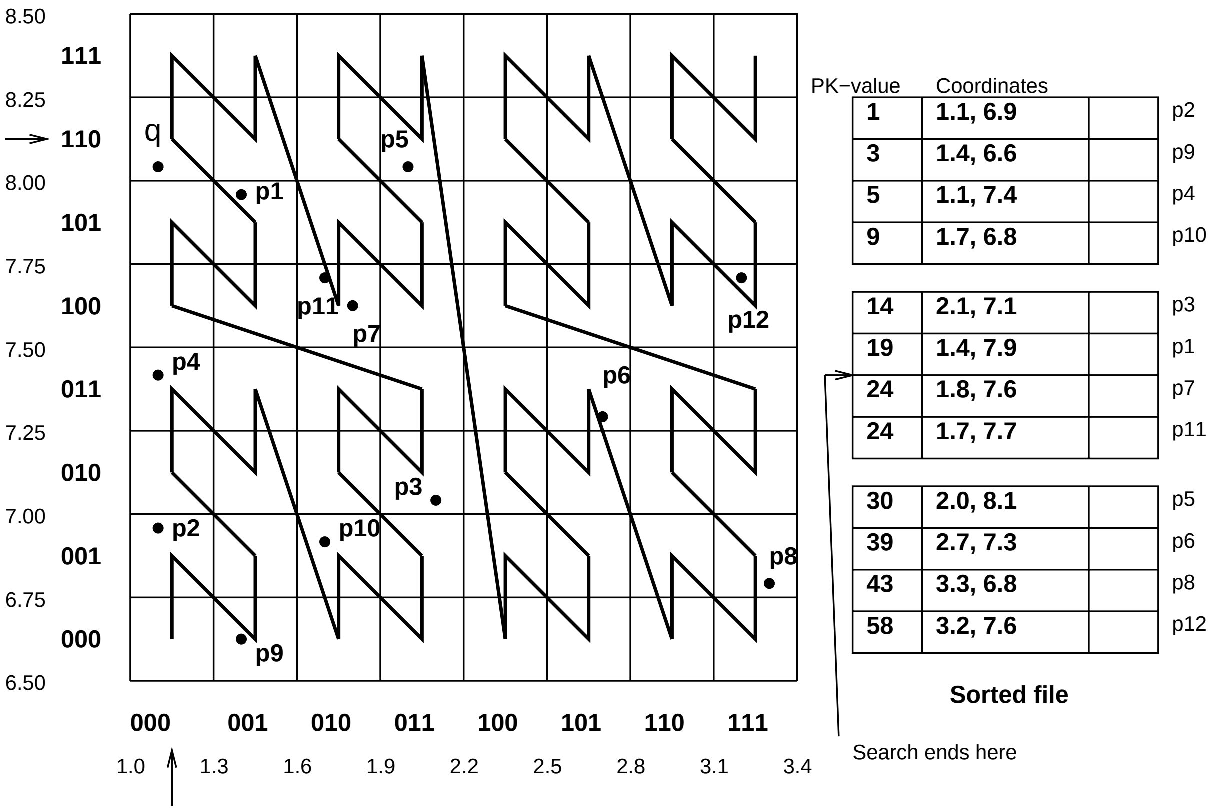
    - Compute y-coordinate of grid cell containing q:

$$\left\lfloor \frac{8.1 - 6.5}{0.25} \right\rfloor = 6$$

    - Compute Peano-key:
      - Binary$(0) = 000$.
      - Binary$(6) = 110$.
      - Interleave: 010100 (decimal: 20).

      $\Rightarrow PK_Z(q) = 20$.

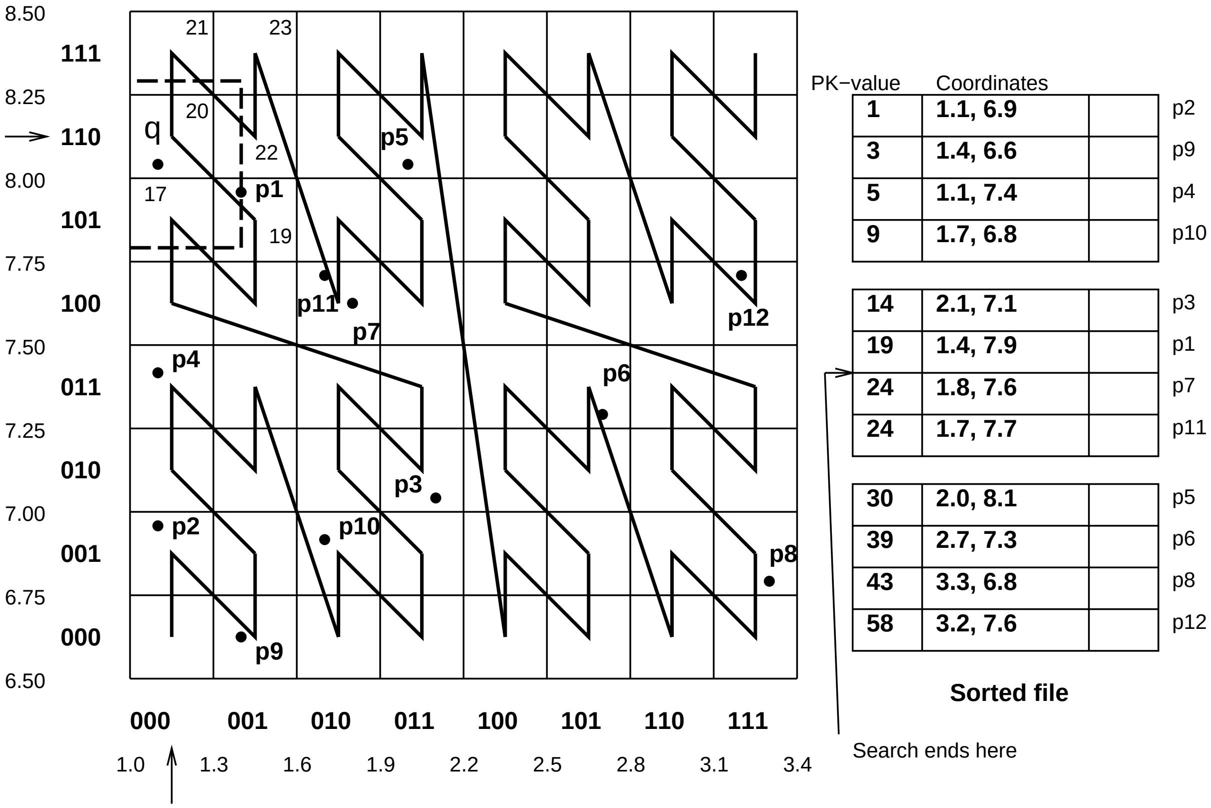
- Use $PK_Z(q) = 20$ to search in sorted file (binary search).



PK−value	Coordinates		
1	1.1, 6.9		p2
3	1.4, 6.6		p9
5	1.1, 7.4		p4
9	1.7, 6.8		p10
14	2.1, 7.1		p3
19	1.4, 7.9		p1
24	1.8, 7.6		p7
24	1.7, 7.7		p11
30	2.0, 8.1		p5
39	2.7, 7.3		p6
43	3.3, 6.8		p8
58	3.2, 7.6		p12

- Nearest Peano-keys in file:
  - $PK_Z(p_1) = 19$ in 2nd block.
  - $PK_Z(p_7) = 24$ in 2nd block.
  - $PK_Z(p_{11}) = 24$ in 2nd block (Must consider others with identical keys).
- Compute distances to these points:
  - $dist(q, p_1) = \left((1.2 - 1.4)^2 + (8.1 - 7.9)^2\right)^{1/2} = 0.28.$
  - $dist(q, p_7) = \left((1.2 - 1.8)^2 + (8.1 - 7.6)^2\right)^{1/2} = 0.78.$
  - $dist(q, p_{11}) = \left((1.2 - 1.7)^2 + (8.1 - 7.7)^2\right)^{1/2} = 0.64.$

  $\Rightarrow p_1$ is closest with distance $d = 0.28$.
- Draw an imaginary square with q as center and d as half-side:



PK−value	Coordinates		
1	1.1, 6.9		p2
3	1.4, 6.6		p9
5	1.1, 7.4		p4
9	1.7, 6.8		p10
14	2.1, 7.1		p3
19	1.4, 7.9		p1
24	1.8, 7.6		p7
24	1.7, 7.7		p11
30	2.0, 8.1		p5
39	2.7, 7.3		p6
43	3.3, 6.8		p8
58	3.2, 7.6		p12

Sorted file

– Find cells covered by square
  ⇒ [0,5], [1,5], [0,6], [1,6], [0,7], [1,7].

– Compute Peano-keys of these cells:

$$PK_Z(0,5) = 17$$
$$PK_Z(1,5) = 19$$
$$PK_Z(0,6) = 20$$
$$PK_Z(1,6) = 22$$
$$PK_Z(0,7) = 21$$
$$PK_Z(1,7) = 23$$

– Sort the above keys using this metric: the distance of each cell to the query point
  (Use closest corner of cell in computing distance).

– For each of the above Peano-keys in sort order:
  If distance of cell is greater than d, stop. Otherwise,

    - ∗ Search for the key in the sorted file (using binary search).
    - ∗ Find closest Peano keys to the key (in the file).
    - ∗ Compute distances to those points.
    - ∗ If any point's distance is less than d, use that point.
    - ∗ Set d to closest distance so far.
  - – In this example, since 2nd block contains all Peano keys in the range 14-24
    $\Rightarrow$ don't need to search further.
  - – Final result: nearest-neighbor of q is p_1.

- Nearest-neighbor query example: $q = (2.4, 7.1)$
  - – Compute $PK_Z(q)$:
    - ∗ Compute x-coordinate of grid cell containing q:

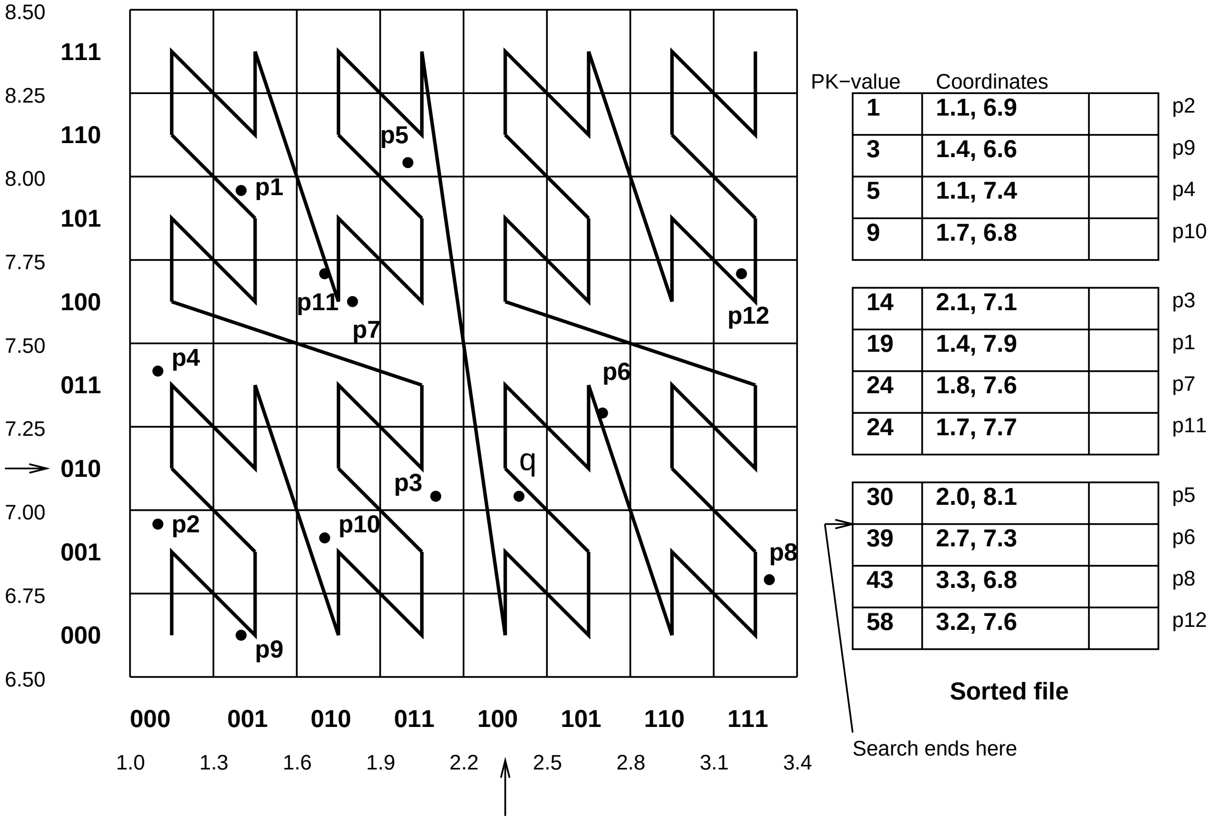
$$\left\lfloor \frac{2.4 - 1.0}{0.3} \right\rfloor = 4$$

    - ∗ Compute y-coordinate of grid cell containing q:

$$\left\lfloor \frac{7.1 - 6.5}{0.25} \right\rfloor = 2$$

    - ∗ Compute Peano-key:
      - · Binary(0) = 100.
      - · Binary(6) = 010.
      - · Interleave: 100100 (Decimal: 36)

      $\Rightarrow PK_Z(q) = 36$.
  - – Use $PK_Z(q) = 36$ to search in sorted file (binary search).



PK−value	Coordinates		
1	**1.1, 6.9**		p2
3	**1.4, 6.6**		p9
5	**1.1, 7.4**		p4
9	**1.7, 6.8**		p10

14	**2.1, 7.1**		p3
19	**1.4, 7.9**		p1
24	**1.8, 7.6**		p7
24	**1.7, 7.7**		p11

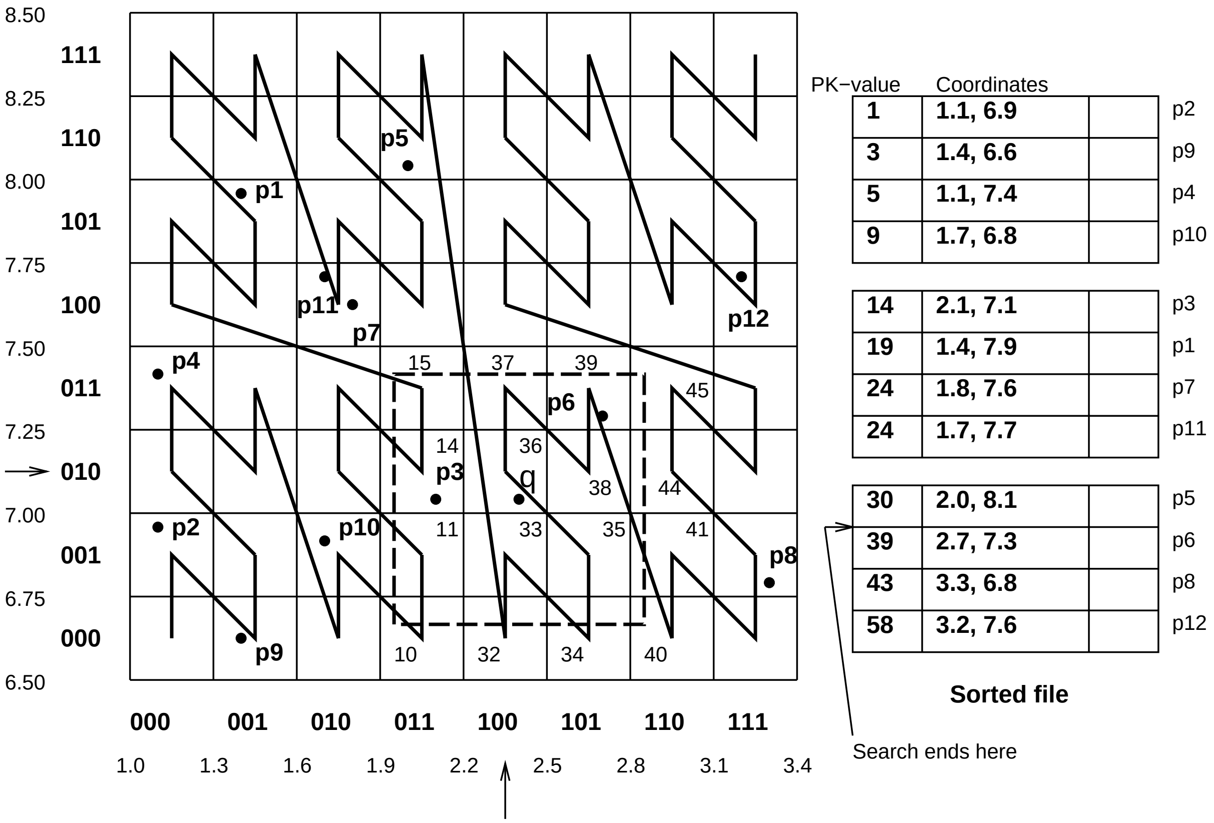
30	**2.0, 8.1**		p5
39	**2.7, 7.3**		p6
43	**3.3, 6.8**		p8
58	**3.2, 7.6**		p12

Sorted file

- Nearest Peano-keys in file:
  - ∗ $PK_Z(p_5) = 30$ in 2nd block.
  - ∗ $PK_Z(p_6) = 39$ in 2nd block.
- Compute distances to these points:
  - ∗ $dist(q, p_5) = \left((2.4 - 2.0)^2 + (7.1 - 8.1)^2\right)^{1/2} = 1.1.$
  - ∗ $dist(q, p_6) = \left((2.4 - 2.7)^2 + (7.1 - 7.3)^2\right)^{1/2} = 0.36.$

  $\Rightarrow p_6$ is closest with distance $d = 0.36$.
- Draw an imaginary square with q as center and d as half-side.
- Find cells covered by square

  ⇒ [3,0], [4,0], [5,0], [6,0], [3,1], [4,1], [5,1], [6,1], [3,2], [4,2], [5,2], [6,2], [3,3], [4,3], [5,3], [6,3].



PK−value	Coordinates		
1	**1.1, 6.9**		p2
3	**1.4, 6.6**		p9
5	**1.1, 7.4**		p4
9	**1.7, 6.8**		p10
14	**2.1, 7.1**		p3
19	**1.4, 7.9**		p1
24	**1.8, 7.6**		p7
24	**1.7, 7.7**		p11
30	**2.0, 8.1**		p5
39	**2.7, 7.3**		p6
43	**3.3, 6.8**		p8
58	**3.2, 7.6**		p12

Sorted file

- Sort by distance: (Spiral order)

  ⇒ [3,1], [4,1], [5,1], [3,2], [4,2], [5,2], [3,3], [4,3], [5,3], [5,0], [6,2], [3,0], [4,0], [6,0], [6,1], [6,3].

- Compute Peano-keys of these cells:

  $PK_Z(3,1) = 11$, $PK_Z(4,1) = 33$, $PK_Z(5,1) = 35$, $PK_Z(6,1) = 41$
  $PK_Z(3,2) = 14$, $PK_Z(4,2) = 36$, $PK_Z(5,2) = 38$, $PK_Z(3,3) = 15$,
  $PK_Z(4,3) = 37$, $PK_Z(5,3) = 39$, $PK_Z(6,3) = 45$, $PK_Z(6,2) = 44$
  $PK_Z(5,0) = 34$, $PK_Z(3,0) = 10$, $PK_Z(4,0) = 32$, $PK_Z(6,0) = 40$

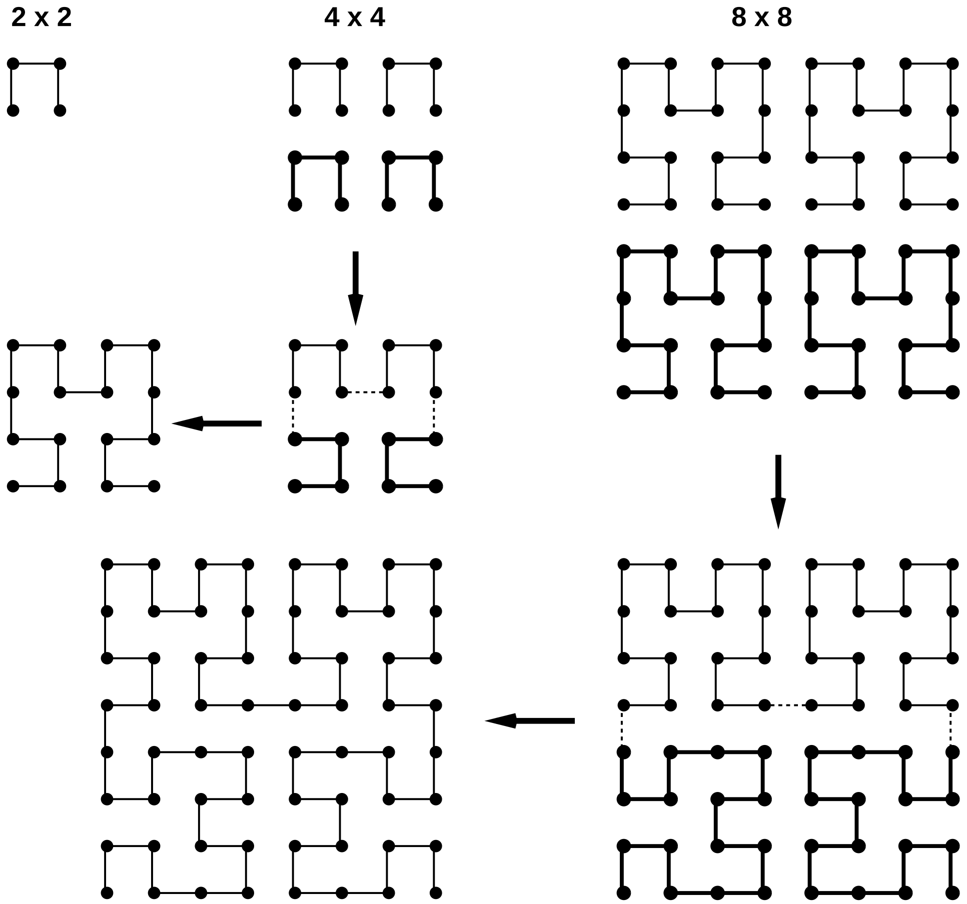
- For each of the above Peano-keys:
  - ∗ Search for the key in the sorted file (using binary search).
  - ∗ Find closest Peano keys to the key (in the file).
  - ∗ Compute distances to those points.

  - Final result: nearest-neighbor of q is $p_3 = (2.1, 7.1)$.

- Note: when searching for multiple cells, it is more efficient to keep track of which blocks have been processed
  $\Rightarrow$ process block by block.

11.3 The Hilbert Curve

- The *Hilbert Curve* is another space-filling fractal.
- It is thought to be slightly "better" than the Z-curve (in terms of neighbor-proximity).
- As with the Z-curve, one starts with a power-of-2:

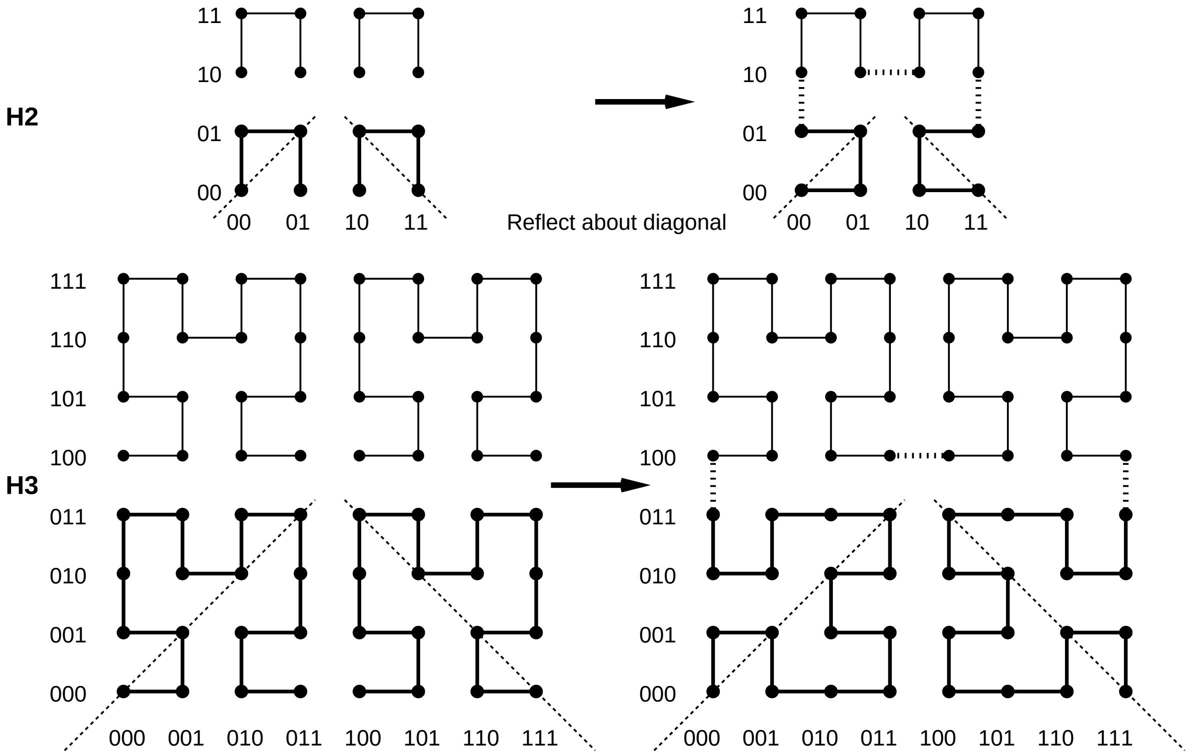


General rule for creating higher-order curves:

- Start with lower order curve.

  - Make 4 copies and place in a square pattern.
  - Rotate bottom-left copy clockwise by 90°.
  - Rotate bottom-right copy anti-clockwise by 90°.
  - Join in the order: bottom-left, top-left, top-right, bottom-right.

- As with the Z-order, we first impose a $2^k \times 2^k$ grid over the data MBR.
- The Hilbert-key of a point is the Hilbert-key of the cell containing the point
  $\Rightarrow$ we need to define the function $PK_H(i, j)$ for a cell address $[i, j]$.
- For the Z-order, we interleaved i and j's bits. The Hilbert-key is more complicated to compute.
- Consider the creation of an $2^k \times 2^k$ curve using a $2^{k-1} \times 2^{k-1}$ curve.

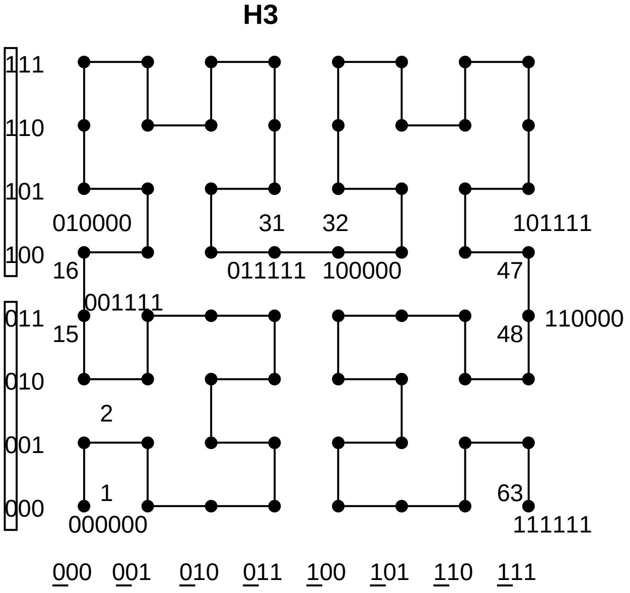


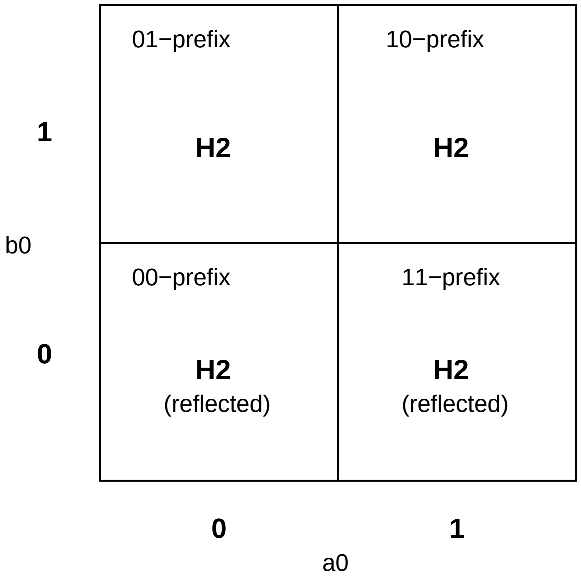
Reflect about diagonal

- Four copies (of a $2^{k-1} \times 2^{k-1}$) curve are made and arranged in a square.
- The bottom two copies are rotated.
- Each rotation can also be considered a *reflection*:
  - * The bottom-left copy is reflected about a 45° line.
  - * The bottom-right copy is reflected about a 135° line.
- The computation of a Hilbert-key $PK_H(i,j)$ reduces to this problem:
  - * Suppose $a = a_{k-1}a_{k-2}\ldots a_0$ is the binary representation of i.
  - * Suppose $b = b_{k-1}b_{k-2}\ldots b_0$ is the binary representation of j.

a2	a1	a0	b2	b1	b0
0	**0**	**1**	**1**	**0**	**1**

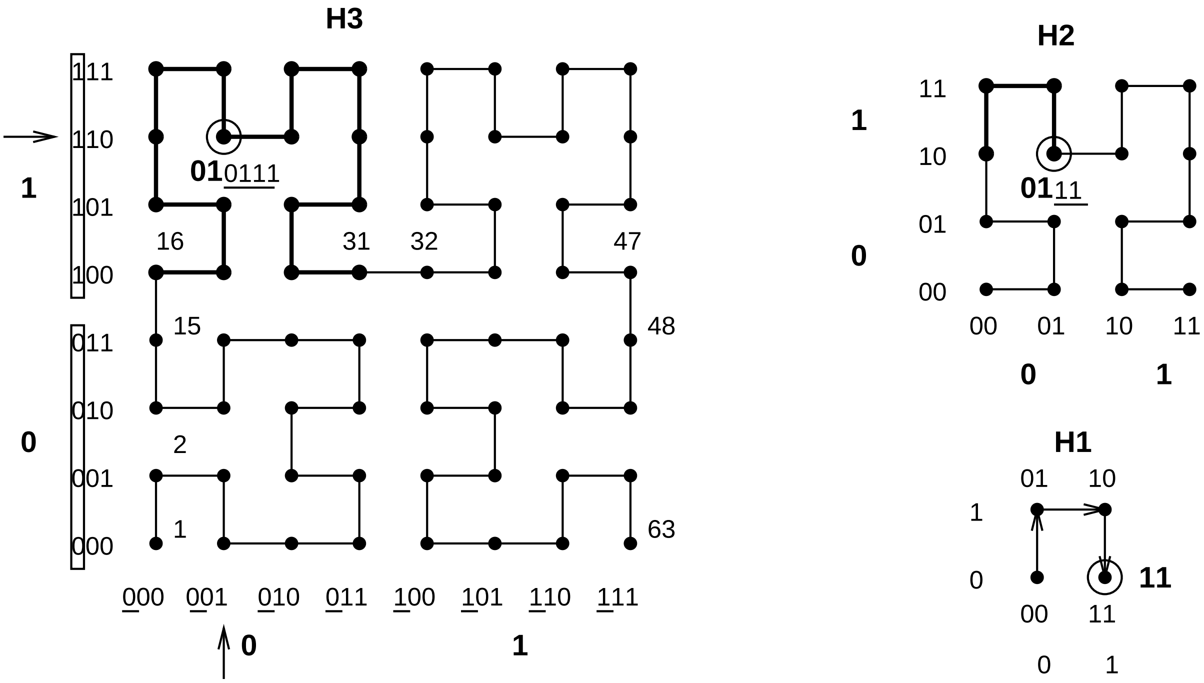
  - * We need to compute $h_{2k-1}h_{2k-2}\ldots h_0$, the Hilbert-key of (i,j) using a and b.
- Notation: let $H_k(a,b)$ denote the Hilbert-key obtained using the lowest k-order bits of bitstrings a and b.
- Observation: the first two bits of H_k depend on the quadrant:





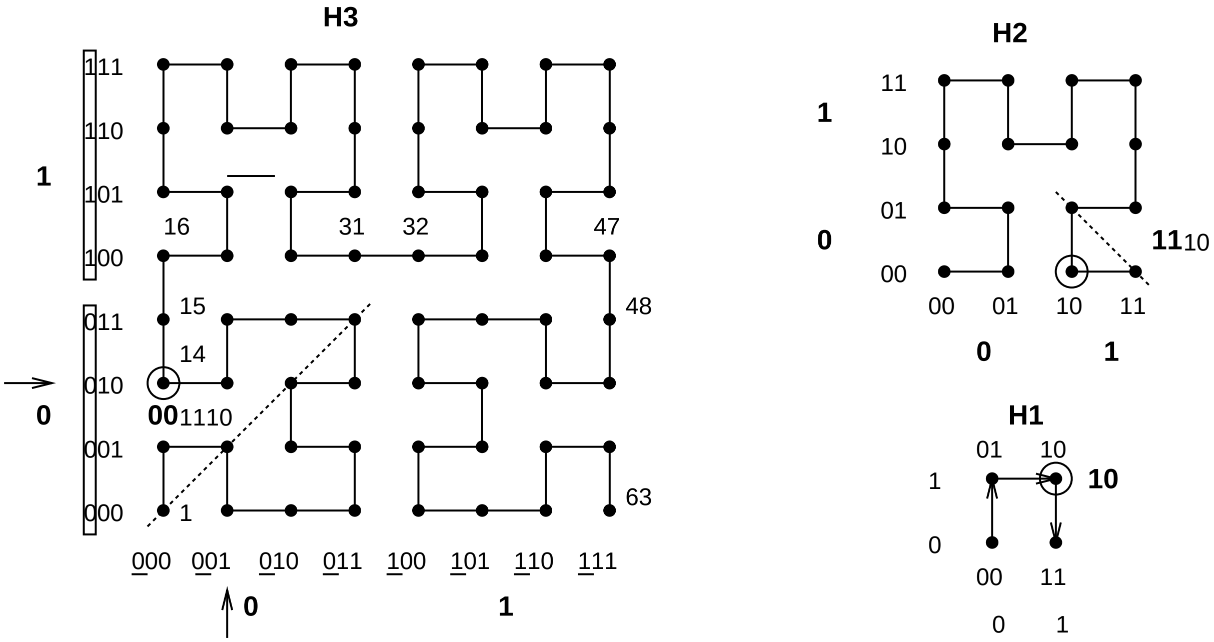
⇒ $h_{2k-1} = a_{k-1}$ and $h_{2k-2} = a_{k-1} \oplus b_{k-1}$.

- Each quadrant contains an H_{k-1}-curve, except that the bottom two quadrants contain *reflected* H_{k-1} curves.
- Consider the coordinates 001 and 110
  ⇒ it turns out that $H_3(001, 110) = 010111$.
- We know that 01 are the first two bits of H_3.
- What about the others?
  ⇒ use the smaller H-curves in each quadrant (recursively).



H3 (001, 110)
= 01 H2(01, 10)
= 01 01 H1(1,0)
= 01 01 11

- The above example was easy because no reflections were involved.
- Example with reflection:



H3 (000, 010)
= 00 H2 (45−deg reflection of 00, 10)
= 00 H2 (10, 00)
= 00 11 H1 (135−deg reflection of 0,1)
= 00 11 H1 (1, 1)
= 00 11 10

- Implementing reflection:
  - About the 45°-line: switch x and y coordinates.
  - About the 135°-line: Subtract x and y from 2^k and switch.
- Let $a^{(k)}$ denote the lowest k bits of a bitstring a.
- Recursive algorithm for computing $H_k(a, b)$:

  1. $h_{2k-1} = a_{k-1}$.
  2. $h_{2k-2} = a_{k-1} \oplus b_{k-1}$.
  3. **if** $b_{k-1} = 1$ **return** $h_{2k-1}h_{2k-2}H_{k-1}(a^{(k-1)}, b^{(k-1)})$
  4. **else if** $a_{k-1} = b_{k-1} = 0$ **return** $h_{2k-1}h_{2k-2}H_{k-1}(b^{(k-1)}, a^{(k-1)})$
  5. **else return** $h_{2k-1}h_{2k-2}H_{k-1}(2^{k-1} - 1 - b^{(k-1)}, 2^{k-1} - 1 - a^{(k-1)})$

11.4 R-trees: Introduction

- An R-tree is a spatial index that supports insertion, search and deletion of spatial objects.
- R-trees are similar to B+-trees:
  - An R-tree is an index; the actual data is in a heapfile.
  - Each R-tree node has a minimum and maximum occupancy.
  - Internal and leaf nodes are different.
  - Internal nodes are used for navigation.
  - Leaf nodes contain datapointers.

  However, R-trees are also different from B+-trees:

  - In a B or B+-tree, a search traverses a unique path from the root.
  - In an R-tree, a search may require traversing multiple paths to the leaf level.
- Minimum Bounding Rectangles:
  - A Minimum Bounding Rectangle (MBR) of an object is the the smallest rectangle that completely contains the object.
  - Conventionally, MBR's are constrained to have sides parallel to the axes.
  - An MBR of a collection of objects is the smallest rectangle enclosing all the objects.
  - An MBR of a collection of MBR's $M_1, M_2, \ldots, M_k$ is the smallest rectangle enclosing $M_1, M_2, \ldots, M_k$.
- While the data objects are usually polygons, they can really be anything, as long as an MBR can be computed for each object (e.g., a circle).
- Point data have degenerate MBR's.

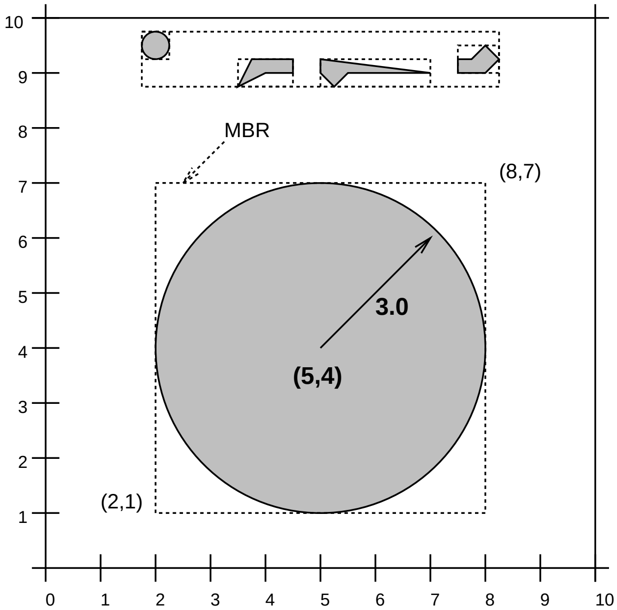
- An R-tree works only with MBR's: you insert MBR's, you delete MBR's and you search for MBR's.

  Thus, a user

  - inserts an object into a heapfile (thereby obtaining a datapointer),
  - computes the object's MBR, and
  - inserts the *MBR and datapointer* into the R-tree.

  For example:

  - Suppose the object is a circle with center (5.0,4.0) and radius 3.0.
  - Suppose the object is inserted into block# 873 as the 6th tuple.
    ⇒ the datapointer is (873,6).
  - The MBR is the rectangle with
    - ∗ Lower left corner: the point (2,1).
    - ∗ Upper right corner: the point (8,7)

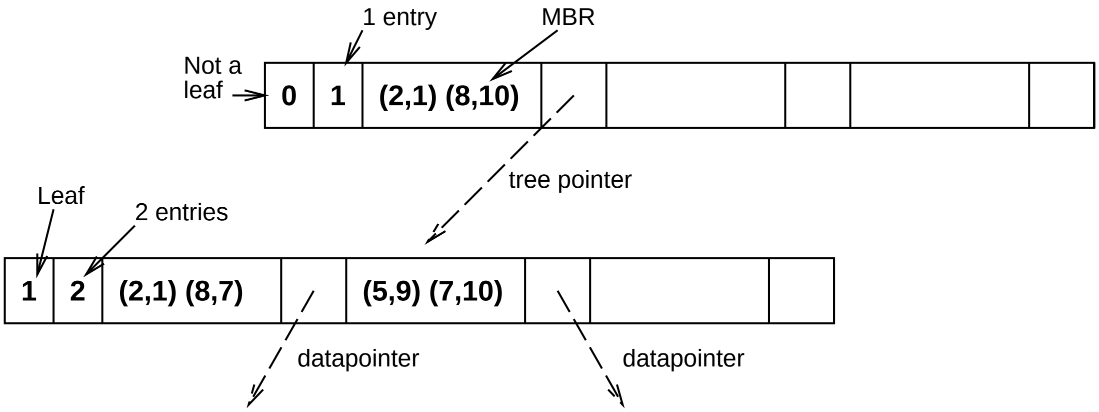


  - The item inserted into the R-tree is the MBR and the datapointer: `[ <(2,1), (8,7)>, (873,6) ]` (6 numbers).
    ⇒ an R-tree leaf entry is of the form *[MBR, datapointer]*.

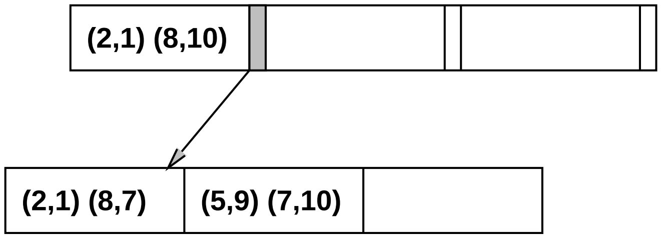
- Description of R-tree nodes:
  - Leaf nodes:
    - A leaf node contains *[MBR, datapointer]* entries.
    - Each leaf node contains a maximum M entries
      $\Rightarrow M$ is determined by the blocksize.
    - Each leaf node contains at least $\frac{M}{2}$ entries.
  - Internal nodes (except the root):
    - An internal node contains *[MBR, treepointer]* entries.
    - The treepointer points to a (child) node at the next level.
    - The MBR value is the MBR of all the MBR's in the child node.
    - Each internal node contains between $\frac{M}{2}$ and M entries.
  - The root node:
    - The root may contain fewer than $\frac{M}{2}$ entries.
    - The root node contains at least 1 entry.

All nodes also contain a boolean value (indicating whether it's a leaf) and an integer (the number of current entries).

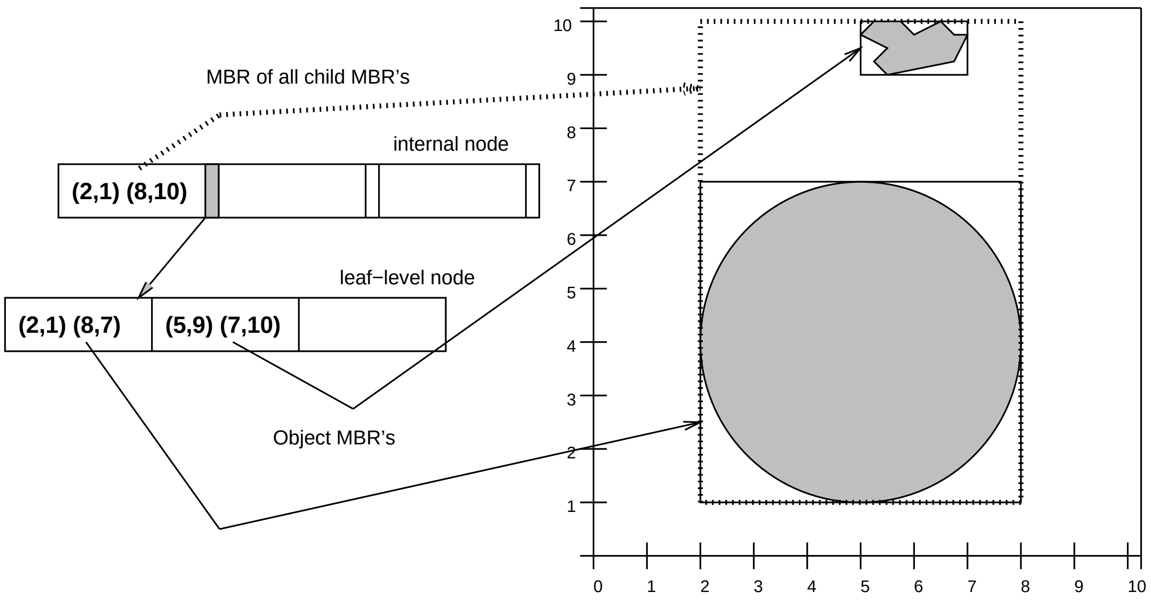
Structure of R-tree nodes (example):



Conceptual view:



How this relates to the object data:



- Example of computing M:
  - Block size 256 bytes.
  - 2 bytes used for block-specific information.
  - 4 bytes per MBR coordinate
    ⇒ 16 bytes per MBR
  - 6 bytes per datapointer or tree pointer
    ⇒ 16+6 = 22 bytes per entry.
  - Max entries $M = \frac{254}{22} = 11$ entries.
- NOTE: it is the user's responsibility to:
  - insert a data object tuple into a heapfile;
  - compute the MBR;
  - present the R-tree with the datapointer and MBR.

  Why not have the R-tree do the heapfile insertion?
  - The R-tree would have to know how to compute the MBR.
  - Object tuple sizes depend on type of object
    ⇒ e.g., circle is fixed-size, polygons are variable-size.
  - MBR computations depend on the type of object
    ⇒ e.g., radically different for polygons and circles.
  - We want the R-tree to be *independent* of the object type
    ⇒ R-tree cannot compute MBR or know tuple size.

11.5 Insertion in an R-tree

- Key ideas:

  R-tree insertion is similar to B+-tree insertion:

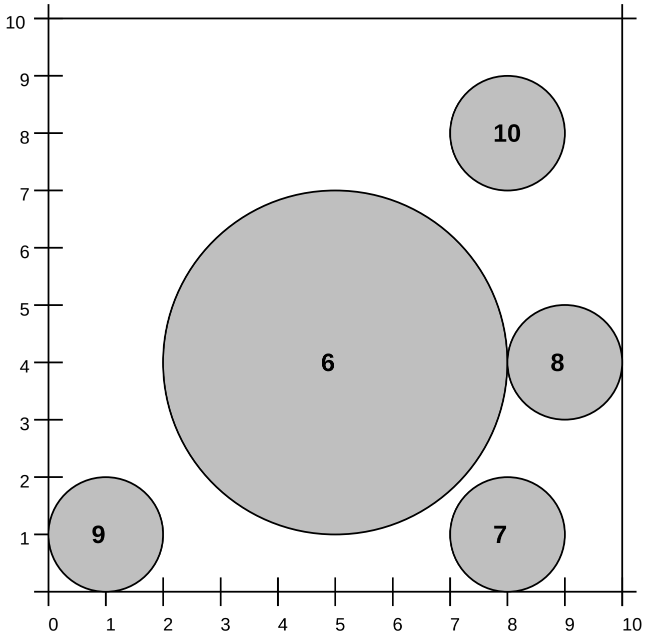
  - Find the appropriate leaf node by navigating down the tree.
  - Insert into leaf node if space is available.
  - If the leaf node is full, split the leaf node and make an insertion at the parent level.
  - If the parent is full, split the parent ... and so on, recursively.

  What's different in an R-tree:

  - The input is an MBR (and a datapointer).
  - In finding an appropriate leaf node:
    - Start at the root.
    - Pick the subtree whose MBR needs the least expansion to accomodate the new rectangle.
    - Go to this child node and repeat the search criterion until the leaf-level is reached.
  - If space is available in the leaf-level, insert the node.
  - Next, go back up the path taken from the root, and adjust (expand) successive parent MBR's to accomodate the new insertion.
  - If a split is required:
    - Place entries $1, \ldots, \lfloor \frac{M}{2} \rfloor$ in the left child (current node).
    - Place entries $\lfloor \frac{M}{2} \rfloor + 1, \ldots, M$ in newly-created right node.
    - Place new rectangle in the right child.
    - Compute new MBR's for both left and right child.

- ∗ At the parent's level, adjust the left child's MBR and insert the new right child's MBR
  ⇒ this may cause a split
  ⇒ split parent ... etc.

- Consider the following simple example with circle objects:

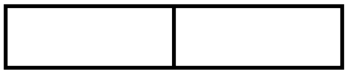


Circle 6:	center at (5,4), radius 3
Circle 7:	center at (8,1), radius 2
Circle 8:	center at (9,4), radius 2
Circle 9:	center at (1,1), radius 2
Circle 10:	center at (8,8), radius 2

We are going to insert the circle objects in the order 6,7,8,9,10.

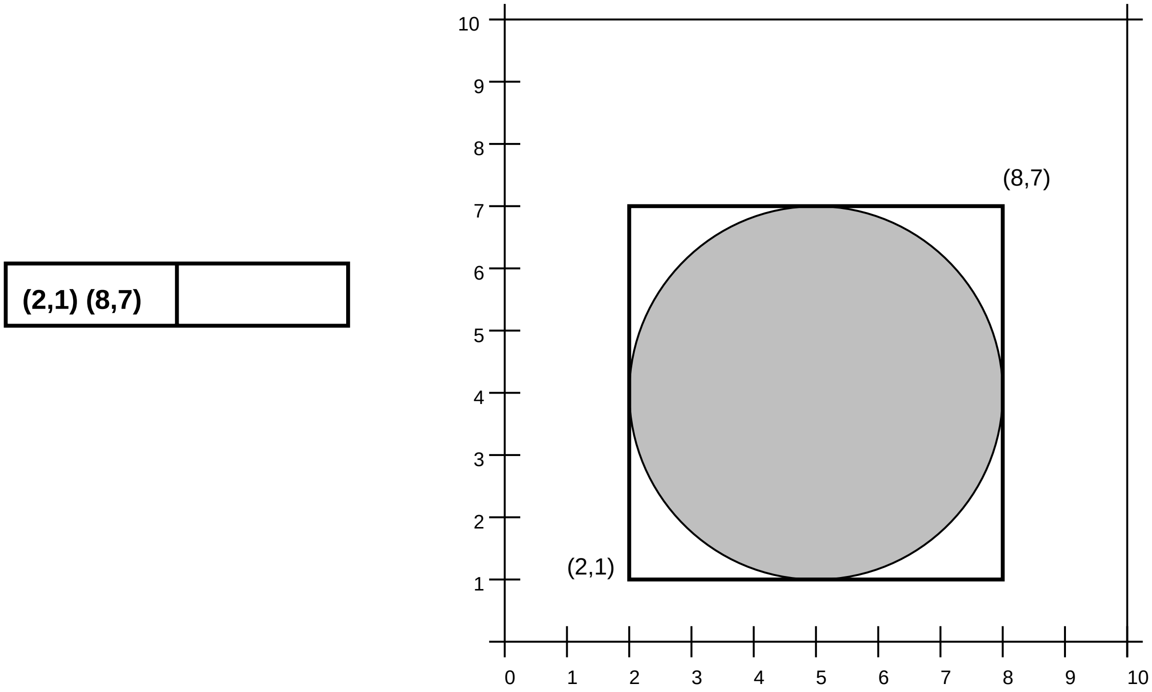
In the example, $M = 2$
⇒ at most 2 entries per node
⇒ at least 1 entry per node.

- Initially: create empty root.



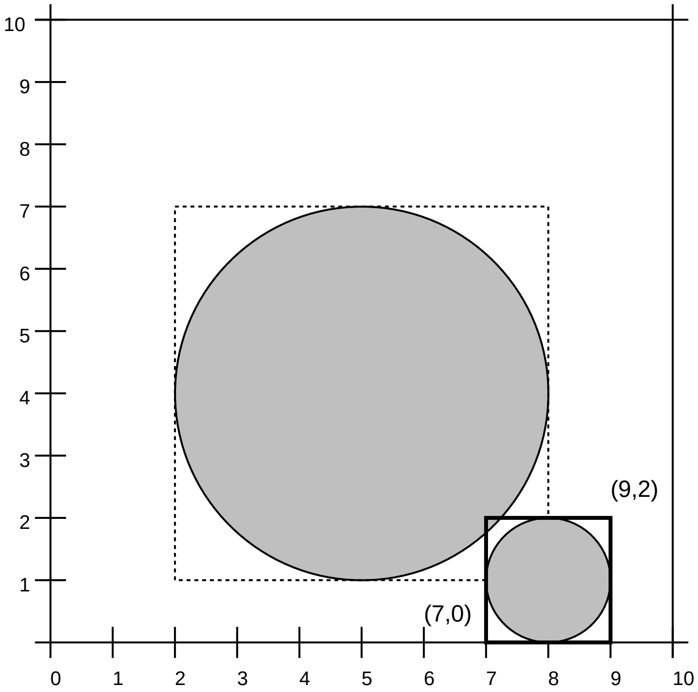
- Insert *circle 6*:
  - User computes MBR of object: `[(2,1) (8,7)]`.
  - Search for appropriate leaf:
    ⇒ only root node.

- Space available ⇒ insert MBR.

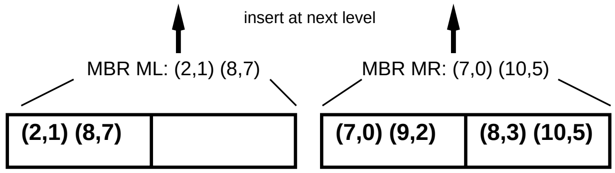


- Insert *circle 7*:
  - User computes MBR of object: `[(7,0) (9,2)]`.
  - Search for appropriate leaf:
    ⇒ only root node.
  - Space available ⇒ insert MBR.

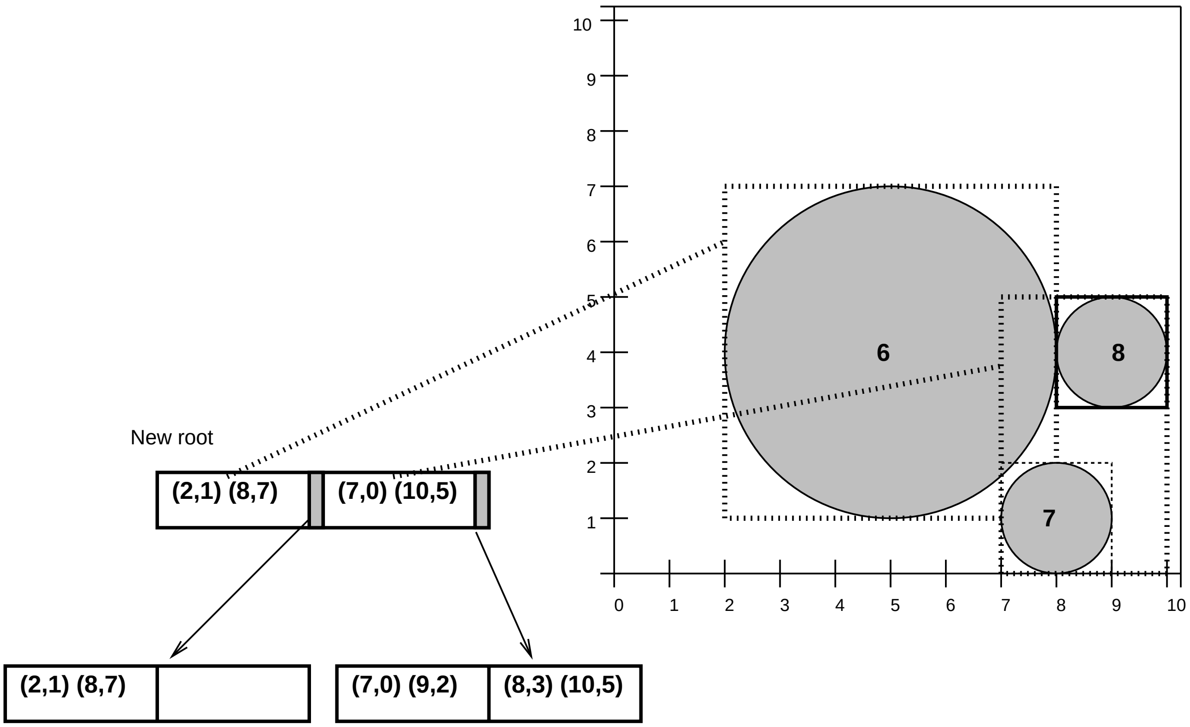
(2,1) (8,7)	**(7,0) (9,2)**



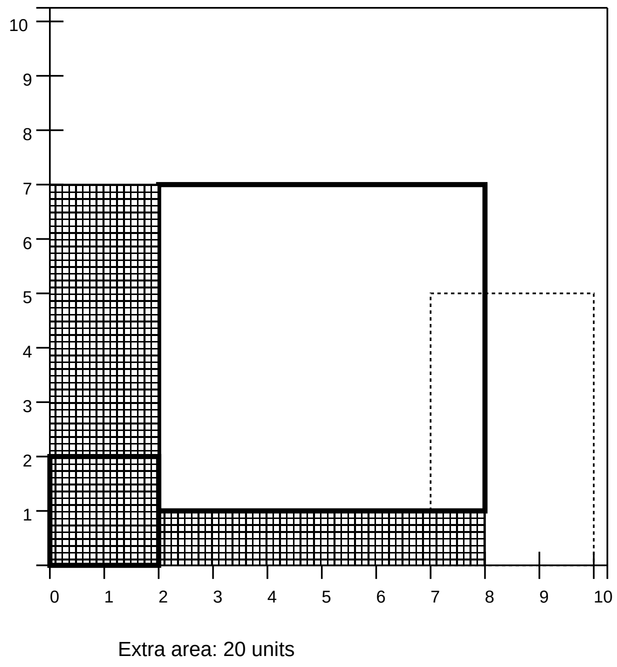
- Insert *circle 8*:
  - User computes MBR of object: `[(8,3) (10,5)]`.
  - Search for appropriate leaf:
    ⇒ root is already a leaf.
  - Node is full ⇒ split node:
    - ∗ Place half the entries in left node (current node):
      ⇒ `[(2,1) (8,7)]`.
    - ∗ Compute MBR of left node (call it M_L):
      ⇒ `[(2,1) (8,7)]`.
    - ∗ Create new block (right node).
    - ∗ Place remaining entries in right node:
      ⇒ `[(7,0) (9,2)]`.
    - ∗ Place new MBR in right node.
    - ∗ Compute MBR of right node (call it M_R):
      ⇒ `[(7,0) (10,5)]`.



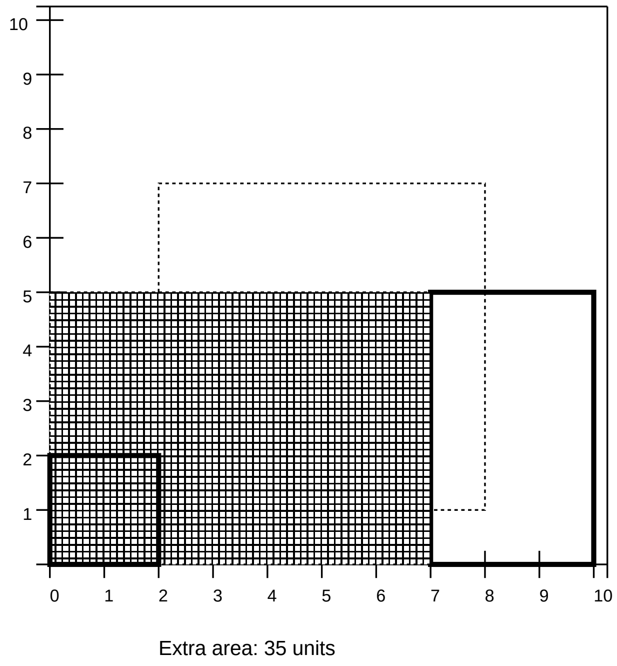
  * Insert M_L and M_R in parent:
    ⇒ no parent
    ⇒ create new root:



- Insert *circle 9*:
  - User computes MBR of object: `[(0,0) (2,2)]`.
  - Search for appropriate leaf:
    * For each MBR in root, compute expansion needed to accomodate new rectangle.



⇒ 20 units of expansion needed for MBR `[(2,1) (8,7)]`.



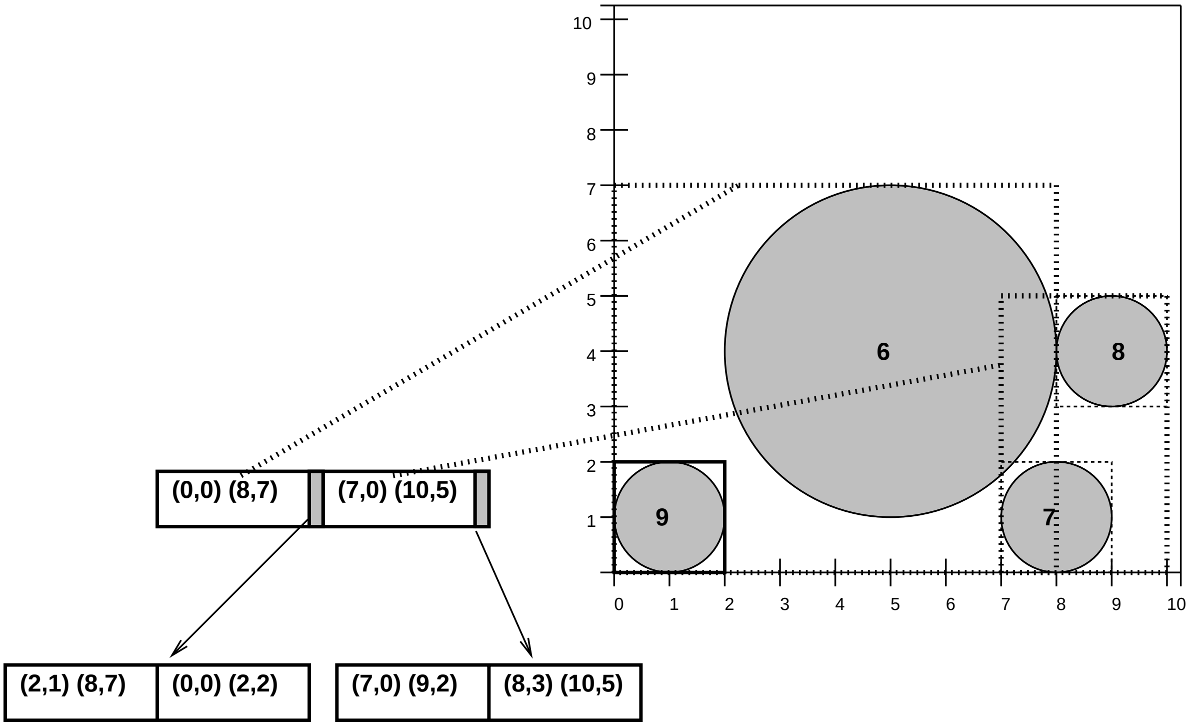
⇒ 35 units of expansion needed for MBR `[(7,0) (10,5)]`

⇒ select left subtree to search.

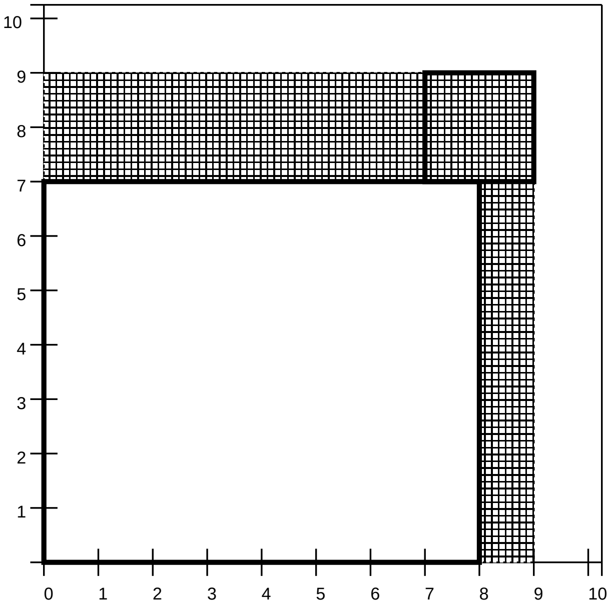
∗ Left child is at leaf level ⇒ search complete.

– Space available ⇒ insert new rectangle.

  - Work backwards up the path to the root and expand MBR's along the way to accomodate new rectangle
    ⇒ expand `[(2,1) (8,7)]` to `[(0,0) (8,7)]`.

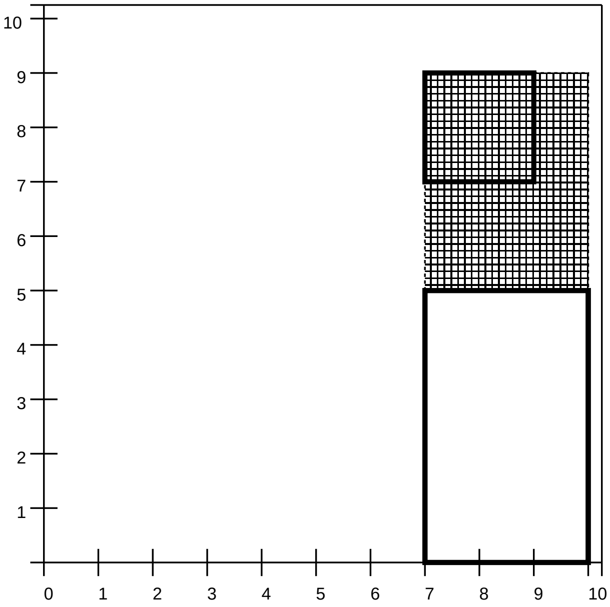


- Insert *circle 10*:
  - User computes MBR of object: `(7,7) (9,9)]`.
  - Search for appropriate leaf:
    - ∗ For each MBR in root, compute expansion needed to accomodate new rectangle.



Extra area: 25 units

⇒ 25 units of expansion needed for MBR `[(2,1) (8,7)]`.



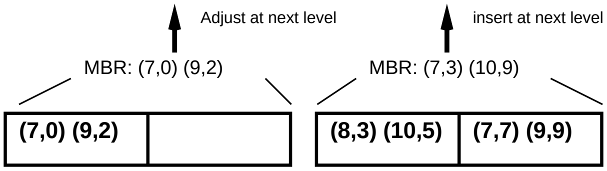
Extra area: 12 units

⇒ 12 units of expansion needed for MBR `[(7,0) (10,5)]`
⇒ select right subtree to search.

∗ Right child is at leaf level ⇒ search complete.

– No space available ⇒ split node:

- Place half the entries in left node (current node):
  ⇒ `[(7,0) (9,2)]`.
- Compute MBR of left node (M_L):
  ⇒ `[(7,0) (9,2)]`.
- Create new block (right node).
- Place remaining entries in right node:
  ⇒ `[(8,3) (10,5)]`.
- Place new MBR in right node.
- Compute MBR of right node (M_R):
  ⇒ `[(7,3) (10,9)]`.



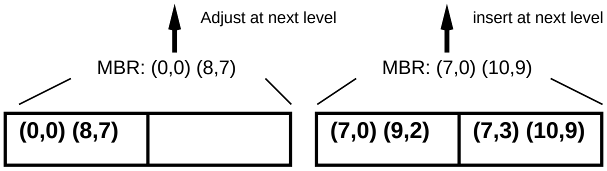
- Remember the MBR of the split node that was in the parent?
  ⇒ `[(7,0) (10,5)]`.
- This needs to be adjusted to accomodate the new left child
  ⇒ new left child MBR is `[(7,0) (9,2)]` (M_L).
  ⇒ old MBR entry is *over-written* with this value.

– Next, insert the new MBR (M_R) into the parent.

– Parent is full ⇒ split node:

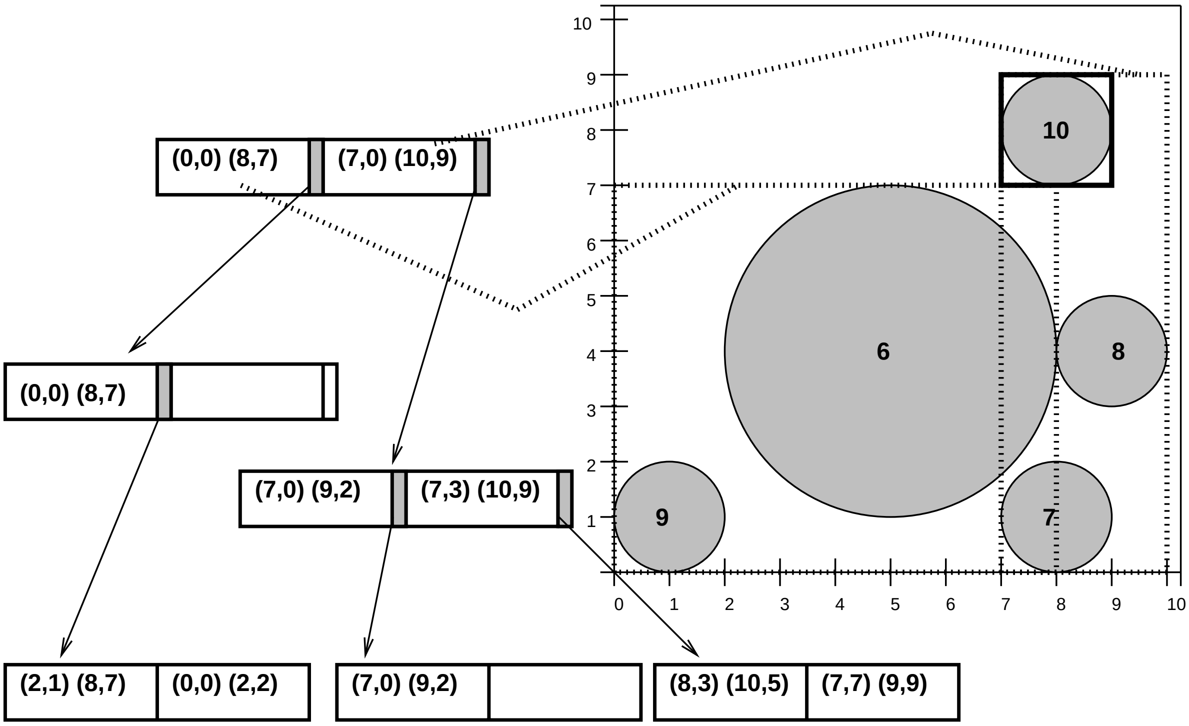
- Place half the entries in left node (current node):
  ⇒ `[(0,0) (8,7)]`.
- Compute MBR of left node (call this M_L):
  ⇒ `[(0,0) (8,7)]`.
- Create new block (right node of split).
- Place new MBR in right node.
- Compute MBR of right node (call this M_R):
  ⇒ `[(7,0) (10,9)]`.

- * These two MBR's are passed to the next level.



– There is no next level

$\Rightarrow$ create new root (with entries M_L and M_R):



- Pseudocode for insertion:

  The following convention will be used:

  - Calls to DISK functions return either a disk block number (an integer type) or a memory address (an address type).
  - DISK-NEWBLOCK returns a block number.
  - DISK-WRITEBLOCK and DISK-READBLOCK take a block number as input.
  - DISK-READBLOCK returns a memory address. Thus,

    b := DISK-READBLOCK (blknum);

    reads block number blknum into memory and returns the address, which can be used as

    b →leaf := false;

  - The MBR's will be stored in b →R[1], b →R[2], ...
  - b →child[i] refers to the node at the next level pointed to by the i-th entry in b.

Algorithm: RTREE-CREATE

Input: none.
Output: root block written to disk.

```
Compute max_entries in an R-tree node;
root := DISK-NEWBLOCK();
$b$ := DISK-READBLOCK (root);
$b$ →leaf := true;
$b$ →num_entries := 0;
DISK-WRITEBLOCK (root);
**return**;
```

Algorithm: Rtree-Insert (R, D)

Input: MBR R and datapointer D.
Output: The pair <R,D> is inserted into the tree.

```
Initialize stack;
b := Disk-Readblock (root);
blknum := root;
// First search for correct leaf
while b →leaf ≠ true
   Find k such that b →R[k] needs the least expansion
      among b →R[1],...,b →R[b →num_entries] to accomodate R.
   Stack-Push (blknum, k);
   blknum := b →child[k];
   b := Disk-Readblock (blknum);
endwhile
// Now b is the desired leaf
if b →num_entries < max_entries
   // Space available ⇒ insert directly
   b →num_entries := b →num_entries + 1;
   b →R[b →num_entries] := R;
   R2 := Block-MBR (b);
   Disk-Writeblock (blknum);
   Rtree-Recursive-Insert (R2, 0, false);
   return;
endif
// Otherwise, a split is needed
(LeftR, RightR) := Split-Rtree-Node (b, R);
// Assume Split-Rtree-Node writes the two blocks
if b is not the root
   Rtree-Recursive-Insert (LeftR, RightR, true);
else
   Create-New-Rtree-Root (LeftR, RightR);
return;
```

Algorithm: RTREE-RECURSIVE-INSERT (LeftR, RightR, new)

Input: MBR's LeftR, RightR, and boolean New.
Output: The MBR's are inserted into node obtained from stack.

```
if STACK-EMPTY() return
(blknum, $k$) := STACK-POP ();
$b$ := DISK-READBLOCK (blknum);
// Re-adjust child rectangle to LeftR
$b$ →R[$k$] := LeftR;
if not new
    // No new MBR being added ⇒ only need to re-adjust
    DISK-WRITEBLOCK (blknum);
    R := BLOCK-MBR ($b$);
    RTREE-RECURSIVE-INSERT (R, 0, false);
    return;
endif
// Otherwise, we need to add RightR
if $b$ →num_entries < max_entries
    // Space available ⇒ insert directly
    $b$ →num_entries := $b$ →num_entries + 1;
    $b$ →R[$b$ →num_entries] := RightR;
    R := BLOCK-MBR ($b$);
    DISK-WRITEBLOCK (blknum);
    RTREE-RECURSIVE-INSERT (R, 0, false);
endif
// Otherwise, a split is needed.
(LeftR2, RightR2) := SPLIT-RTREE-NODE ($b$, RightR);
if $b$ is not the root
    RTREE-RECURSIVE-INSERT (LeftR2, RightR2, true);
else
    CREATE-NEW-RTREE-ROOT (LeftR2, RightR2);
return;
```

Algorithm: Create-New-Rtree-Root (LeftR, RightR)

Input: MBR's LeftR and RightR.
Output: A new root written to disk.

```
root := Disk-Newblock ();
b := Disk-Readblock (root);
b →num_entries := 2;
b →R[1] := LeftR;
b →R[2] := RightR;
Disk-Writeblock (root);
return;
```

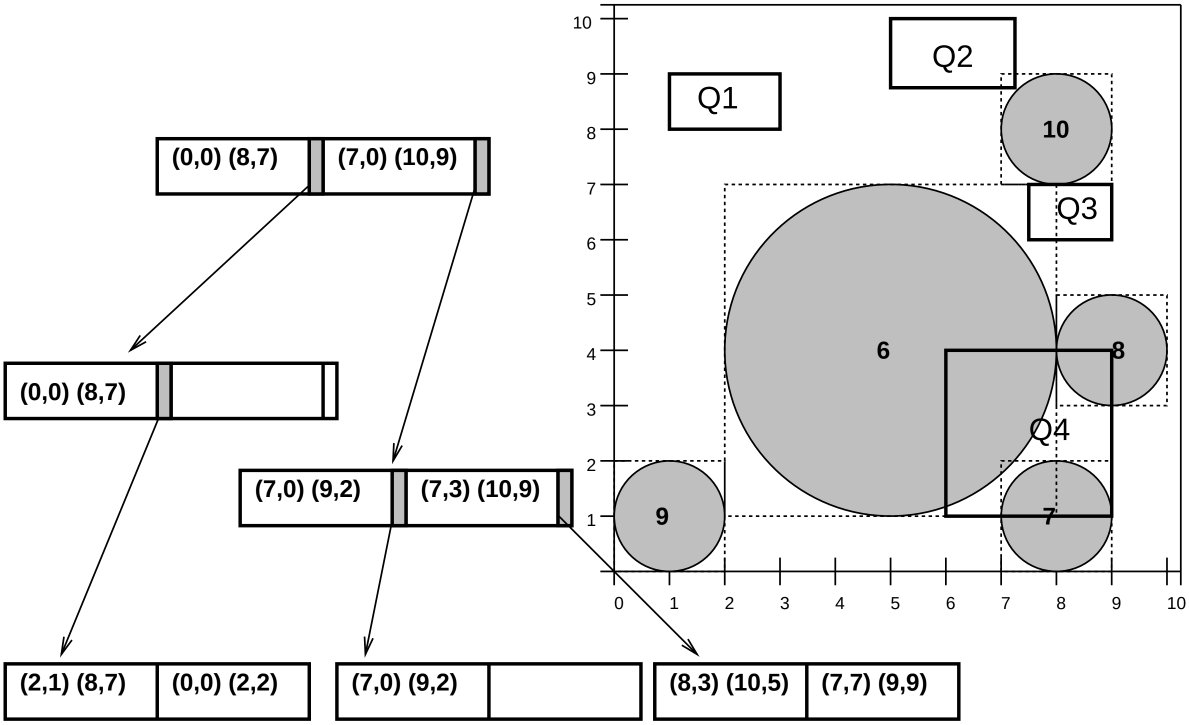
Algorithm: Split-Rtree-Node (*b*, R, LeftR, RightR)

Input: block pointer *b*, MBR R.
Output: Split blocks written to disk, MBR's for each block returned.

```
median := (max_entries + 1) / 2;
b →num_entries := median;
LeftR := Block-MBR (b);
blknum := Disk-Newblock ();
b2 := Disk-Readblock (blknum);
Place entries median+1, ..., max_entries in b2;
Place new MBR R in b2;
RightR := Block-MBR (b2);
Write blocks b and b2 to disk;
return LeftR, RightR;
```

11.6 Searching in an R-tree

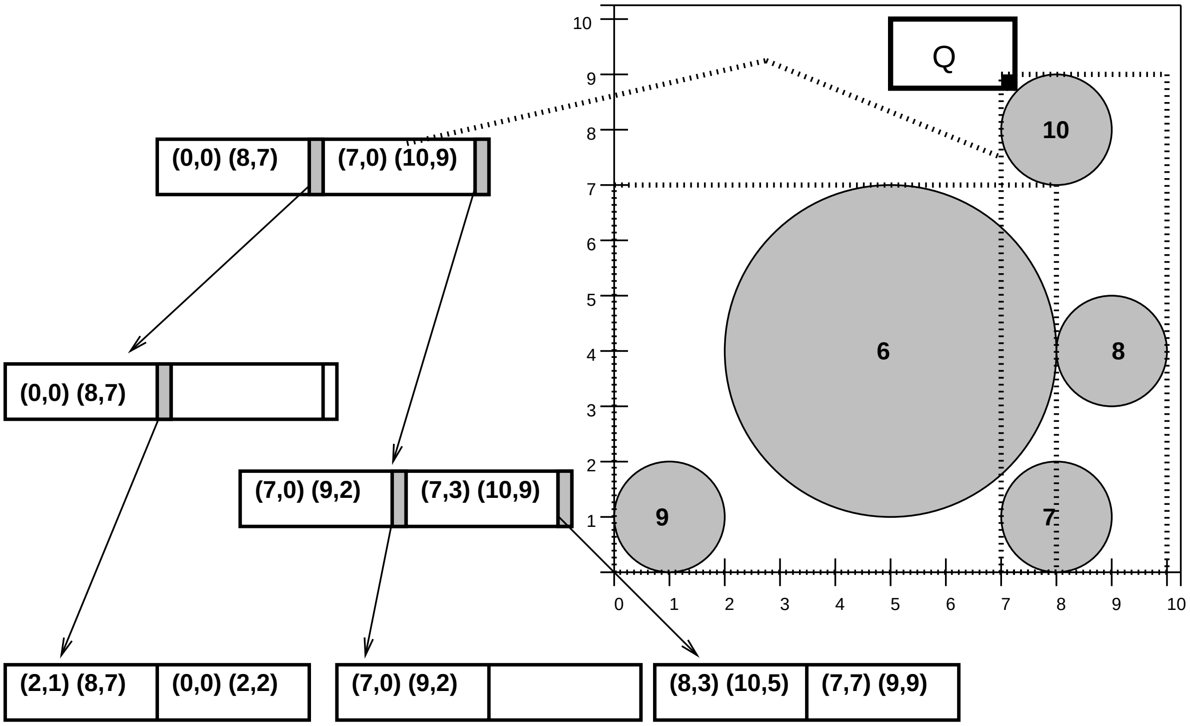
- Key ideas:
  - Searching usually consists of specifying a *query rectangle*.
  - The R-tree is required to return all data MBR's that intersect with the given query rectangle.
  - For example, consider the queries Q1, Q2, Q3, Q4:



    - ∗ Q1 returns no MBR (and therefore no object).
    - ∗ Q2 returns MBR for Circle 10, although a subsequent retrieval of the object reveals no intersection.
    - ∗ Q3 returns the MBR of Circle 6 and the MBR of Circle 10 (if touching counts as intersection).
    - ∗ Q4 returns the MBR's of circles 6,7 and 8.

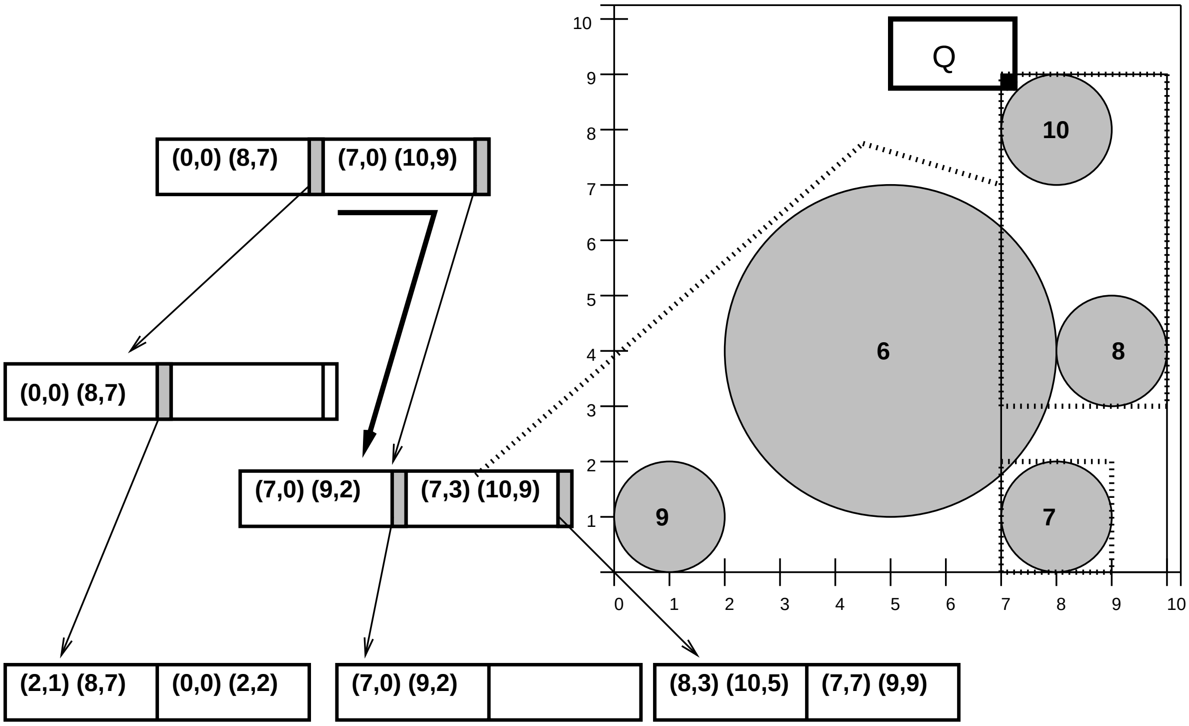
- Consider a query rectangle Q.
- Start the search by computing which of the MBR's in the root intersect with Q
  ⇒ Explore all those corresponding subtrees.
- In general, for each internal node encountered, find all the MBR's which intersect with Q
  ⇒ Explore all those corresponding subtrees.
- For a leaf node, return all MBR's (and datapointers) that intersect with Q.

- Example: Q = `[(5, 8.8) (7.2, 10)]`.
  - Check MBR's in root
    ⇒ Q intersects only `[(7,0) (10,9)]`:



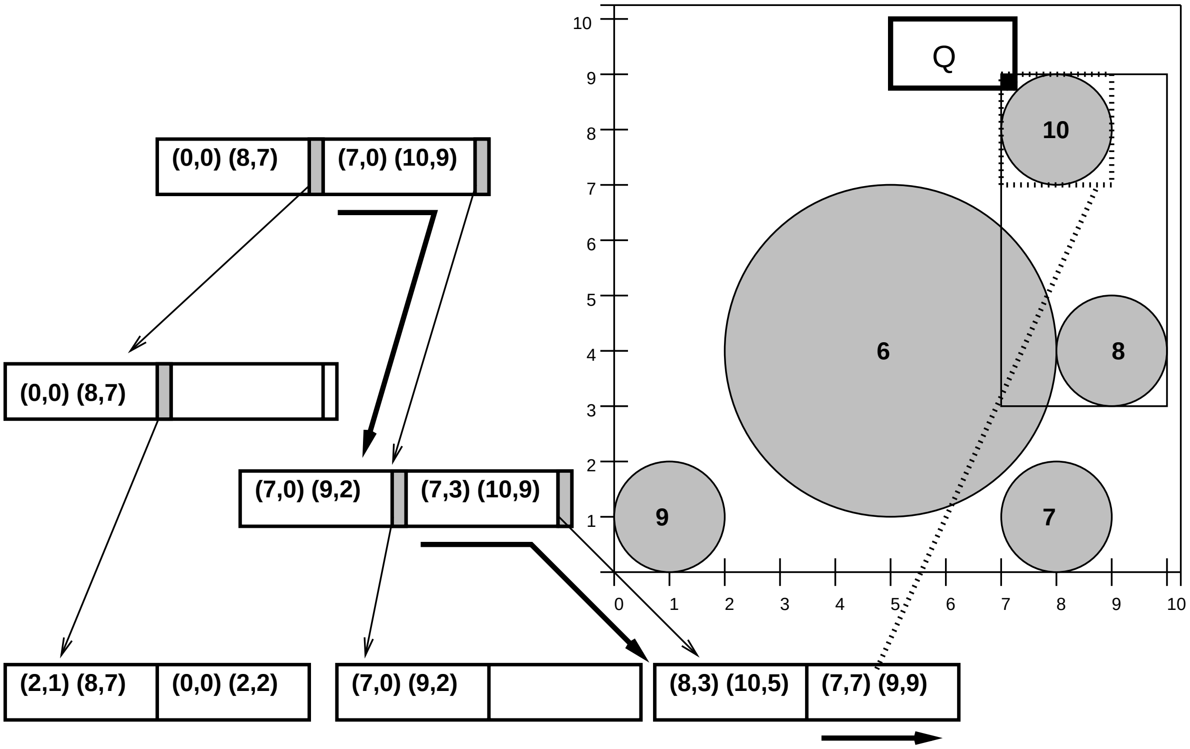
  - Explore the subtree of `[(7,0) (10,9)]`.

– Check MBR's in child node (next level)
⇒ Q intersects only `[(7,3) (10,9)]`

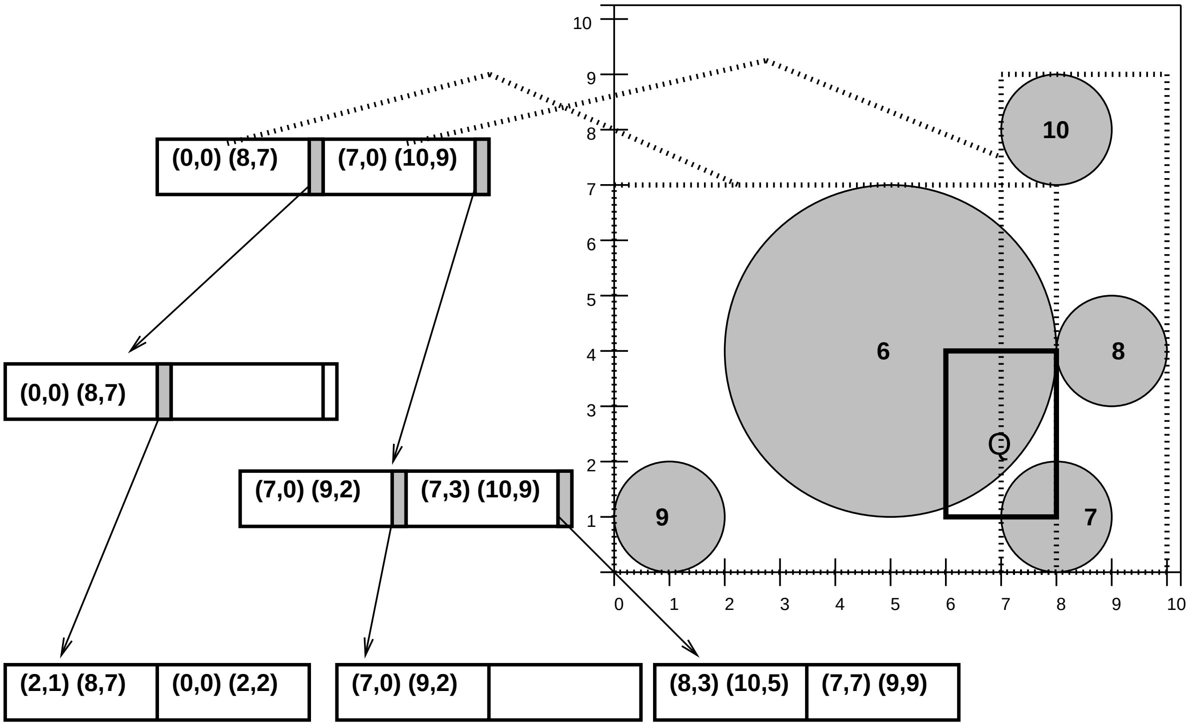


– Explore the subtree of `[(7,3) (10,9)]`.

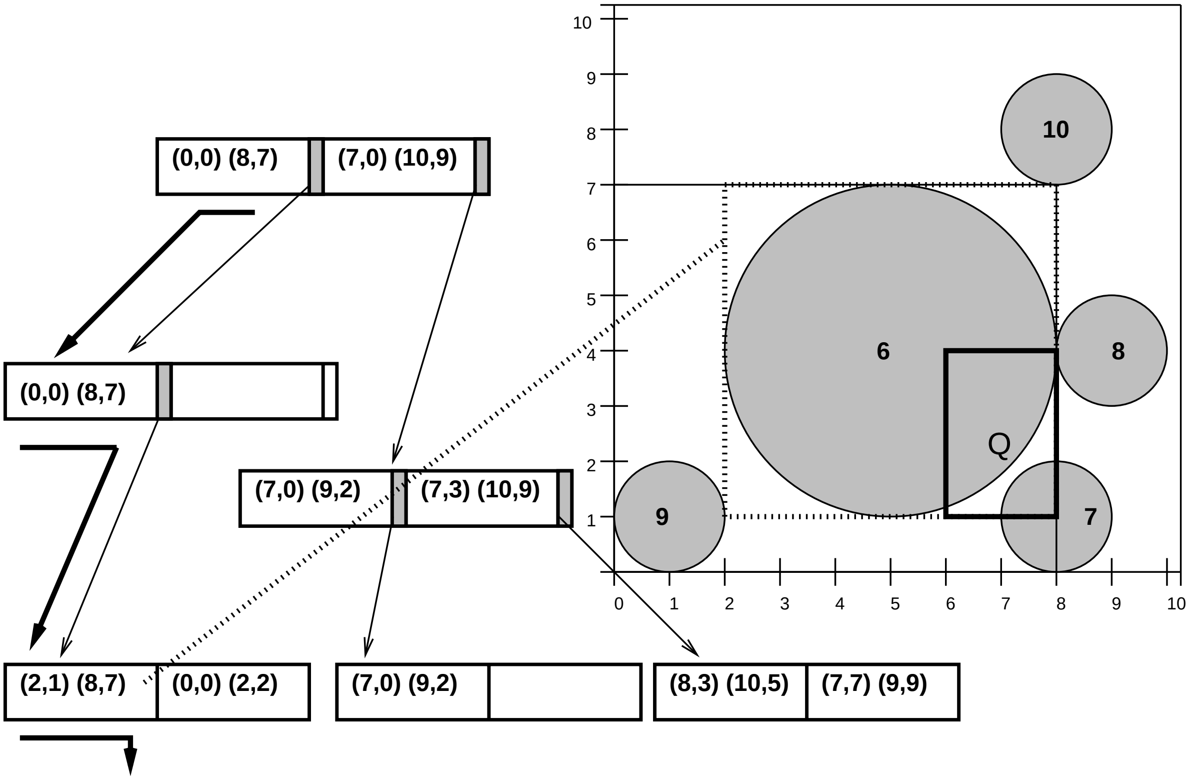
– Check MBR's at next (leaf) level
⇒ Q intersects `[(7,7) (9,9)]`
⇒ return the corresponding datapointer.


(0,0) (8,7)
(7,0) (10,9)
(0,0) (8,7)
(7,0) (9,2)
(7,3) (10,9)
(2,1) (8,7)
(0,0) (2,2)
(7,0) (9,2)
(8,3) (10,5)
(7,7) (9,9)
Q
10
6
8
9
7
0
1
2
3
4
5
6
7
8
9
10

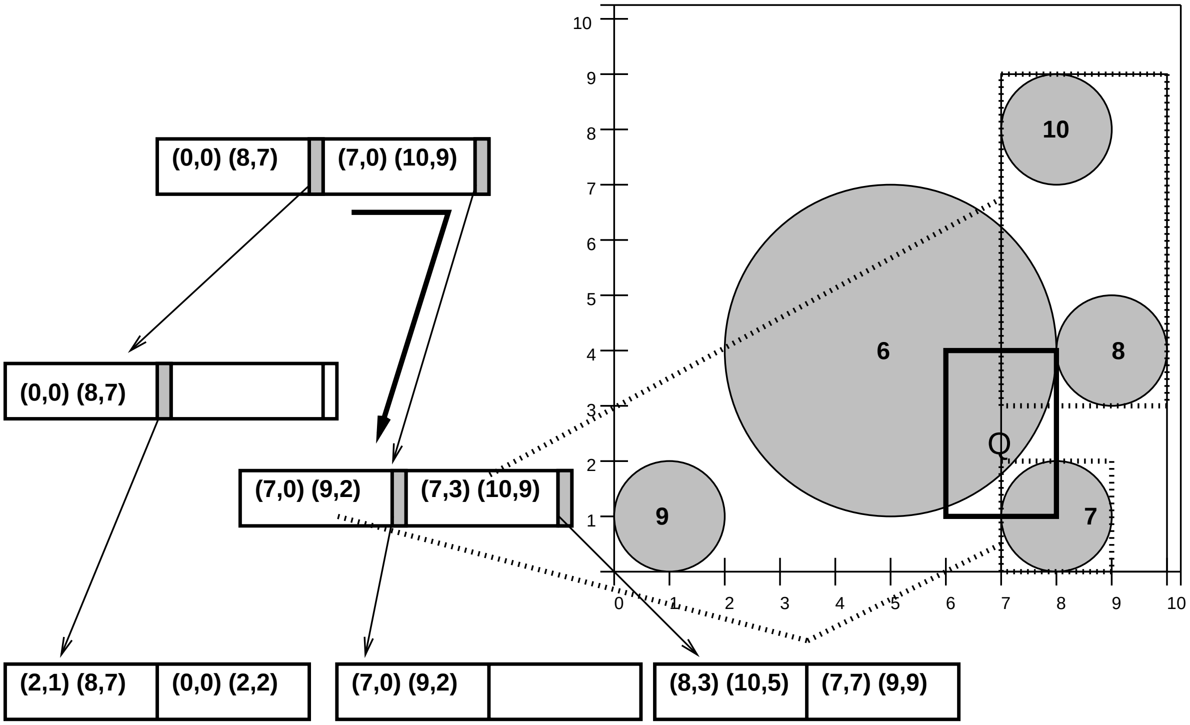
- Example: Q = `[(6,1) (8,4)]`.
  - Check MBR's in root
    ⇒ Q intersects both MBR's
    ⇒ must explore both subtrees.



  - Explore first subtree (child of `[(0,0) (8,7)]`).
  - Check MBR's at next level
    ⇒ Q intersects `[(0,0) (8,7)]`
    ⇒ must explore subtree of `[(0,0) (8,7)]`.
  - Check MBR's of child of `[(0,0) (8,7)]`:
    ⇒ Q intersects `[(2,1) (8,7)]`
    ⇒ return corresponding datapointer.



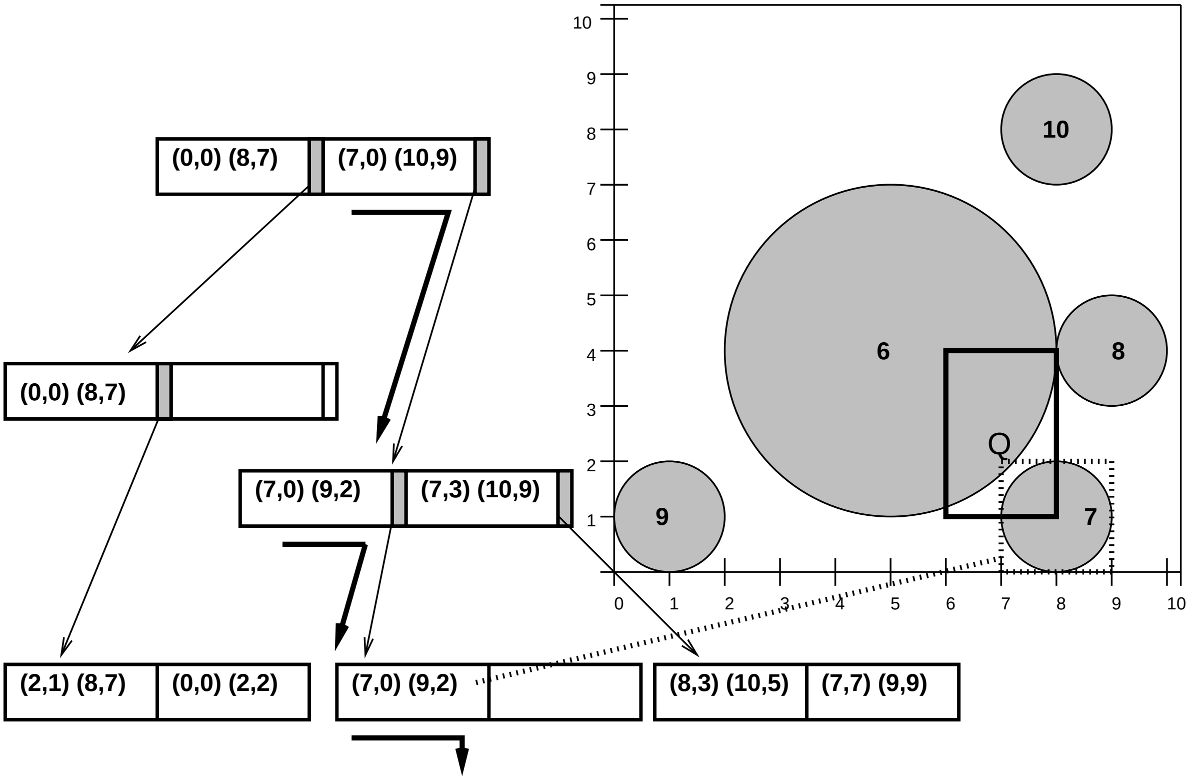
- Are there unexplored subtrees?
  ⇒ yes, right subtree of root.
- Check subtree of `[(7,0) (10,9)]`.
  ⇒ Q intersects both MBR's
  ⇒ must explore both subtrees.



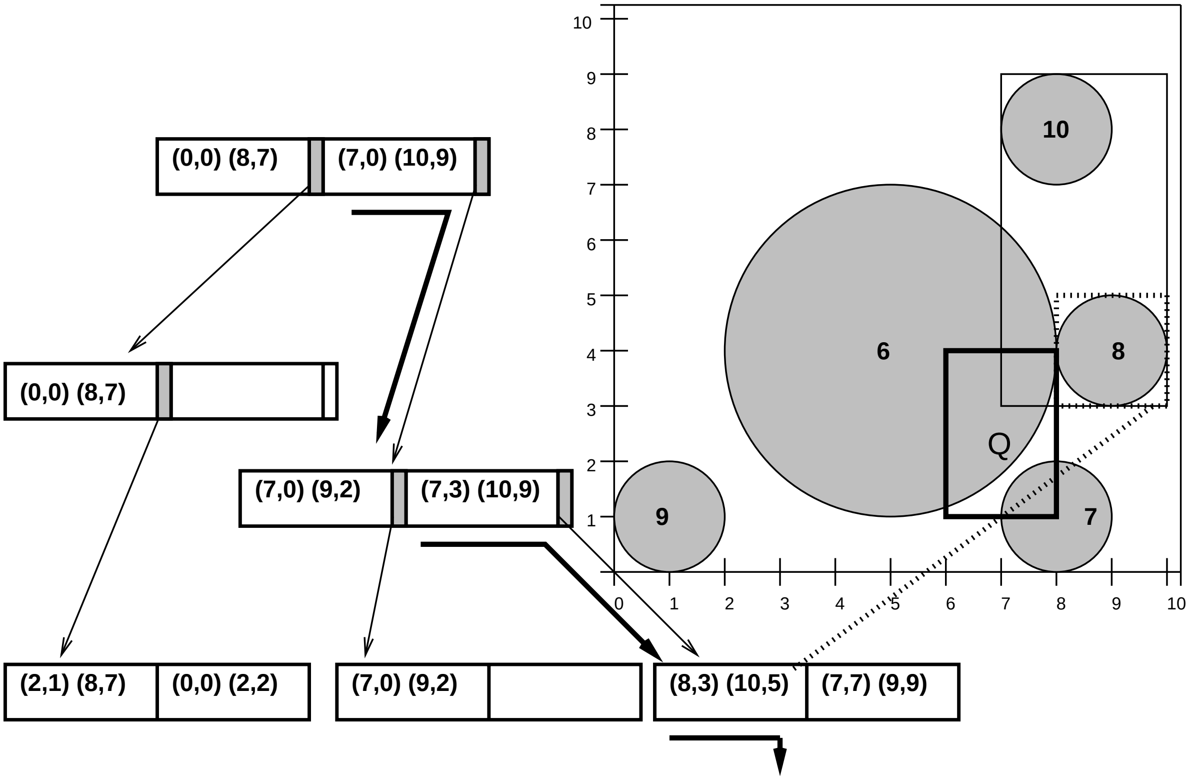
– Check subtree `[(7,0) (9,2)]`

⇒ Q intersects MBR `[(7,0) (9,2)]`

⇒ return corresponding datapointer.



- Any unexplored subtrees?
  - ⇒ check `[(7,3) (10,9)]` at the second level
  - ⇒ check MBR's at leaf level
  - ⇒ Q intersects (touches) MBR `[(8,3) (10,5)]`.
  - ⇒ return corresponding datapointer.



- Summary: datapointers for circles 6,7 and 8 are returned.

- The above examples illustrate what's good and bad about an R-tree:
  - An R-tree works well when a query rectangle interects only a few high-level MBR's
    ⇒ search is similar to a B+-tree.
  - An R-tree works poorly when a query intersects many high-level MBR's
    ⇒ lots of subtrees must be explored
    ⇒ cannot provide $O(\log n)$ minimum performance.
  - Worst-case, the whole tree might have to be searched.

- Pseudocode for search:

Algorithm: Rtree-Search (Q)

Input: Query rectangle Q.
Output: List of MBR's (with associated datapointers).

```
Initialize MBR-list.
Rtree-Recursive-Search (root, Q);
```

Algorithm: Rtree-Recursive-Search (blknum, Q)

Input: block number blknum, query rectangle Q.
Output: if a leaf MBR in blknum intersects Q, add it to MBR-list.

```
$b$ := Disk-Readblock (blknum);
for $i \leftarrow 1$ to $b \rightarrow$num_entries
   if Q intersects $b \rightarrow$R[$i$]
      if $b \rightarrow$leaf = true
         Add R[$i$] to MBR-list;
      else
         Rtree-Recursive-Search ($b \rightarrow$child[$i$], Q);
   endif
endfor
return;
```

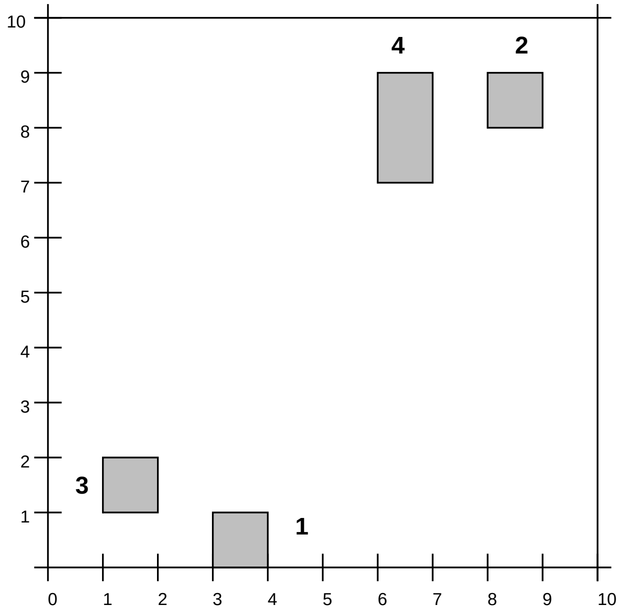
Algorithm: RTREE-GETNEXT

Input: none.
Output: next MBR in MBR-list if one exists, NULL otherwise.

```
if MBR-list is empty
    return null;
else
    return next MBR in list;
```

11.7 Rtrees: Discussion

- Better splitting algorithms:
  - The splitting algorithm above might as well be called *Random* since no effort is made to separate MBR's prudently.
  - Consider the following example (with $M = 3$):



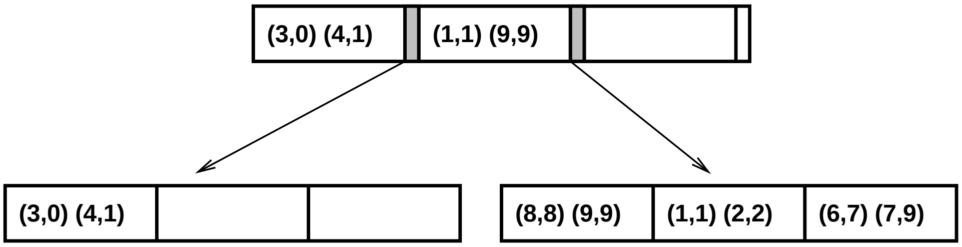
  Insert data MBR's 1,2,3, and 4.

  - After inserting MBR's 1,2 and 3, we get:

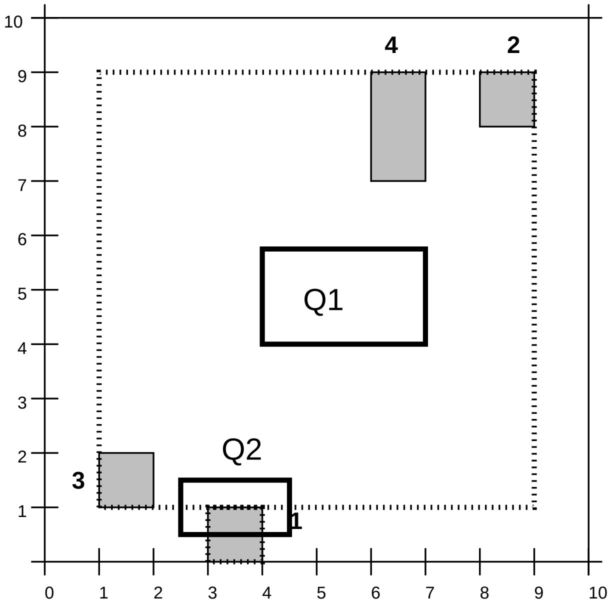
(3,0) (4,1)	**(8,8) (9,9)**	**(1,1) (2,2)**

  - The insertion of MBR `[(6,7) (7,9)]` causes a split.

– Using the simple splitting algorithm:



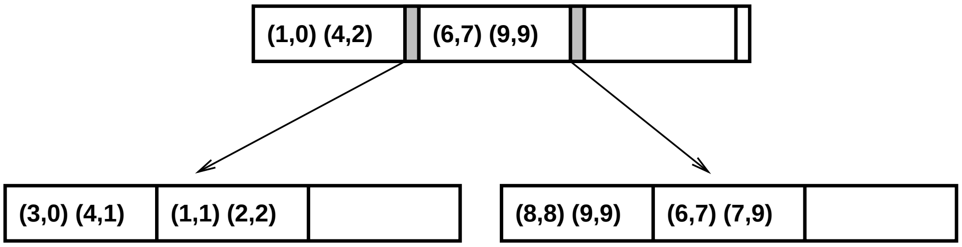
This corresponds to the parent MBR's:



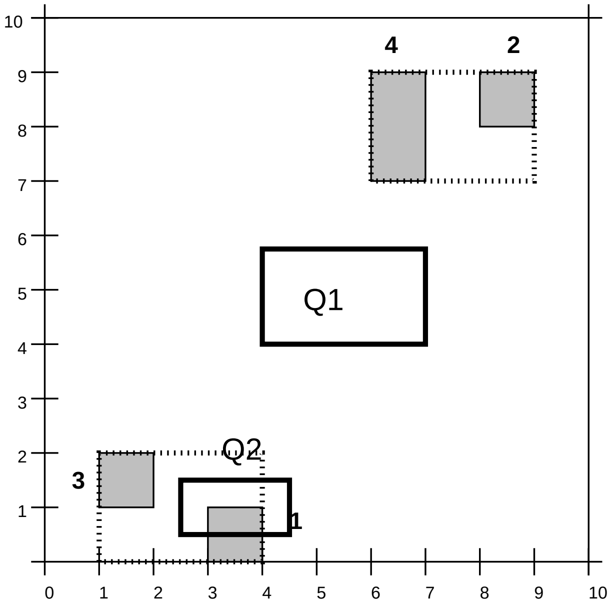
Consider the queries:

- ∗ Query Q1 (unnecessarily) searches the right subtree.
- ∗ Query Q2 searches both subtrees.

– Instead, consider a different split during the insertion of MBR 4:



which corresponds to the parent MBR's:



– Q1 does not need searching of subtrees.

– Q2 searches only one subtree.

– What was the problem with the simple method?
⇒ rectangles were randomly distributed among the split nodes.

– Is it possible to do better?

- *Quadratic split algorithm*:
  - Consider a node to be split.
  - For each pair of rectangles, R_i and R_j:
    - compute MBR of the pair R_i, R_j;
    - compute area of pair-MBR
    - compute joint area of R_i and R_j;
    - compute deadspace area:

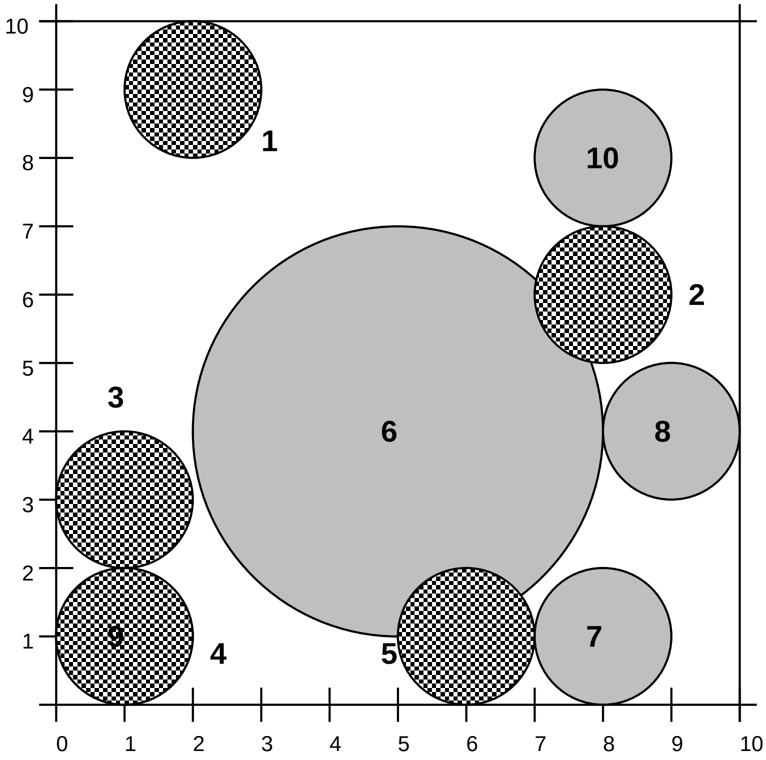
      Area(pair-MBR) - (Area(R_i) + Area(R_j));
  - Pick pair with largest such difference.
  - Place these rectangles in different nodes.
  - For each remaining rectangle:
    - Compute resulting MBR if it were placed in each block.
    - Place it in block whose MBR needs least enlargement to accomodate it.
- Note identifying the pair takes quadratic (in the number of entries per node) time (why?).
- Similar linear heuristics also exist.

- Variants of R-trees:
  - R+-trees:
    - In an R-tree, MBR's overlap
      $\Rightarrow$ multiple overlaps could cause searching multiple subtrees.
    - R+-trees eliminate overlap.
    - Instead, object MBR's themselves are split and duplicated at the leaf-level.
    - While R+-trees do not overlap MBR's, they however duplicate object MBR's
      $\Rightarrow$ a tradeoff.
    - R+-trees work well in some applications.
  - R*-trees:
    - When a node is split, MBR's are re-inserted into the R-tree
      $\Rightarrow$ greater (tree-wide) reorganization during split.
    - In practice, R*-trees are thought to work slightly better than R-trees and R+-trees.
  - Hilbert-Rtrees:
    - Similar to R-tree except that Hilbert-values of the MBR's are used *in addition.*
    - Define the *Hilbert-value* of an MBR to be the Hilbert value of its center.
    - Each data MBR descriptor also contains its Hilbert-value.
    - Each interior node of the tree also contains the largest Hilbert value among all the MBR's in the node.
      (Define LHV = largest Hilbert-value of an interior node).
    - *Search* is exactly the same as in the R-tree.
    - *Insertion* differs slightly from R-tree insertion:
      - In an R-tree, the child whose MBR needs the least expansion is selected.
      - In a Hilbert R-tree, the child whose MBR's Hilbert value is closest to the input MBR's Hilbert value is selected.

        - When working back up the tree to update MBR's, the LHV values are re-computed up the tree.

- Bulk-loading of R-trees:
    - Given a large collection of data objects, it is inefficient to load the R-tree by successive insertions.
    - Bulk-loading algorithms:
        - are faster;
        - achieve faster packing.
    - Various bulk loading algorithms exist
      ⇒ most arrange MBR's according to their centers.

- Spatial joins using R-trees:
  - What is a *spatial join*?
    - ∗ Consider two files of objects, F_A and F_B.
    - ∗ A spatial join based on intersection (overlap) is a list of pairs of objects, one from each file, that overlap.
    - ∗ We will use the convention that intersection includes “touching”.
    - ∗ Other spatial join operators are possible. For example, a *containment* join will have all those pairs of objects (O_A, O_B) where $O_A \in F_A, O_B \in F_B$ such that O_B is completely contained in O_A.
    - ∗ Example (intersection join):



FILE 1

Circle 1:	center at (2,9), radius 2
Circle 2:	center at (8,6), radius 2
Circle 3:	center at (1,3), radius 2
Circle 4:	center at (1,1), radius 2
Circle 5:	center at (6,1), radius 2

FILE 2

Circle 6:	center at (5,4), radius 3
Circle 7:	center at (8,1), radius 2
Circle 8:	center at (9,4), radius 2
Circle 9:	center at (1,1), radius 2
Circle 10:	center at (8,8), radius 2

The tuples that get joined are:

From first file	From second file
Circle 2	Circle 6
Circle 2	Circle 10
Circle 3	Circle 9
Circle 4	Circle 9
Circle 5	Circle 6
Circle 5	Circle 7

– The straightforward algorithm is a nested loop (as in the case of relational databases).

– The R-tree can be used effectively by scanning one file (say, File 1) tuple-by-tuple, and using an R-tree for the second file:

  ∗ Insert items from File 2 into R-tree.

  ∗ Scan tuples in File 1.

  ∗ For each tuple (object) scanned in File:

    · Compute MBR of object.

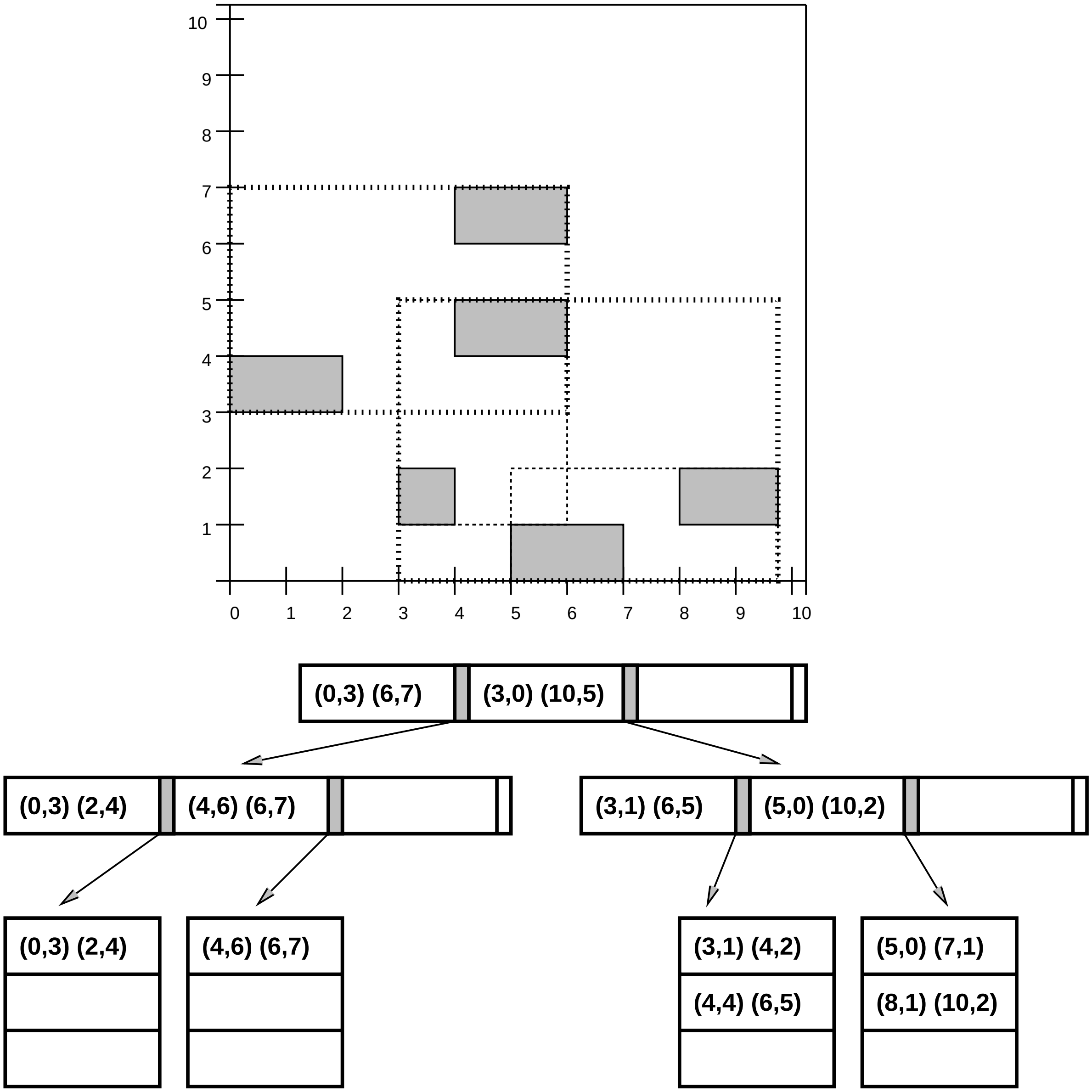
    · Use this MBR as query rectangle in R-tree.

    · Retrieve list of intersected object-MBR's.

    · Test for true intersection.

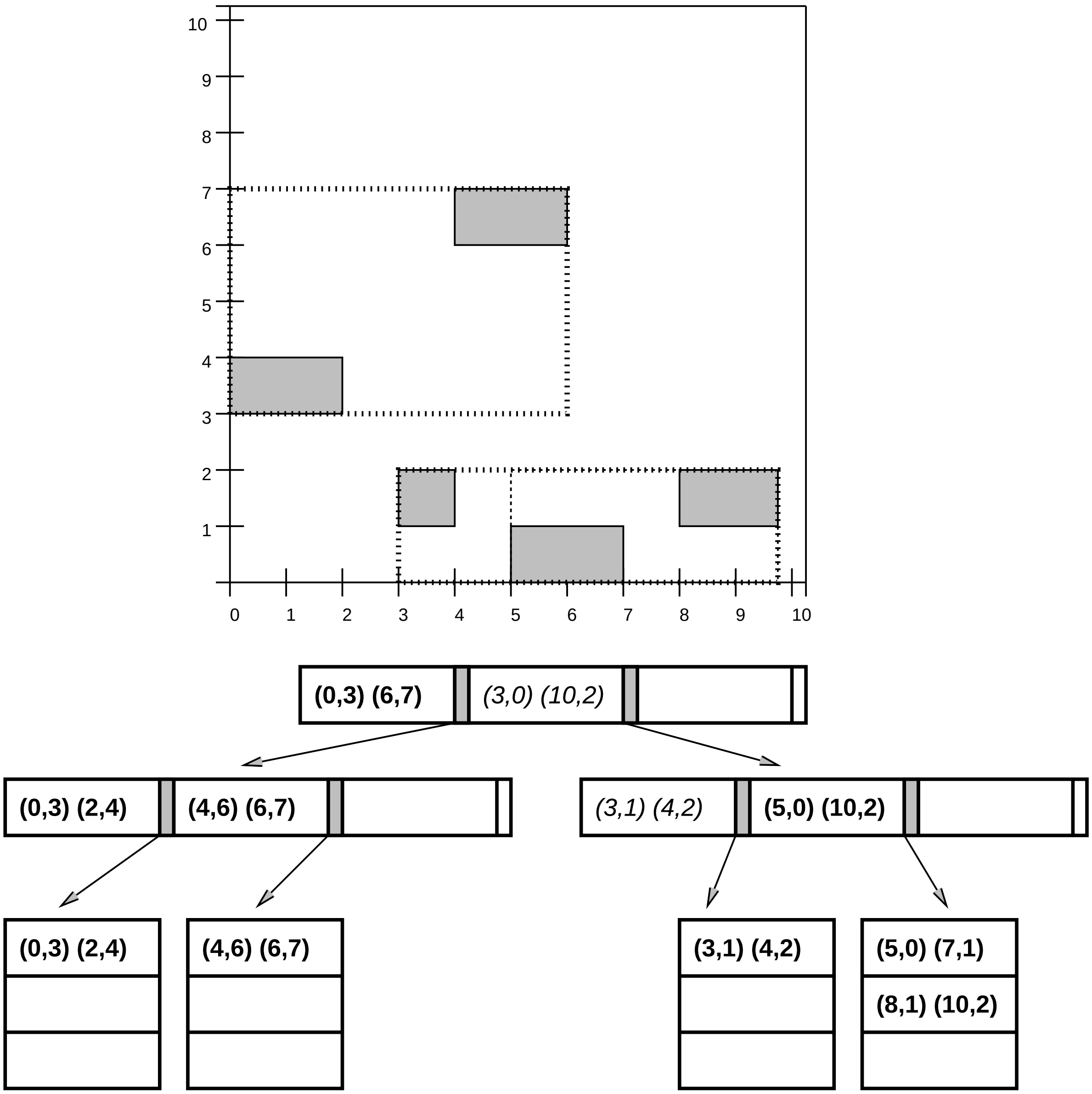
11.8 Deletion in an R-tree

- Several deletion methods are known. We will consider only one.
- Key ideas:
  - Find object MBR using search and delete from leaf
    ⇒ Leaf may or may not underflow.
  - If leaf does *not* underflow:
    - Compute block MBR of leaf after deletion.
    - Adjust this value in the parent.
    - Now, the parent's own MBR has changed
      ⇒ adjust MBR entry in grandparent...
    - ... and so on, recursively all the way to the root.
  - If leaf underflows:
    - Try to borrow from (any) sibling:
      - If possible, re-arrange MBR's with sibling to get more compact block-MBR's (as we did in splitting).
      - Recompute MBR's all the way up to the root.
    - If borrow is not possible, merge with sibling if possible:
      - Recompute block MBR of merged node.
      - Propagate MBR's to root.
    - If no sibling is available, collapse one level.
    - If merging causes underflow at parent's level, merging may need to be propagated upwards.
- Example with $M = 3$
  ⇒ at least $\lfloor \frac{M}{2} \rfloor = 1$ entries per node.



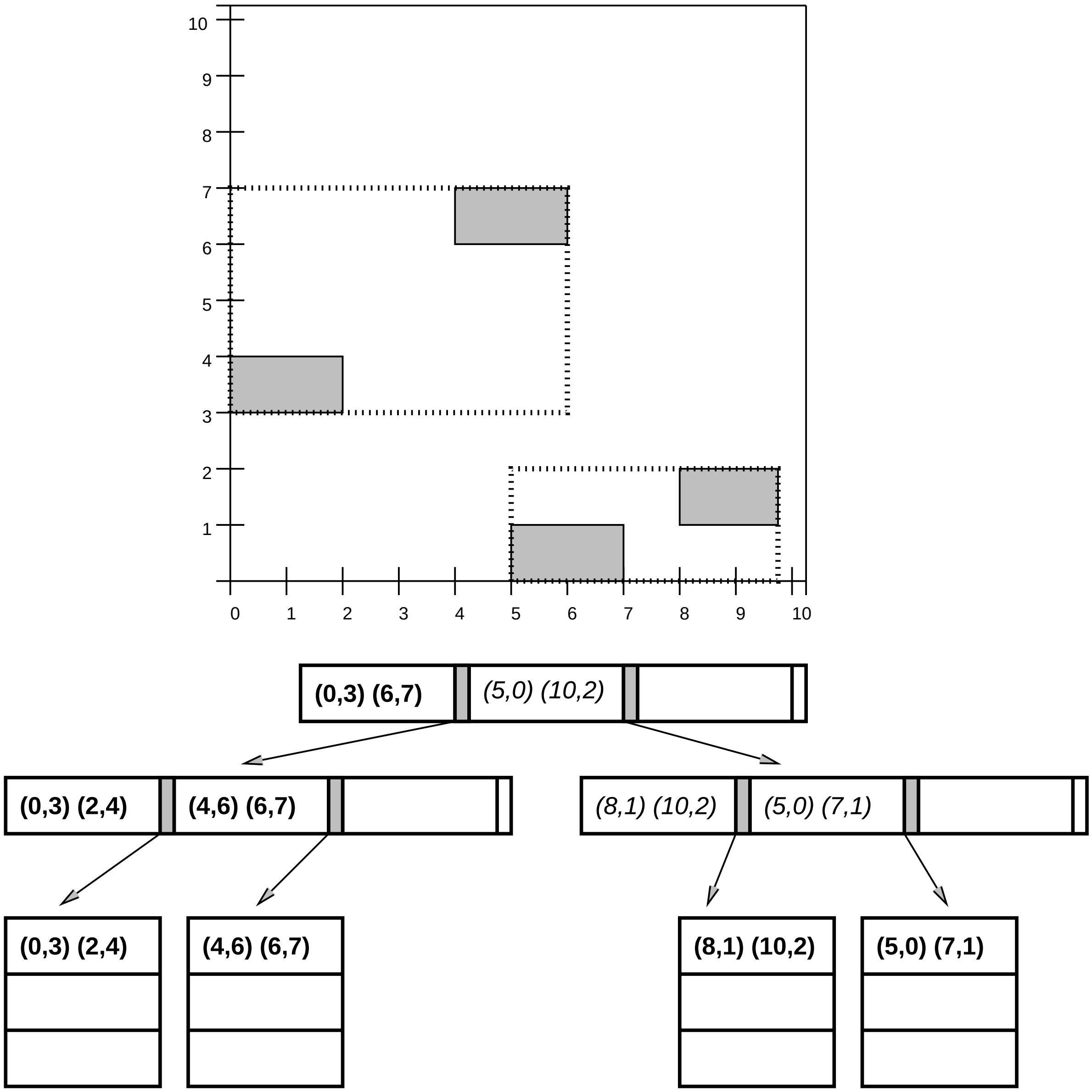
- Delete `[(4,4) (6,5)]`:
  - Delete at leaf level.
  - Adjust node MBR: `[(3,1) (4,2)]`.
  - Adjust at parent's level: `[(3,1) (4,2)]`.
  - Adjust parent's MBR
    ⇒ compute MBR of `[(3,1) (4,2)]` and `[(5,0) (10,2)]`
    ⇒ `[(3,0) (10,2)]`.

- Adjust at next level (root).
- Root reached ⇒ done.

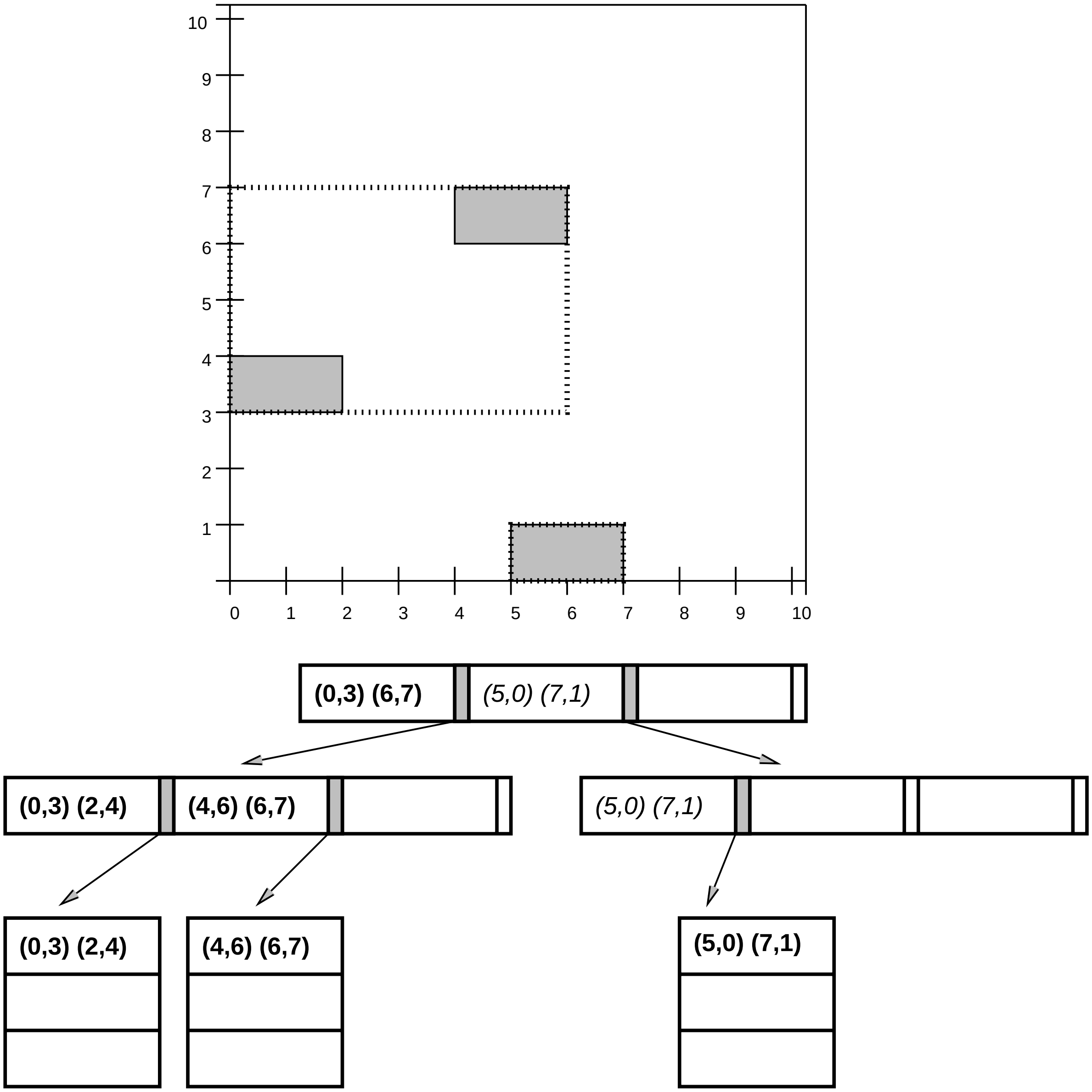


- Delete `[(3,1) (4,2)]`:
  - Delete at leaf
    ⇒ underflow.
  - Borrow from sibling (borrow `[(8,1) (10,2)]`).

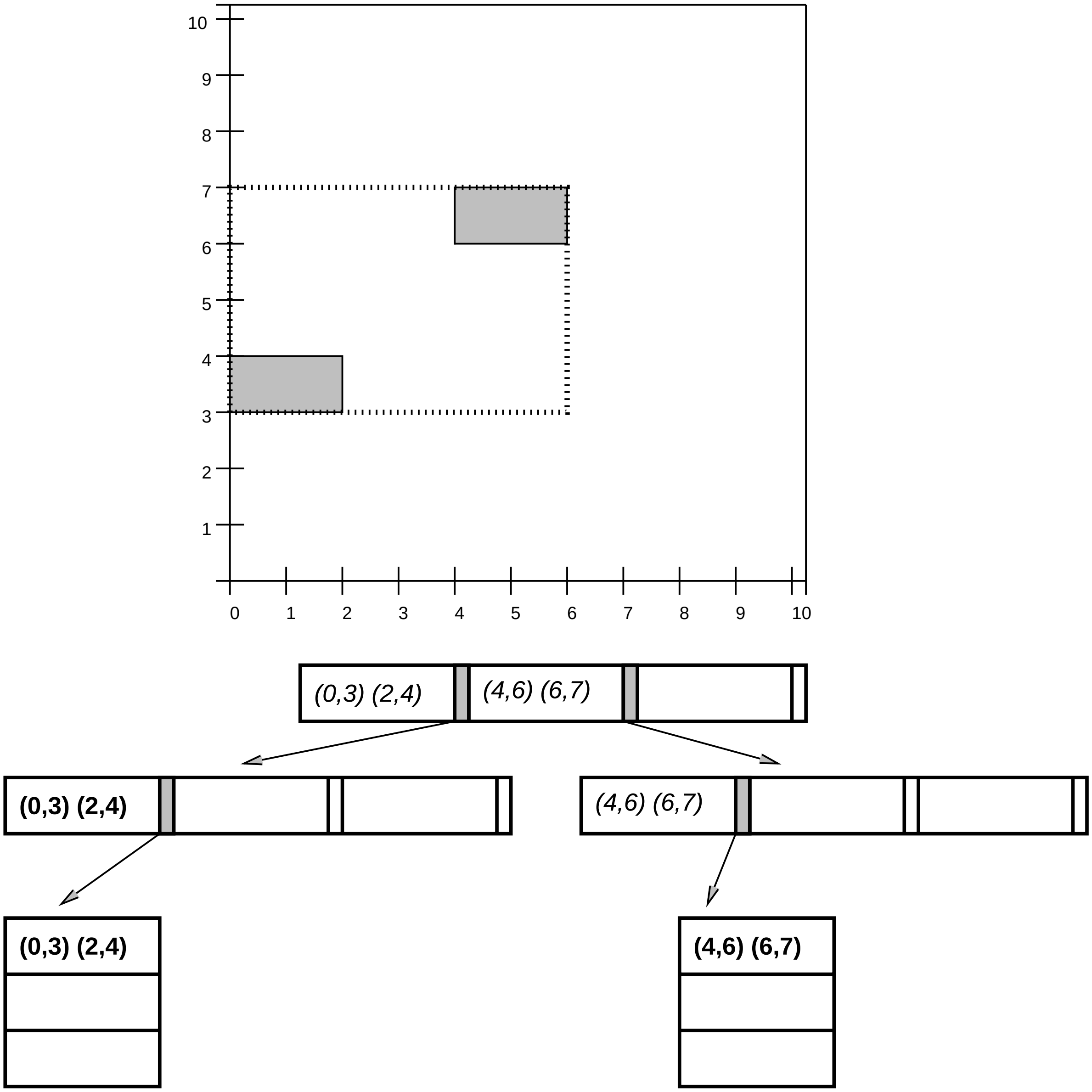
- Recompute MBR's at parent's level
  ⇒ [(8,1) (10,2)] and [(5,0) (7,1)].
- Recompute parent's MBR
  ⇒ [(5,0) (10,2)].
- Re-write this value at grandparent's level:



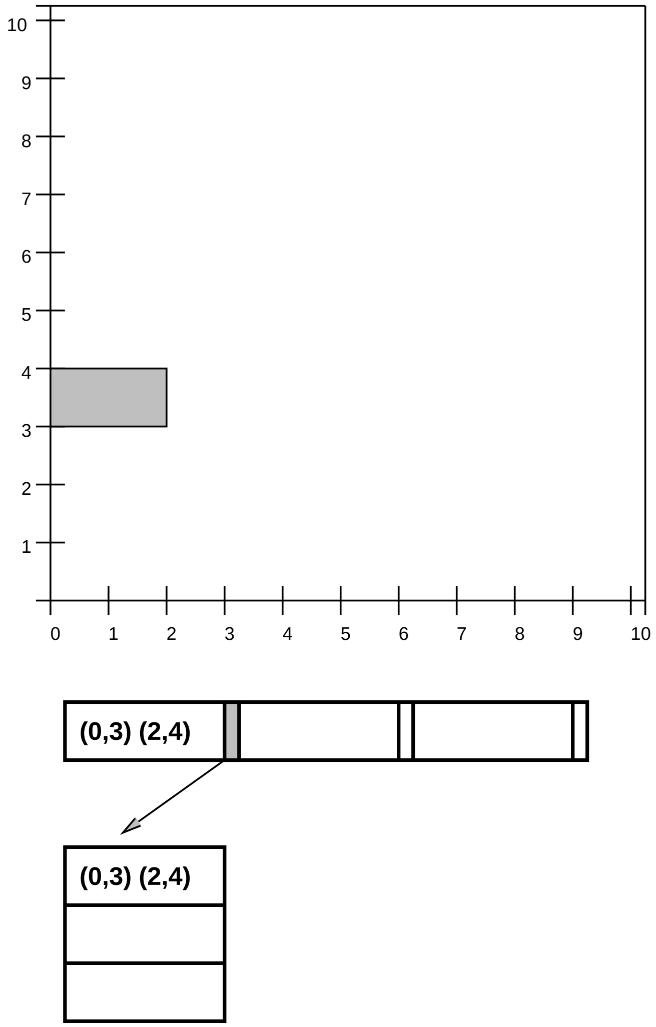
- Delete `[(8,1) (10,2)]`:
  - Delete from leaf
    ⇒ underflow.
  - Cannot borrow from sibling
    ⇒ merge.
  - Recompute MBR's all the way up.



- Delete `[(5,0) (7,1)]`:
  - Delete from leaf
    ⇒ underflow.
  - No sibling
    ⇒ borrow at parent's level.
  - Recompute MBR's as needed.



- Delete `[(4,6) (6,7)]`:
  - Delete at leaf
    ⇒ underflow.
  - Cannot borrow or merge
    ⇒ look at parent's level.
  - Merge at parent's level.



11.9 Grid Files: Introduction

- A *grid file* is an index structure for multidimensional queries – queries on multiple attibutes.
- Recall: B-trees, B+-trees and Hashing were used for unidimensional queries.
- Example of a multidimensional query:

  Consider the relation

  PASSENGER (NAME, SSN, FLT_ID, MILES)

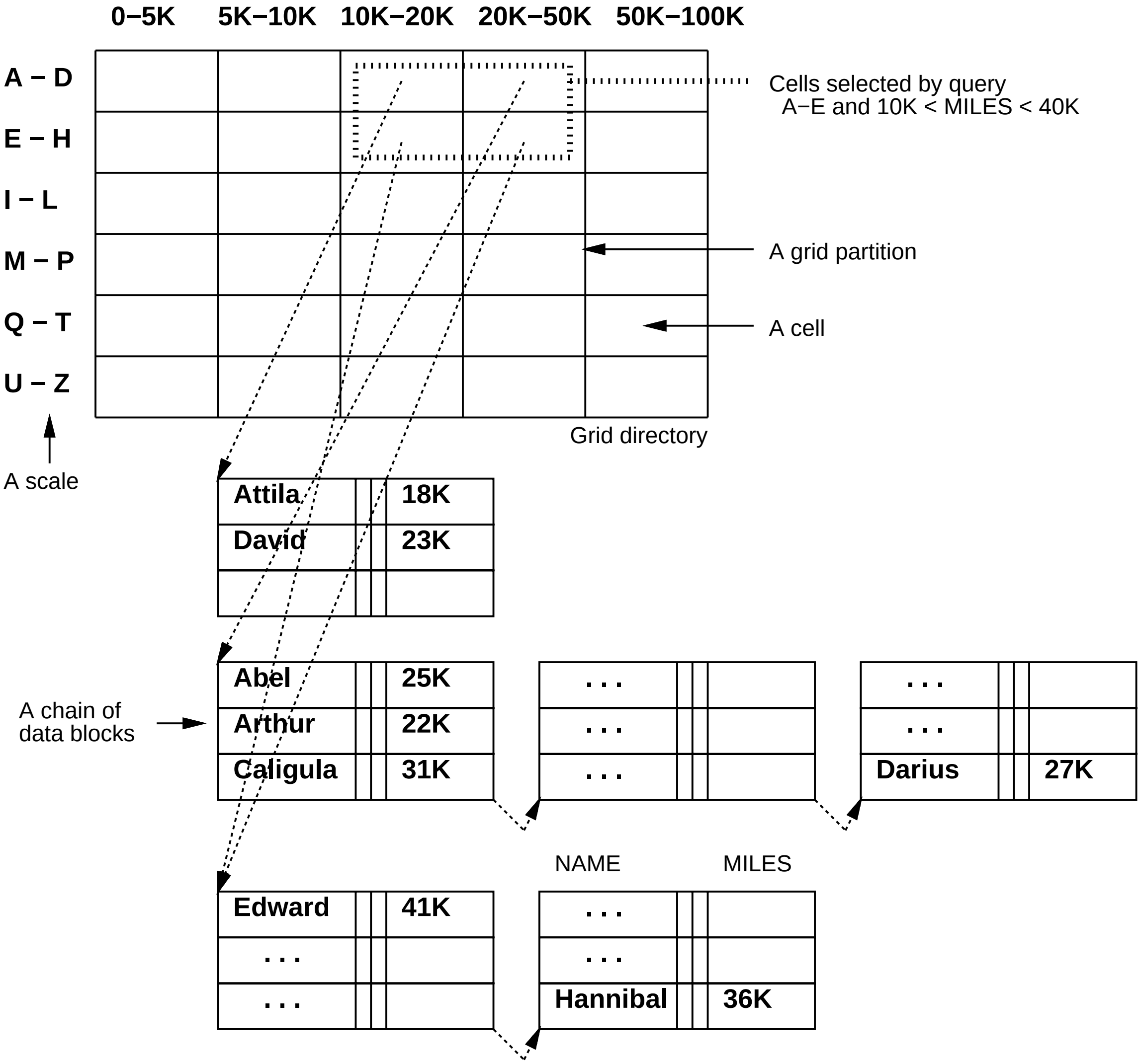
  and the query: "Find all passengers with names that begin with letters in A–E and have between 10000 and 40000 miles".
  In SQL:

  ```
  select  P.NAME, P.MILES
  from    PASSENGER P
  where   P.NAME >= 'A' and P.NAME < 'F'
          and P.MILES >= 10000 and P.MILES <= 40000;
  ```

  Note: this is sometimes called a *multikey* query even though NAME and MILES are not keys.
- Suppose we use a B+-tree on NAME:
  - Suppose PASSENGER has 100,000 tuples with 20 tuples per block ⇒ 5000 blocks.
  - About 4-5 levels of the B+-tree.
  - Suppose there are 20000 tuples in the A–E range ⇒ 1000 data blocks if clustered.
  - Also, scanning the leaf level of B+-tree will need, say, 400 blocks.
  - The 1000 data blocks have to be tested for the MILES condition.

- Suppose only 5 blocks satisfy the MILES condition
  ⇒ lot of I/O for only a 5-block result.

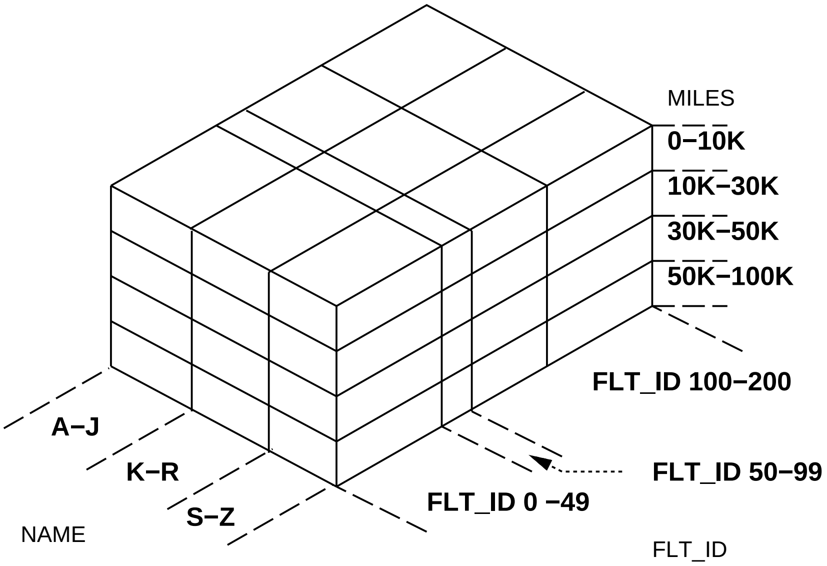
- Key idea in a grid file:
  - Create a grid directory:



  - Perform range search (A–E, 10000-40000) on each axis to determine which cells apply.
  - Use pointers in cells to get to data.

- A *grid file* consists of:
  - A *grid directory*:
    - ∗ A collection of grid cells.
    - ∗ Each cell contains a pointer to a chain of data blocks.
  - One *grid scale* per dimension:
    - ∗ A scale consists of the intervals in each dimension.
      ⇒ e.g., the NAME scale above has A-D, E-H etc.
    - ∗ Note: interval boundaries mark grid partitions.
  - Chains of data blocks:
    - ∗ Each cell points to a data block chain.
    - ∗ Multiple cells may point to the same chain.
  - Additional meta-data such as the number of data blocks, the number of tuples etc.
- Main principle in grid files: choose partitions judiciously
  ⇒ keep chains small.

  Note: many implementations allow at most one block per chain.
- Grid files can be of more than 2 dimensions:



- Grid files can be used for geometric point data. We will consider 2D point data.

- Grid files support insertion, search and deletion of point data.

- Grid files support range queries efficiently.

- There are two basic types of grid file:

  1. Grid files with fixed-size cells.
  2. Grid files with variable-size cells.

- There are many different algorithms for insertion and deletion within the class of grid files.

11.10 Grid Files: Insertion

- We will study insertion via an example:
  - Insert the following point data:

p_1	(0.35,0.37)	p_6	(0.82,0.87)	p_{11}	(0.31,0.24)
p_2	(0.74,0.23)	p_7	(0.25,0.53)	p_{12}	(0.94,0.35)
p_3	(0.55,0.81)	p_8	(0.24,0.77)	p_{13}	(0.82,0.15)
p_4	(0.82,0.66)	p_9	(0.23,0.35)	p_{14}	(0.05,0.12)
p_5	(0.75,0.78)	p_{10}	(0.25,0.24)	p_{15}	(0.03,0.24)

  - We will assume all the point data falls in the unit square.
    ⇒ i.e., assume MBR of the set of points is known.
  - Assume blocksize = 64 bytes.
  - Suppose we use

4 bytes	for x-coordinate (float)
4 bytes	for y-coordinate (float)
2 bytes	for a point ID (integer)
10 bytes	for a text label (character string)

    ⇒ totally 20 bytes
    ⇒ 3 point-tuples per block.

- We will first consider the fixed-size cell version of the grid file.

  The values *x-interval* and *y-interval* determine the size of each cell.

- Key ideas in inserting a point $p = (x_0, y_0)$:
  - Locate the cell that contains the point:
    * Divide x-value by *x-interval* and round up:

$$i \leftarrow \left\lceil \frac{x_0}{x\text{-}interval} \right\rceil$$

    - ∗ Divide y-value by *y-interval* and round up:

$$j \leftarrow \left\lceil \frac{y_0}{y\text{-}interval} \right\rceil$$

    - ∗ This gives us Cell$[i, j]$.
    - ∗ Example: $p = (0.74, 0.33)$, *x-interval*=*y-interval*=0.2

$$\left\lceil \frac{0.74}{0.2} \right\rceil = 4$$

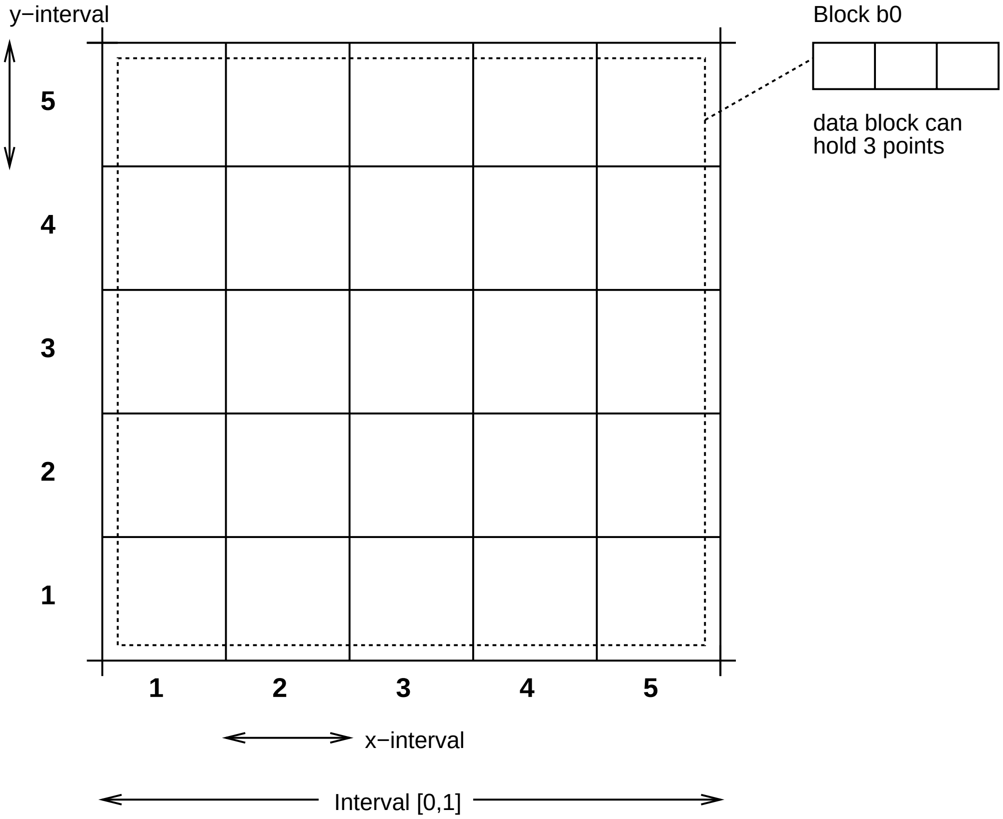
$$\left\lceil \frac{0.33}{0.2} \right\rceil = 2$$

      ⇒ Cell[4,2]

  - – Retrieve cell from grid directory.
  - – Retrieve data block pointed to by cell.
  - – If space available, insert point in block.
  - – Otherwise, block is split:
    - ∗ Try to use an existing partition.
    - ∗ If no existing partition does the job, create a new partition.
    - ∗ Insert point in appropriate block.
  - – Partitions or cuts are always made along cell boundaries.
  - – Sometimes, the grid needs to be refined
    ⇒ double grid size.
  - – Try to alternate between vertical and horizontal partitions:
    - ∗ Use a variable *Current* to indicate which direction.
    - ∗ If a vertical cut (partition) is made:
      ⇒ set *Current* := horizontal.
    - ∗ Initially, say, *Current* := vertical.

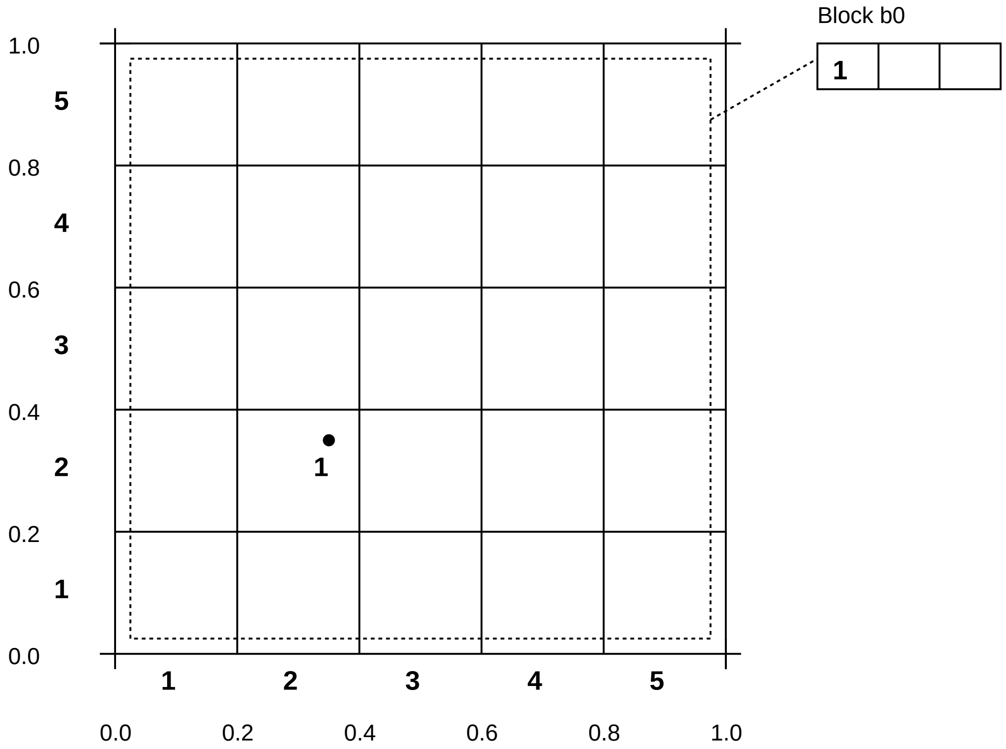
- The example will emphasize concepts; we will consider implementation details later.

- Initially:
  - Suppose we start with 5×5 grid in the unit square.
  - Take *x-interval*=*y-interval*=0.2.



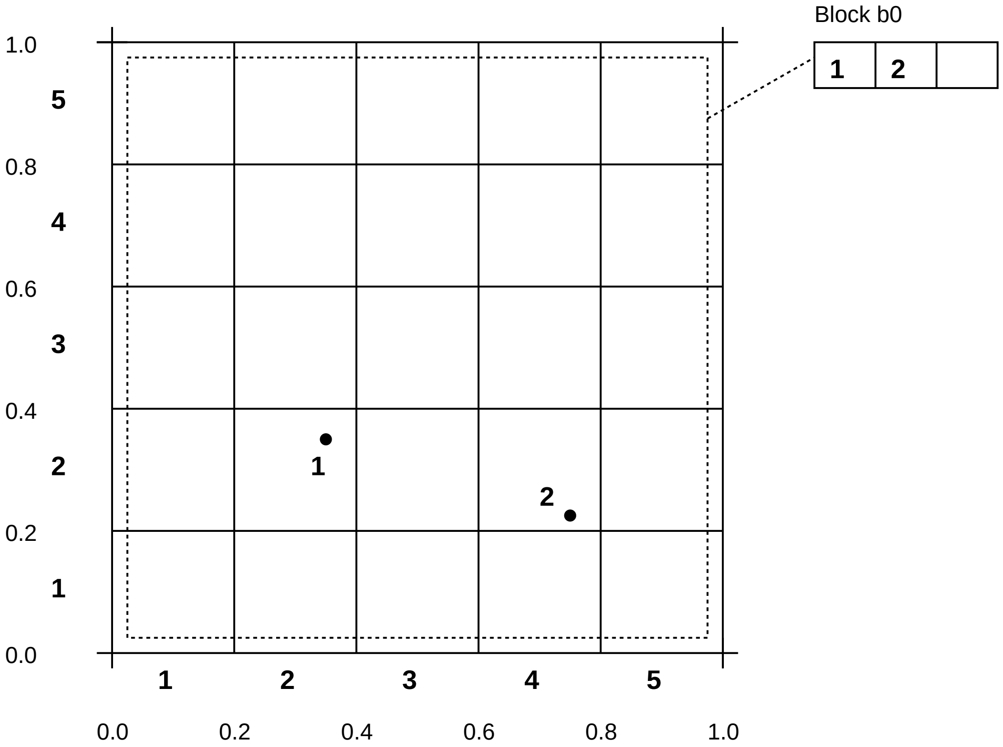
  - Initially, all grid cells point to a single data block, b_0.
  - *Current* := vertical.

- Insert $p_1 = (0.35, 0.37)$:
  - Locate cell: $\lceil\frac{0.35}{0.2}\rceil = 2$, $\lceil\frac{0.37}{0.2}\rceil = 2$
    ⇒ Cell[2,2].
  - Cell points to block b_0
    ⇒ retrieve block b_0.
  - Space available
    ⇒ insert point in block.

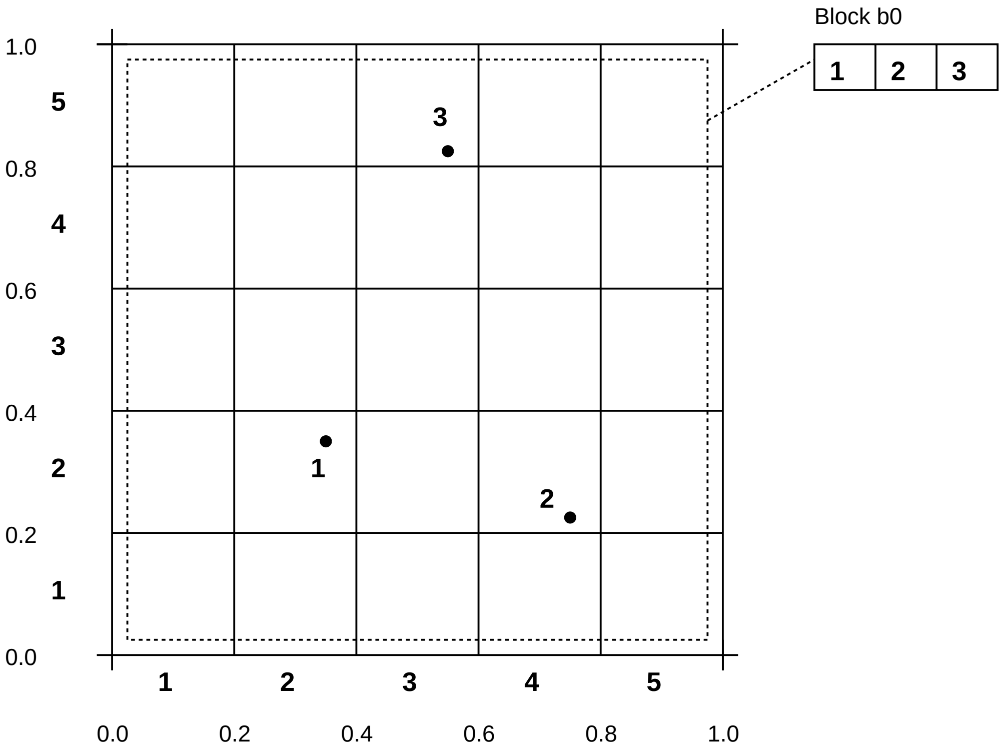


Note: The point is shown in Cell[2,2] only for conceptual ease
⇒ only the data block contains the point.

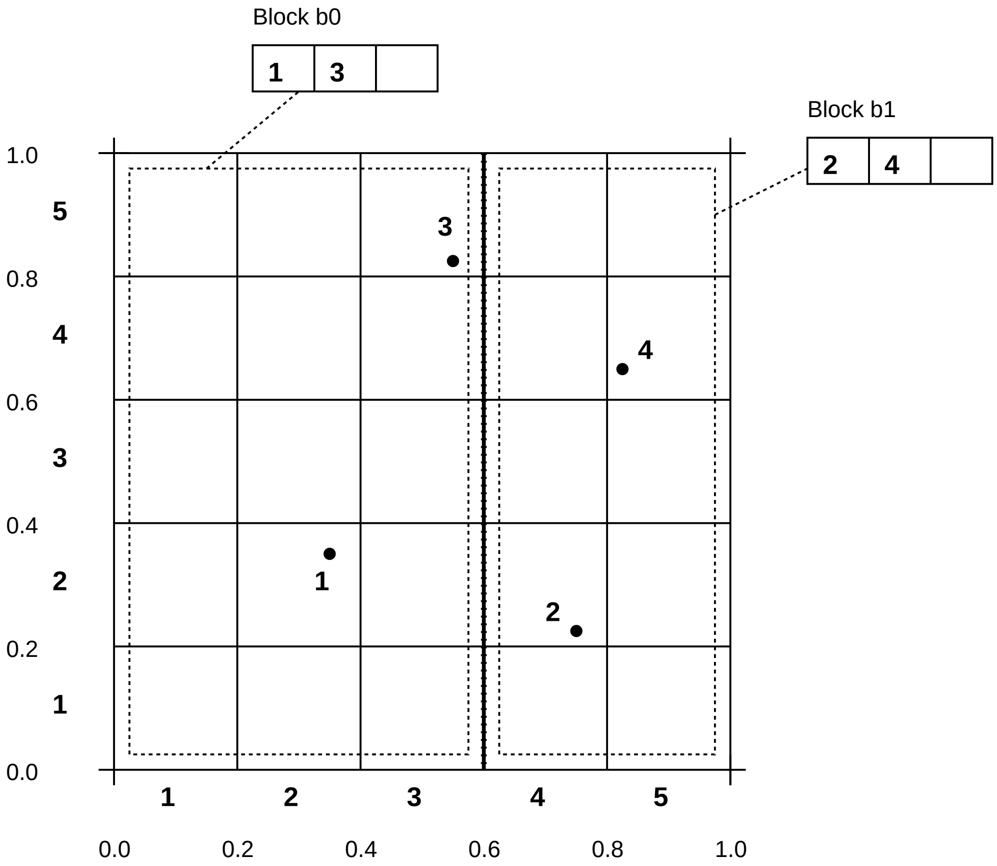
- Insert $p_2 = (0.74, 0.23)$:
  - Locate cell: $\left\lceil \frac{0.74}{0.2} \right\rceil = 4$, $\left\lceil \frac{0.23}{0.2} \right\rceil = 2$
    ⇒ Cell[4,2].
  - Cell points to block b_0
    ⇒ retrieve block b_0.
  - Space available
    ⇒ insert point in block.



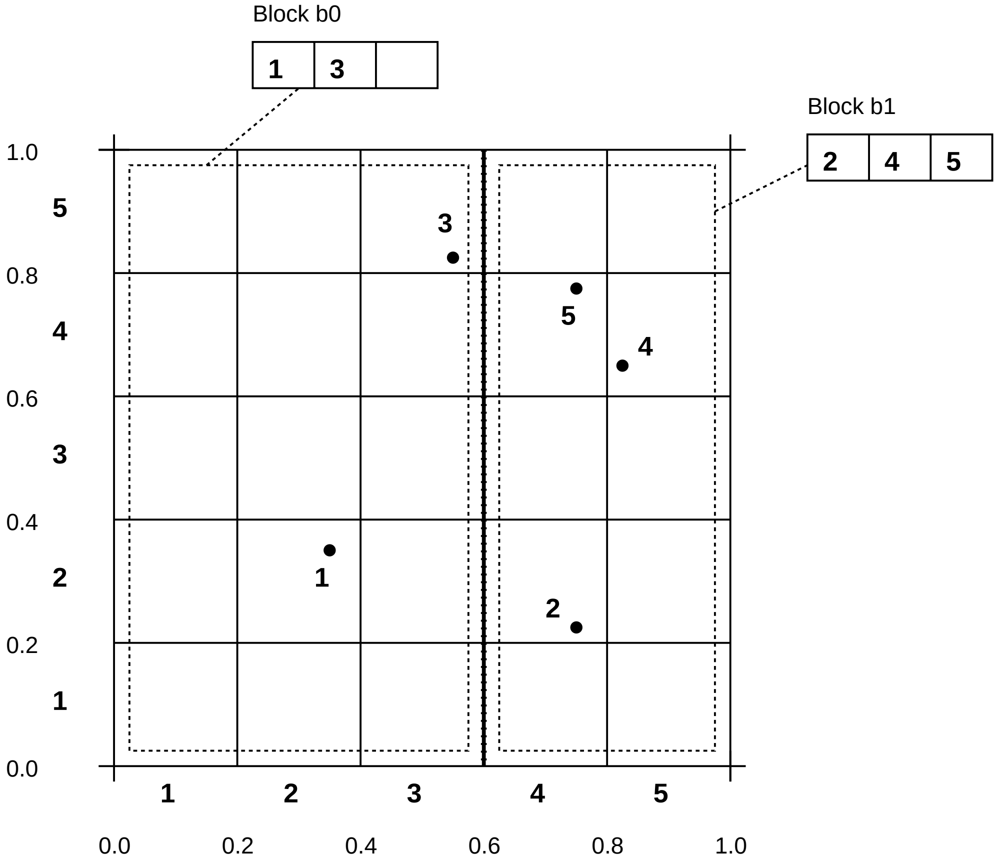
- Insert $p_3 = (0.55, 0.81)$:
  - Locate cell: $\lceil \frac{0.55}{0.2} \rceil = 3$, $\lceil \frac{0.81}{0.2} \rceil = 5$
    $\Rightarrow$ Cell[3,5].
  - Cell points to block b_0
    $\Rightarrow$ retrieve block b_0.
  - Space available
    $\Rightarrow$ insert point in block.



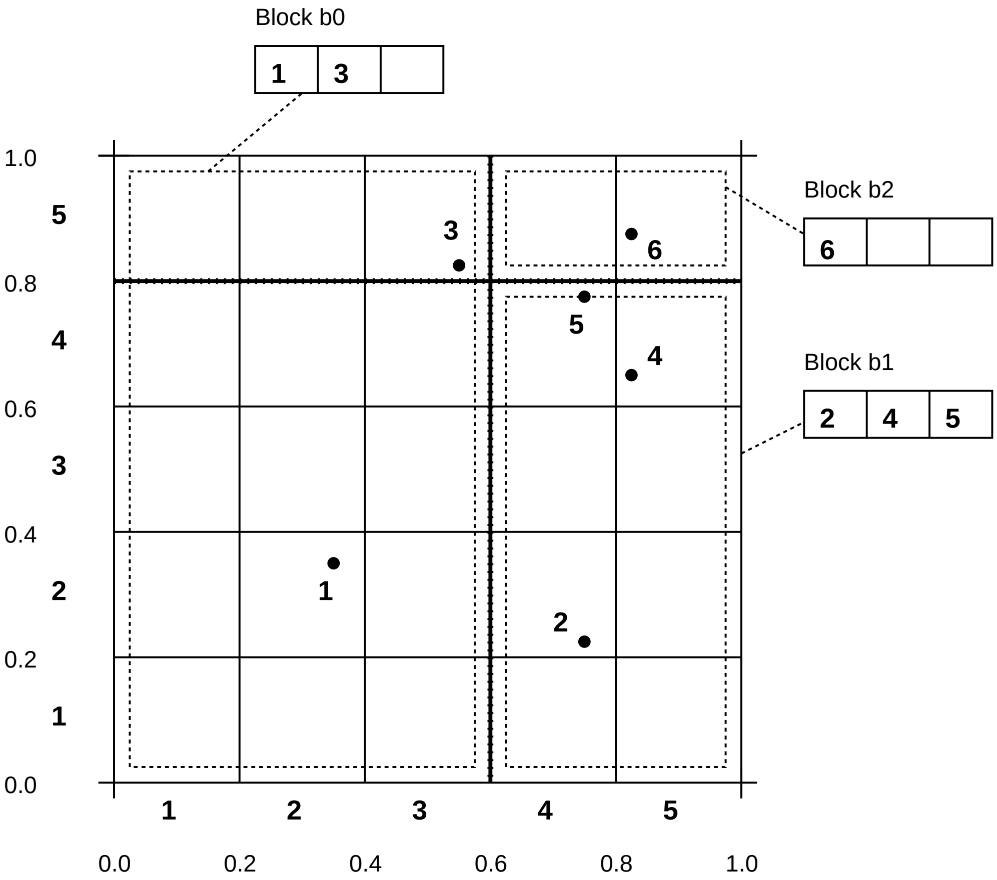
- Insert $p_4 = (0.82, 0.66)$:
  - Locate cell: $\left\lceil \frac{0.82}{0.2} \right\rceil = 5$, $\left\lceil \frac{0.66}{0.2} \right\rceil = 4$
    $\Rightarrow$ Cell[5,4].
  - Cell points to block b_0
    $\Rightarrow$ retrieve block b_0.
  - Block b_0 is full
    $\Rightarrow$ split needed.
  - No existing partition available
    $\Rightarrow$ create new partition.
  - *Current* = vertical
    $\Rightarrow$ make a vertical partition to approximately halve the total number of points (4)
  - Note: consider new point when making partition.
  - *Current* := horizontal.



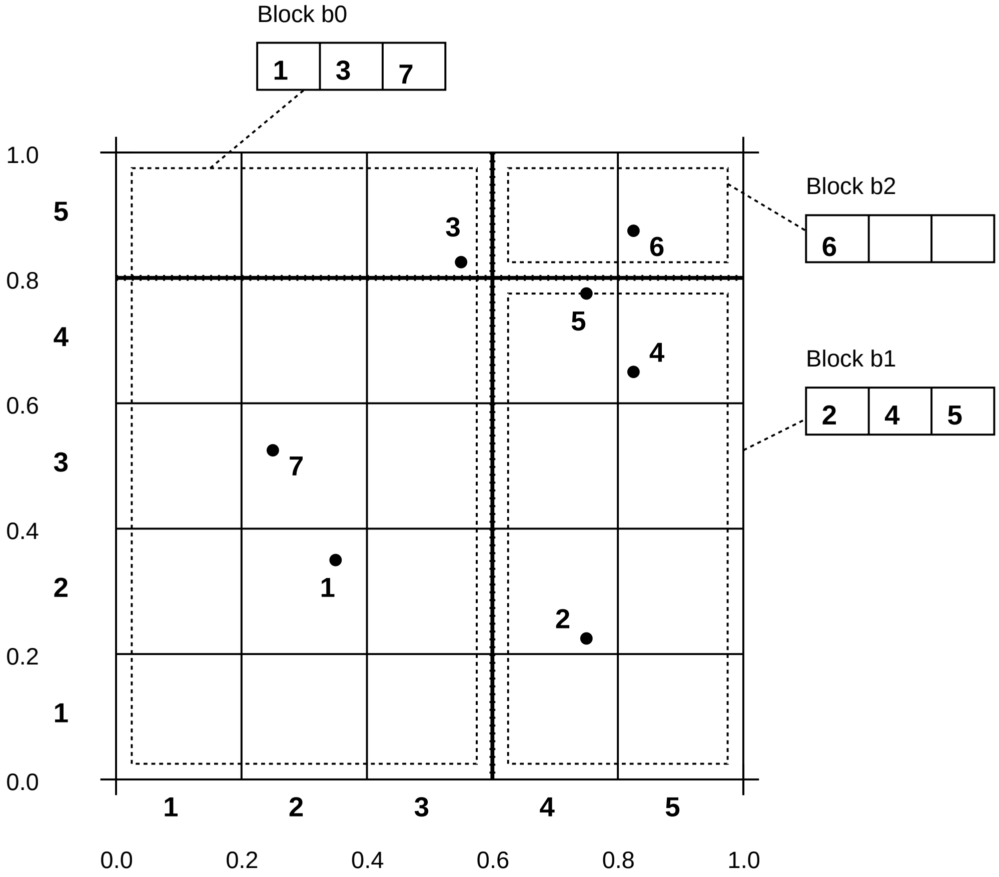
- Insert $p_5 = (0.75, 0.78)$:
  - Locate cell: $\left\lceil \frac{0.75}{0.2} \right\rceil = 4$, $\left\lceil \frac{0.78}{0.2} \right\rceil = 4$
    $\Rightarrow$ Cell[4,4].
  - Cell points to block b_1
    $\Rightarrow$ retrieve block b_1.
  - Space available
    $\Rightarrow$ insert point.



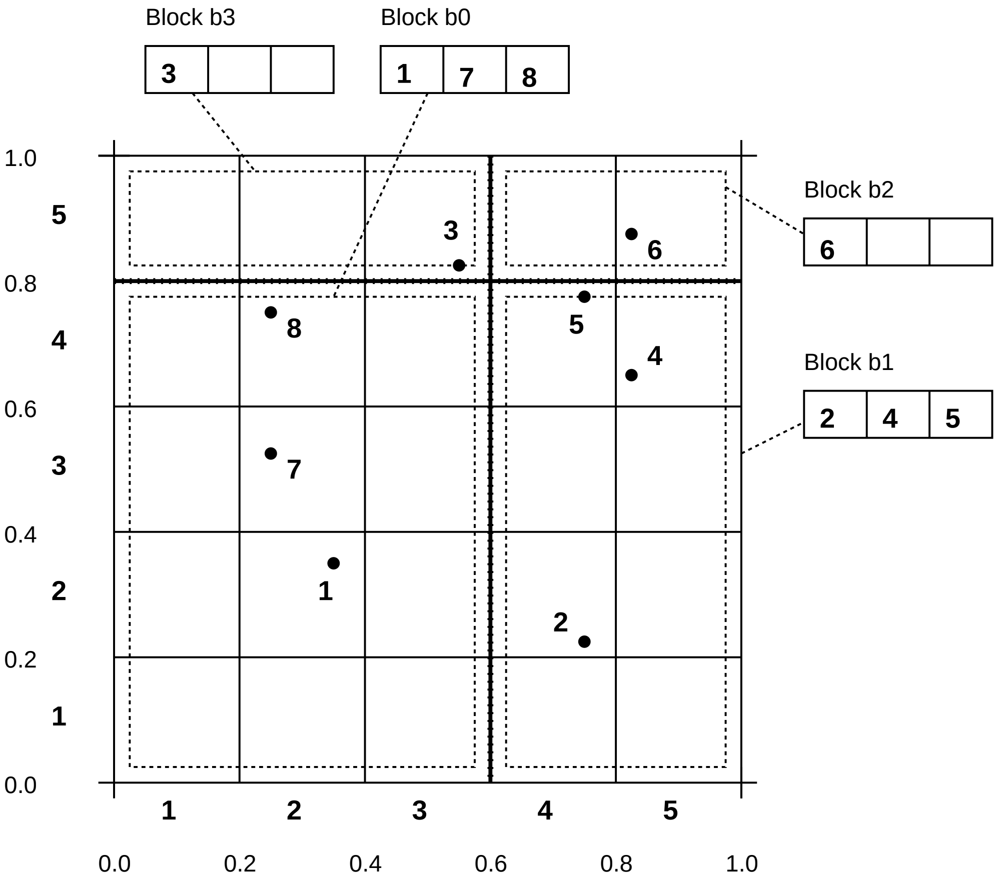
- Insert $p_6 = (0.82, 0.87)$:
  - Locate cell: $\left\lceil \frac{0.82}{0.2} \right\rceil = 5$, $\left\lceil \frac{0.87}{0.2} \right\rceil = 5$
    $\Rightarrow$ Cell[5,5].
  - Cell points to block b_1
    $\Rightarrow$ retrieve block b_1.
  - Block b_1 is full
    $\Rightarrow$ split needed.
  - No existing partition does the job
    $\Rightarrow$ create new partition.
  - *Current* = horizontal
    $\Rightarrow$ make a horizontal partition.
  - *Current* := vertical.



- Insert $p_7 = (0.25, 0.53)$:
  - Locate cell: $\left\lceil \frac{0.25}{0.2} \right\rceil = 2$, $\left\lceil \frac{0.53}{0.2} \right\rceil = 3$
    $\Rightarrow$ Cell[2,3].
  - Cell points to block b_0
    $\Rightarrow$ retrieve block b_0.
  - Space available
    $\Rightarrow$ insert point.

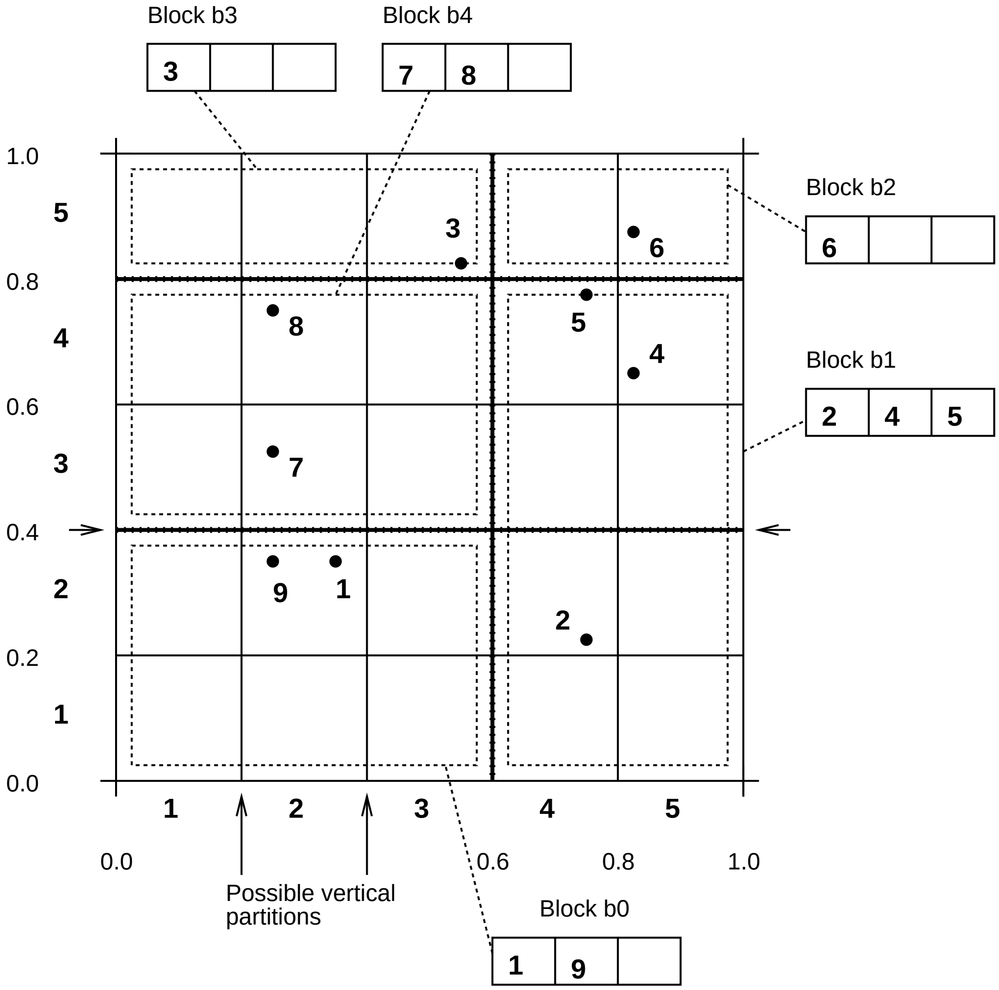


- Insert $p_8 = (0.24, 0.77)$:
  - Locate cell: $\lceil\frac{0.24}{0.2}\rceil = 2$, $\lceil\frac{0.77}{0.2}\rceil = 4$
    $\Rightarrow$ Cell[2,4].
  - Cell points to block b_0
    $\Rightarrow$ retrieve block b_0.
  - Block b_0 is full
    $\Rightarrow$ split needed.
  - *Current* = vertical
    $\Rightarrow$ No existing vertical partition is sufficient.
  - However, the horizontal partition at 0.8 works
    $\Rightarrow$ use partition to split block b_0.
  - *Current* := vertical.

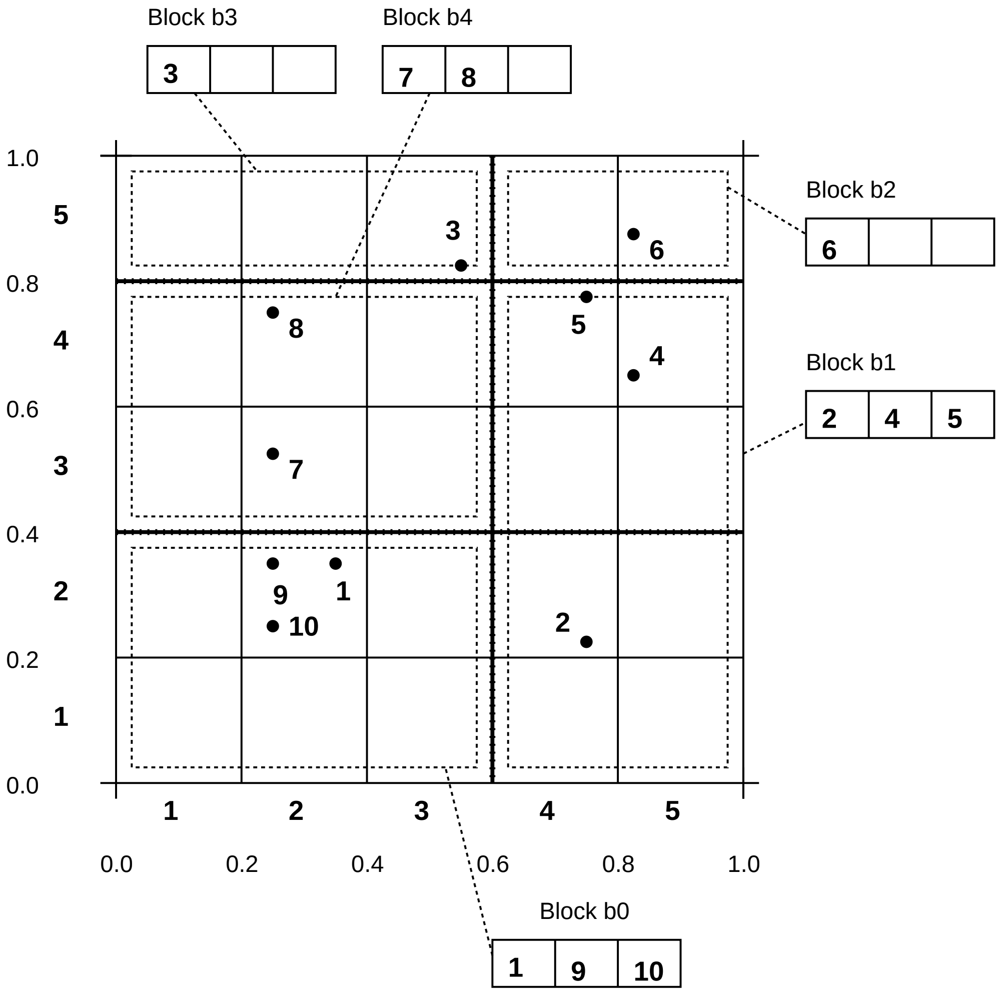


- Insert $p_9 = (0.23, 0.35)$:
  - Locate cell: $\left\lceil\frac{0.23}{0.2}\right\rceil = 2$, $\left\lceil\frac{0.35}{0.2}\right\rceil = 2$
    ⇒ Cell[2,2].
  - Cell points to block b_0
    ⇒ retrieve block b_0.
  - Block b_0 is full
    ⇒ split needed.
  - *Current* = vertical.
  - No existing vertical partition is sufficient.
  - No existing horizontal partition is sufficient.
  - No new vertical partition (either at 0.2 or 0.4) is sufficient
    ⇒ try new horizontal partition.

- Horizontal partition at 0.4.
- *Current* := vertical.

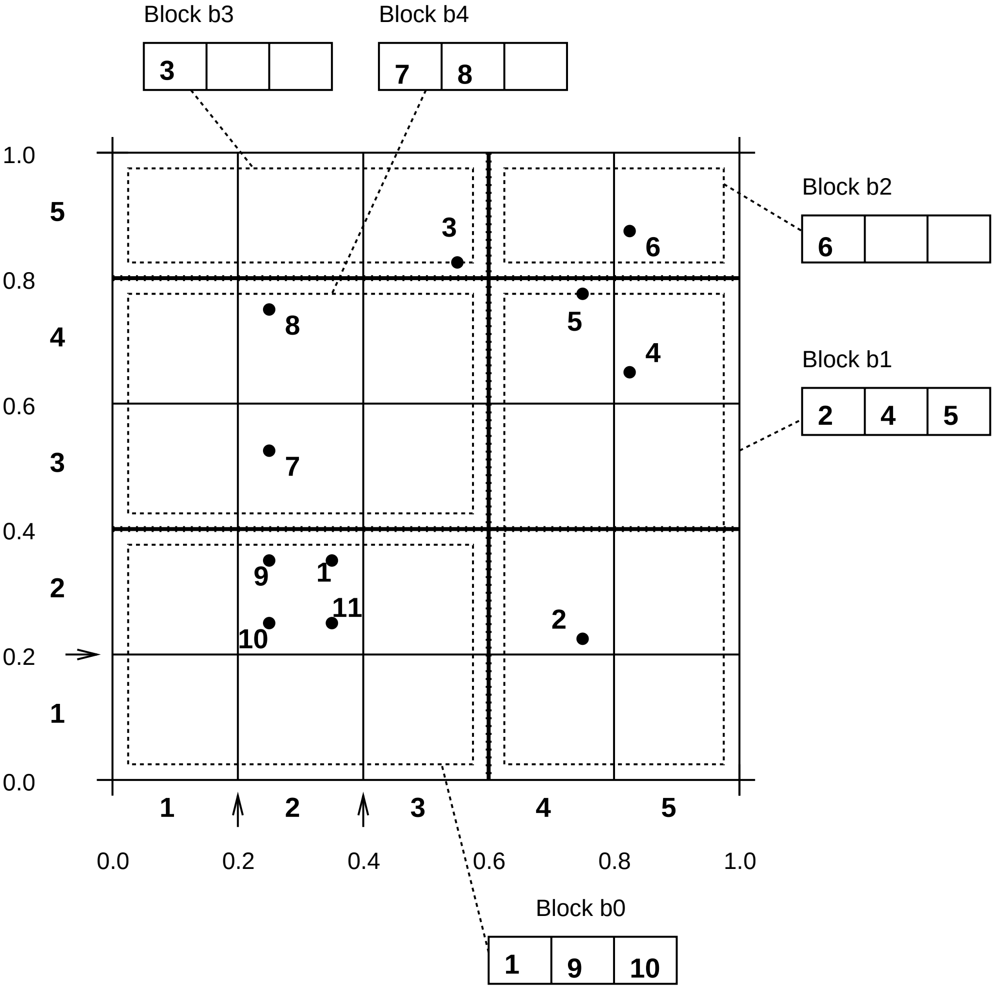


- Insert $p_{10} = (0.25, 0.24)$:
  - Locate cell: $\lceil \frac{0.25}{0.2} \rceil = 2$, $\lceil \frac{0.24}{0.2} \rceil = 2$
    ⇒ Cell[2,2].
  - Cell points to block b_0
    ⇒ retrieve block b_0.
  - Space available
    ⇒ insert point.



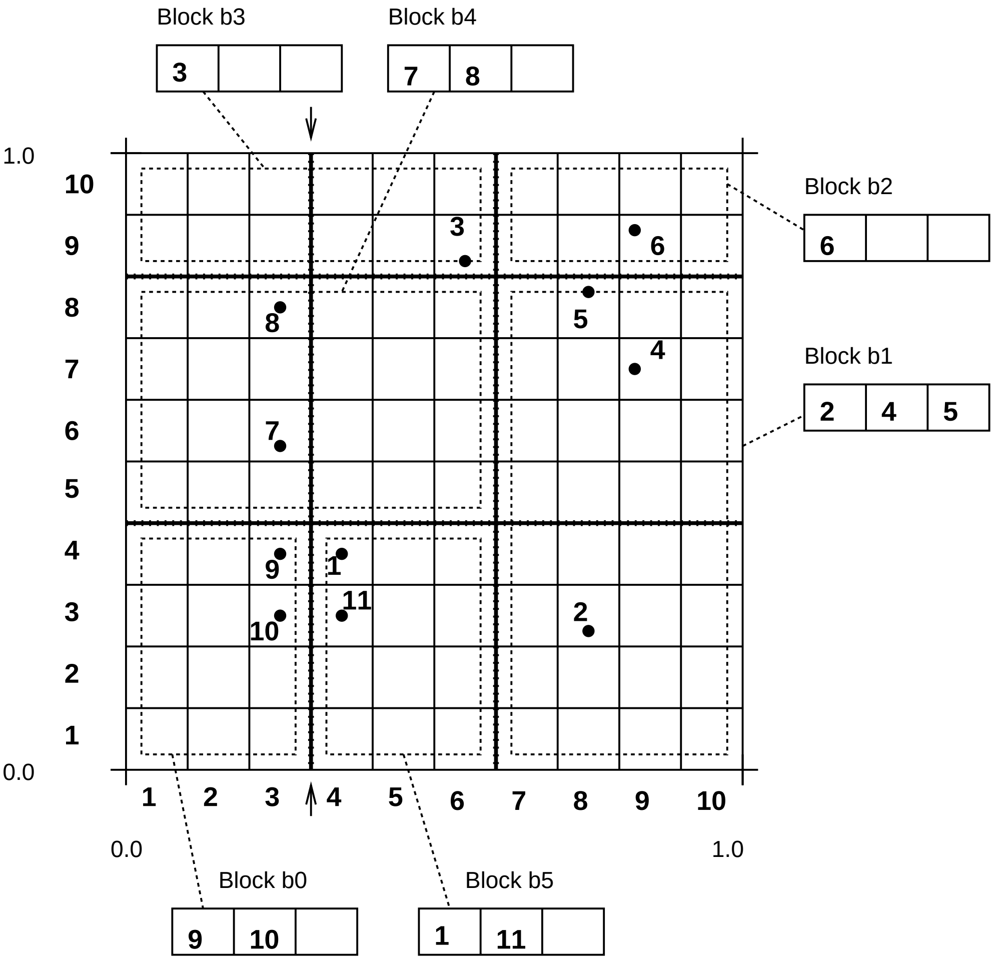
- Insert $p_{11} = (0.31, 0.24)$:
  - Locate cell: $\left\lceil \frac{0.31}{0.2} \right\rceil = 2$, $\left\lceil \frac{0.24}{0.2} \right\rceil = 2$
    $\Rightarrow$ Cell[2,2].
  - Cell points to block b_0
    $\Rightarrow$ retrieve block b_0.
  - Block b_0 is full
    $\Rightarrow$ split needed.
  - *Current* = vertical
    $\Rightarrow$ No existing vertical partition is sufficient.

- No existing horizontal partition is sufficient.
- Try vertical partition
  ⇒ partitions at 0.2 and 0.4 are not sufficient.
- Try horizontal partition
  ⇒ partitions at 0.2 and 0.4 are not sufficient.



- Need to refine grid
  ⇒ double grid size
- *x-interval* := 0.1; *y-interval* := 0.1.
- *Current* = vertical
  ⇒ try vertical cut (at 0.3).

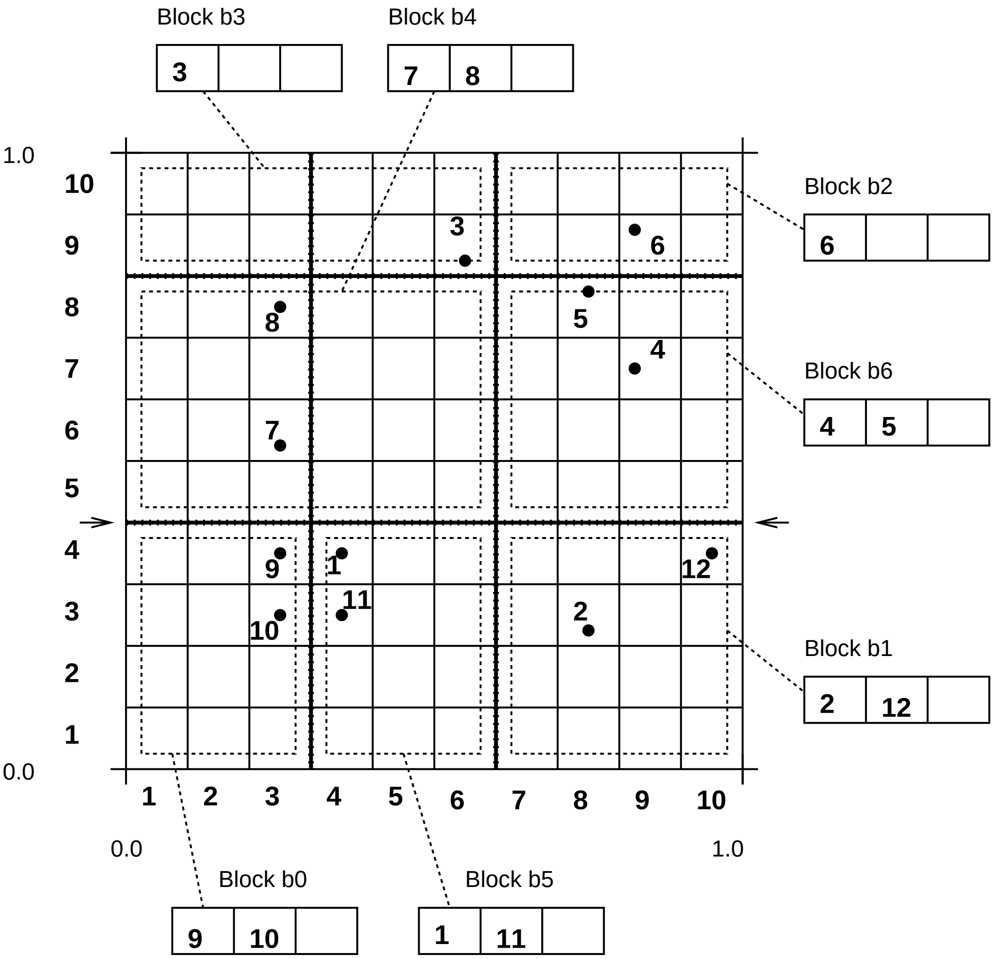
– *Current* := horizontal.



- Insert $p_{12} = (0.94, 0.35)$:
  - Locate cell: (x-*interval*=y-*interval*=0.1) $\lceil\frac{0.94}{0.1}\rceil = 10$, $\lceil\frac{0.35}{0.1}\rceil = 4$ ⇒ Cell[10,4].
  - Cell points to block b_1 ⇒ retrieve block b_1.
  - Block b_1 is full ⇒ split needed.
  - *Current* = horizontal ⇒ try existing horizontal partitions.

– Horizontal partition at 0.4 is sufficient.

– *Current* := vertical.

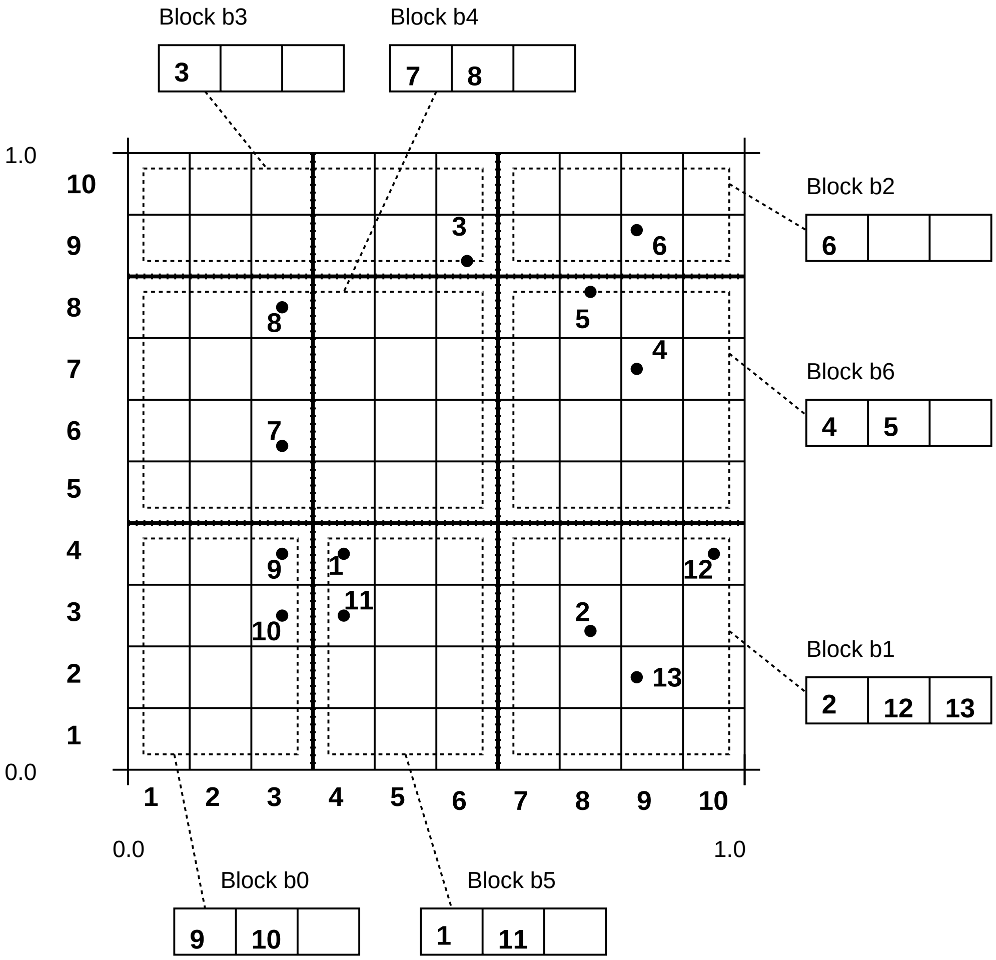


- Insert $p_{13} = (0.82, 0.15)$:

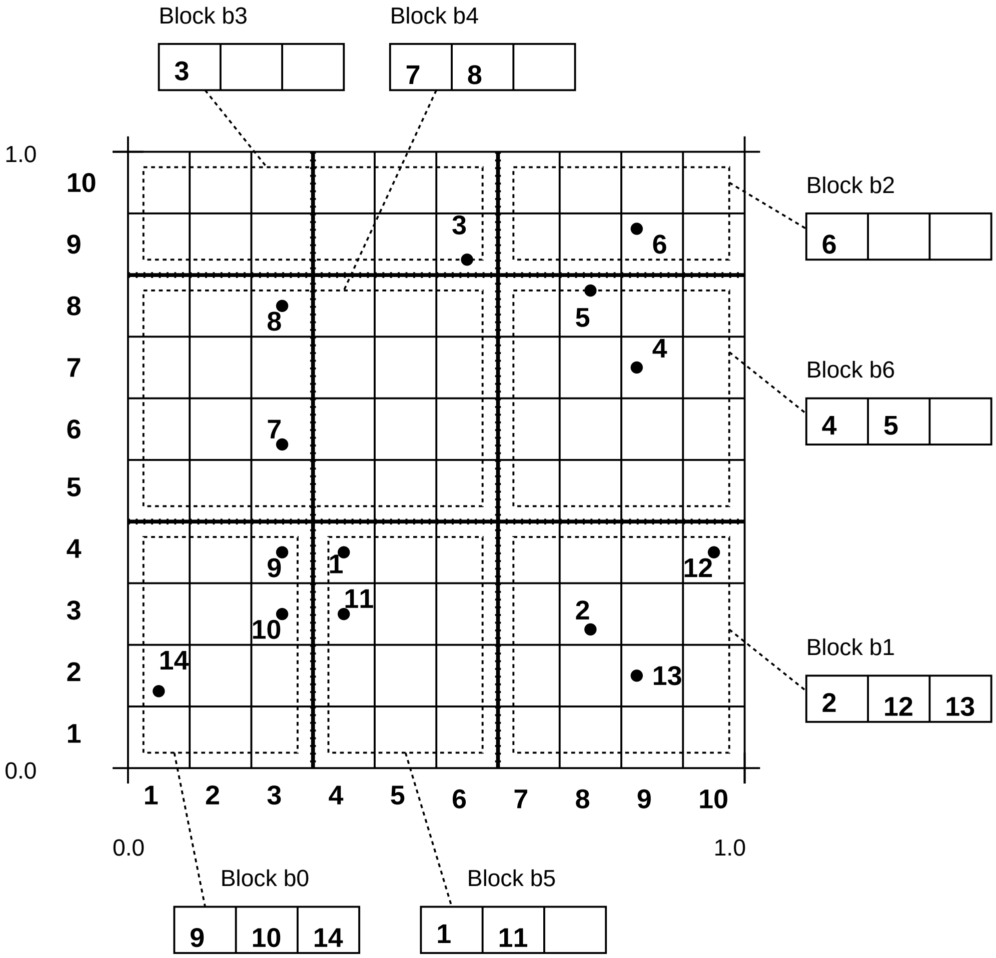
  – Locate cell: $\lceil \frac{0.82}{0.1} \rceil = 9$, $\lceil \frac{0.15}{0.1} \rceil = 2$
  $\Rightarrow$ Cell[9,2].

  – Cell points to block b_1
  $\Rightarrow$ retrieve block b_1.

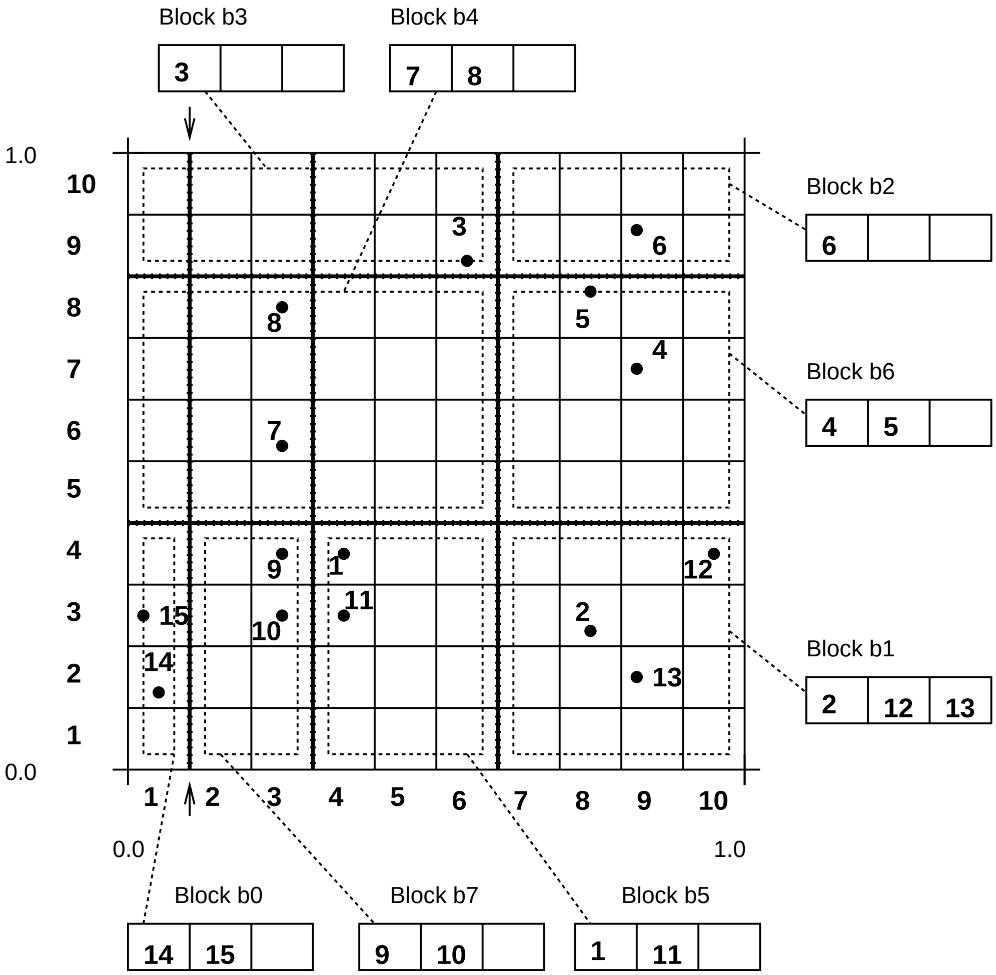
  – Space available
  $\Rightarrow$ insert point.



- Insert $p_{14} = (0.05, 0.12)$:
  - Locate cell: $\lceil \frac{0.05}{0.1} \rceil = 1$, $\lceil \frac{0.12}{0.1} \rceil = 2$
    $\Rightarrow$ Cell[1,2].
  - Cell points to block b_0
    $\Rightarrow$ retrieve block b_0.
  - Space available
    $\Rightarrow$ insert point.



- Insert $p_{15} = (0.03, 0.24)$:
  - Locate cell: $\lceil\frac{0.03}{0.1}\rceil = 1$, $\lceil\frac{0.24}{0.1}\rceil = 3$
    $\Rightarrow$ Cell[1,3].
  - Cell points to block b_0
    $\Rightarrow$ retrieve block b_0.
  - Block b_0 is full
    $\Rightarrow$ split needed.
  - No existing vertical or horizontal partition is sufficient.
  - *Current* = vertical
    $\Rightarrow$ partition at 0.1.
  - *Current* := horizontal.



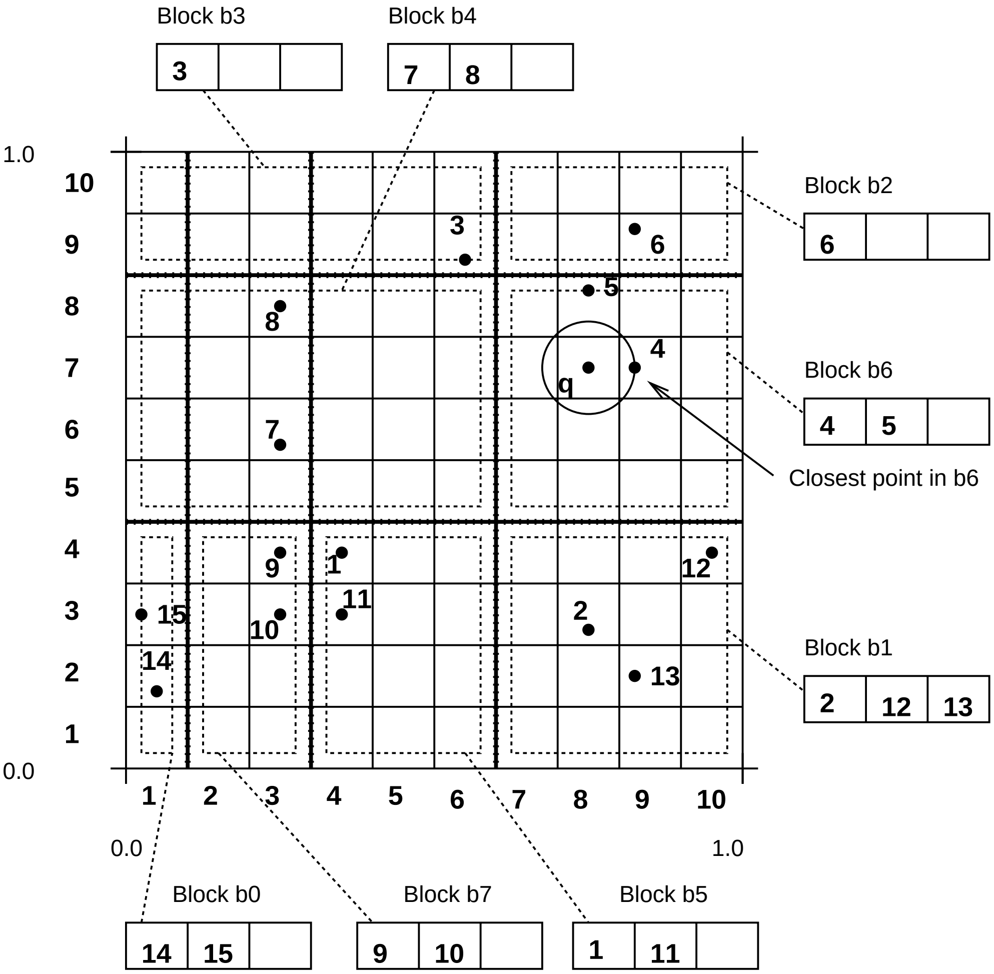
- To summarize:
  - Locate cell for given point and retrieve corresponding data block.
  - If block has space, insert point.
  - Otherwise use splitting rules to split block:
    - ∗ If an existing partition can be used, use it.
    - ∗ Otherwise, create a new partition.
    - ∗ If creating a partition doesn't work, a grid refinement is needed.

11.11 Grid Files: Nearest Neighbor Query

- Given a query point $q = (x_0, y_0)$ we wish to find the closest point to q in the grid file.
- Key ideas:
  - Obtain the cell containing q.
  - Get the data block pointed to by the cell.
  - Find the closest point among points in the block.
  - Suppose d is the distance to this point.
  - Consider a circle with radius d around q.
  - Retrieve all cells that lie in or on the circle.
  - Search corresponding data blocks.
- Example: $q = (0.75, 0.65)$
  - Locate cell: $\left\lceil \frac{0.75}{0.1} \right\rceil = 8$, $\left\lceil \frac{0.65}{0.1} \right\rceil = 7$
    $\Rightarrow$ Cell[8,7].
  - Cell points to block b_6
    $\Rightarrow$ retrieve block b_6.
  - Block b_6 has points p_4 and p_5.
  - Point $p_4 = (0.82, 0.66)$ is closest.
  - $d = dist(q, p_4) = \left((0.75 - 0.82)^2 + (0.65 - 0.66)^2\right)^{1/2} = 0.07.$
  - A circle about q with radius 0.07 covers cells [7,6], [8,6], [9,6], [7,7], [8,7], [9,7], [7,8], [8,8], [9,8].
  - For each of these cells:
    - $*$ Retrieve block pointed to by cell:
      $\Rightarrow$ in the example, block b_6.

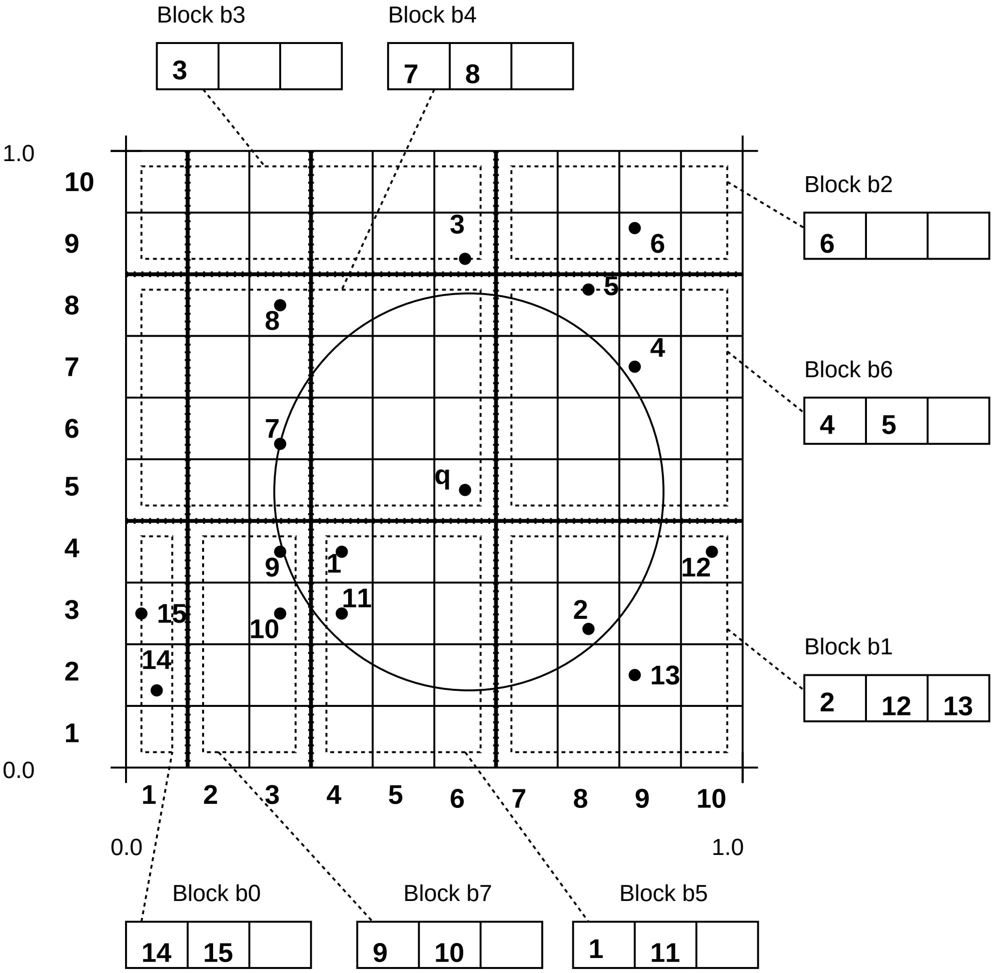
- $*$ Compute distance to points in block:
  $\Rightarrow$ in the example, no need to compute.

- – Pick closest point
  $\Rightarrow p_4 = (0.82, 0.66)$.

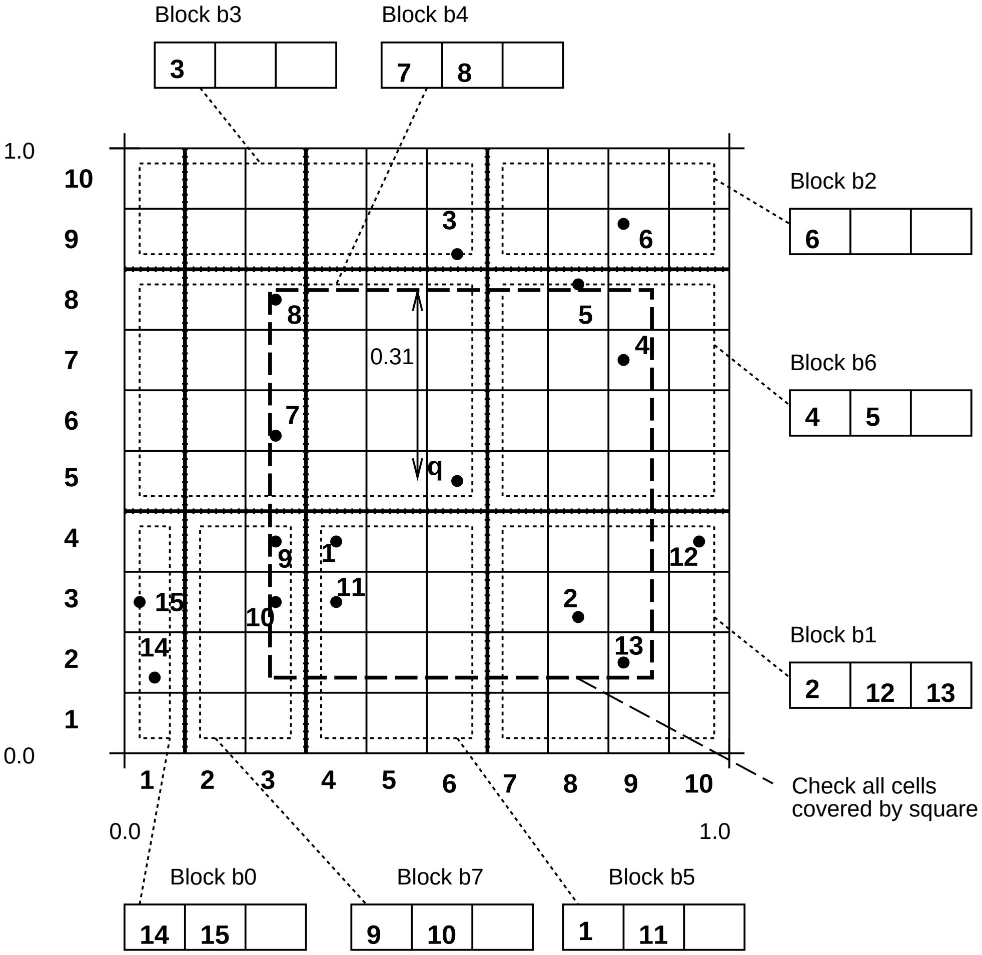


- Example: $q = (0.55, 0.45)$
  - – Locate cell: $\lceil \frac{0.55}{0.1} \rceil = 6$, $\lceil \frac{0.45}{0.1} \rceil = 5$
    $\Rightarrow$ Cell[6,5].
  - – Cell points to block b_4
    $\Rightarrow$ retrieve block b_4.

- Block b_4 has points p_7 and p_8.
- Point $p_7 = (0.25, 0.53)$ is closest.
- $d = dist(q, p_7) = \big((0.55 - 0.25)^2 + (0.45 - 0.53)^2\big)^{1/2} = 0.31$.
- A circle about q with radius 0.31 covers cells that point to blocks b_1, b_4, b_5, b_6 and b_7.
- For each of these cells:
  - Retrieve block pointed to by cell.
  - Compute distance to points in block:
- Pick closest point
  $\Rightarrow p_1 = (0.35, 0.37)$.



- Note: instead of using a circle to determine "must-see" cells, a square can be used:
  - Consider the above example.
  - $d = dist(q, p_7) = 0.31$.
  - Construct the square that circumscribes the circle with radius 0.31 about q:



  - Check cells in the square.
  - Advantage: easier to compute cells covered.
  - Disadvantage: covers more cells.
- Range queries:

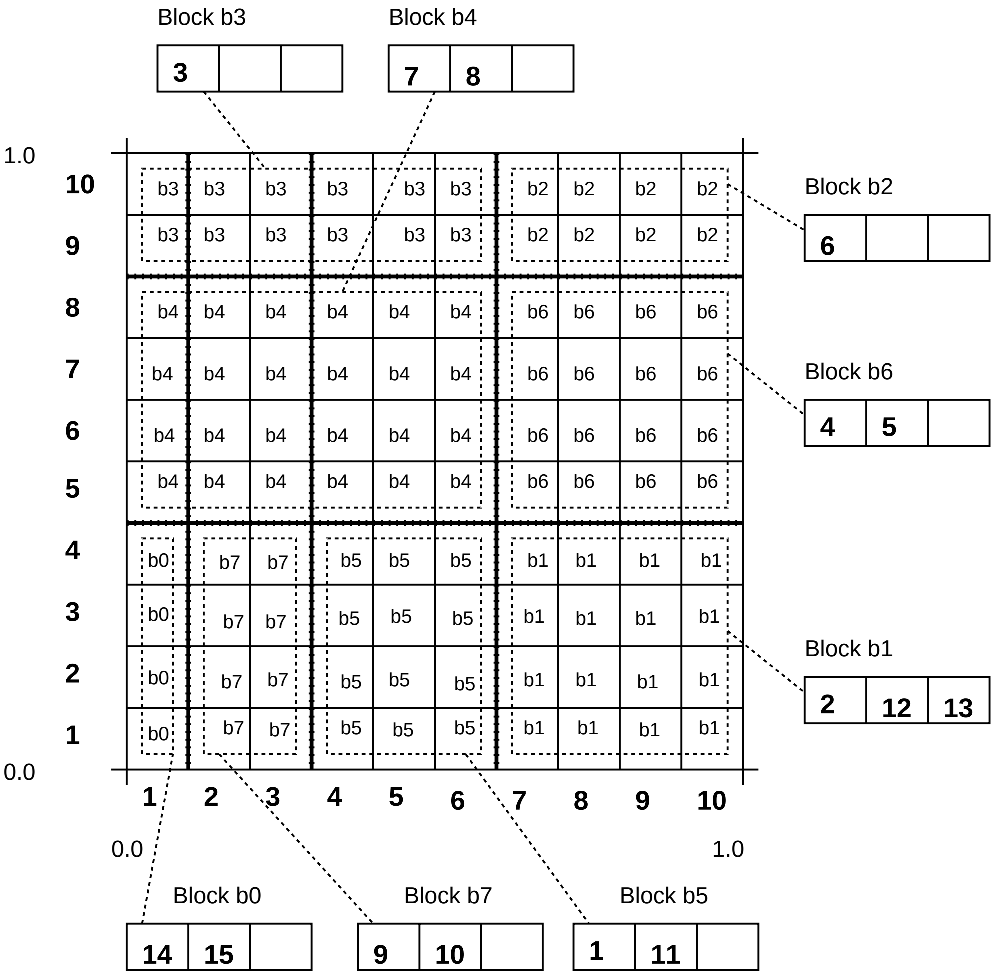
Range queries can implemented by finding all the cells in the given range.

Example: find all point in the range [0.0,0.3].

- The range corresponds to all cells with x-coordinates in the range [1,3]
  ⇒ columns 1, 2 and 3.
- Next, identify all blocks in these columns
  ⇒ b_0, b_3, b_4 and b_7.
- Retrieve points in these blocks:
  ⇒ points 1,3,7,8,9,10,11,14,15.
- Identify which points lie in the desired range:
  ⇒ points 7,8,9,10,14,15.

11.12 Grid Files: Some Implementation Details

- Details: the grid directory
  - What really is a grid directory?
    - ∗ Basically, a collection of grid cells.
    - ∗ Each cell has a block pointer.
  - Thus, in the previous example:

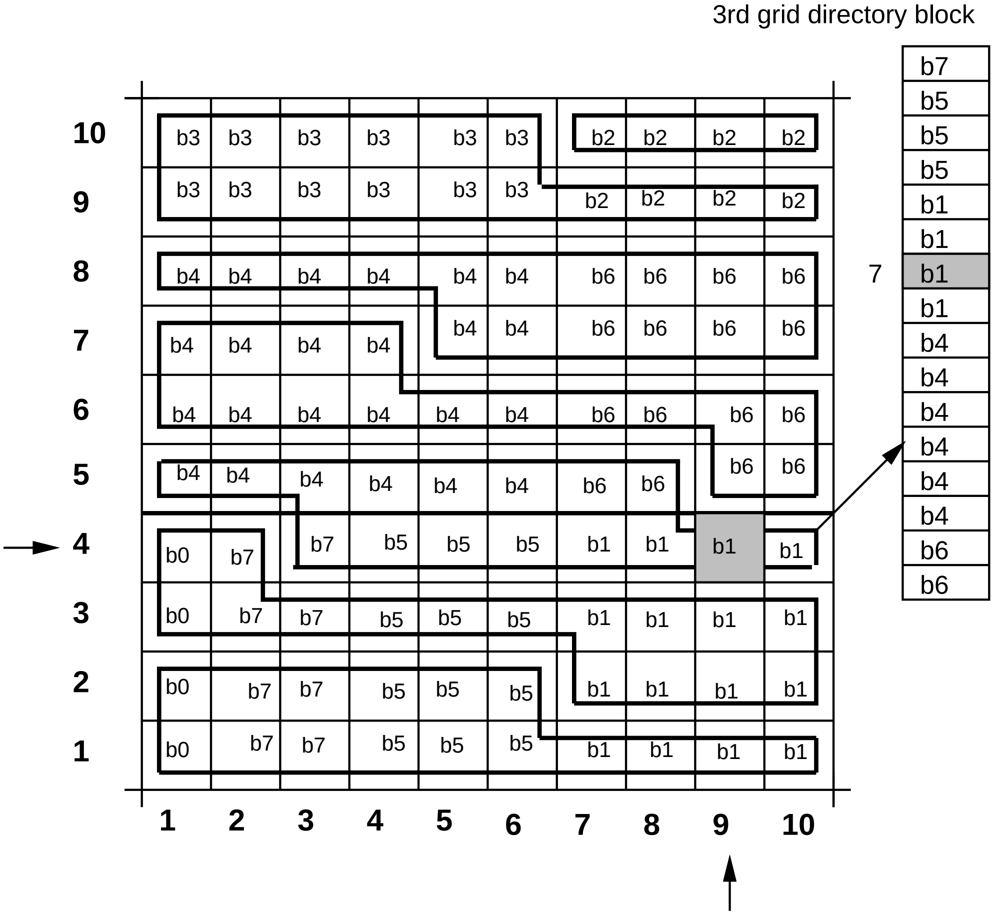


	1	2	3	4	5	6	7	8	9	10
10	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
9	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
8	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
7	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
6	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
5	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
4	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
3	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
2	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
1	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1

Note that each cell has a block pointer to the appropriate block.

– How is the directory stored on disk?

- ∗ Cells have addresses, e.g., Cell[8,3].
- ∗ We need to retrieve a cell given its address.
- ∗ Suppose blocksize = 64 bytes.
- ∗ Suppose a block pointer is 4 bytes
  ⇒ a cell needs 4 bytes
  ⇒ 16 cells per block.
- ∗ If we stored the cells in row-major order:



3rd grid directory block

10	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
9	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
8	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
7	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
6	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
5	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
4	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
3	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
2	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
1	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
	1	2	3	4	5	6	7	8	9	10

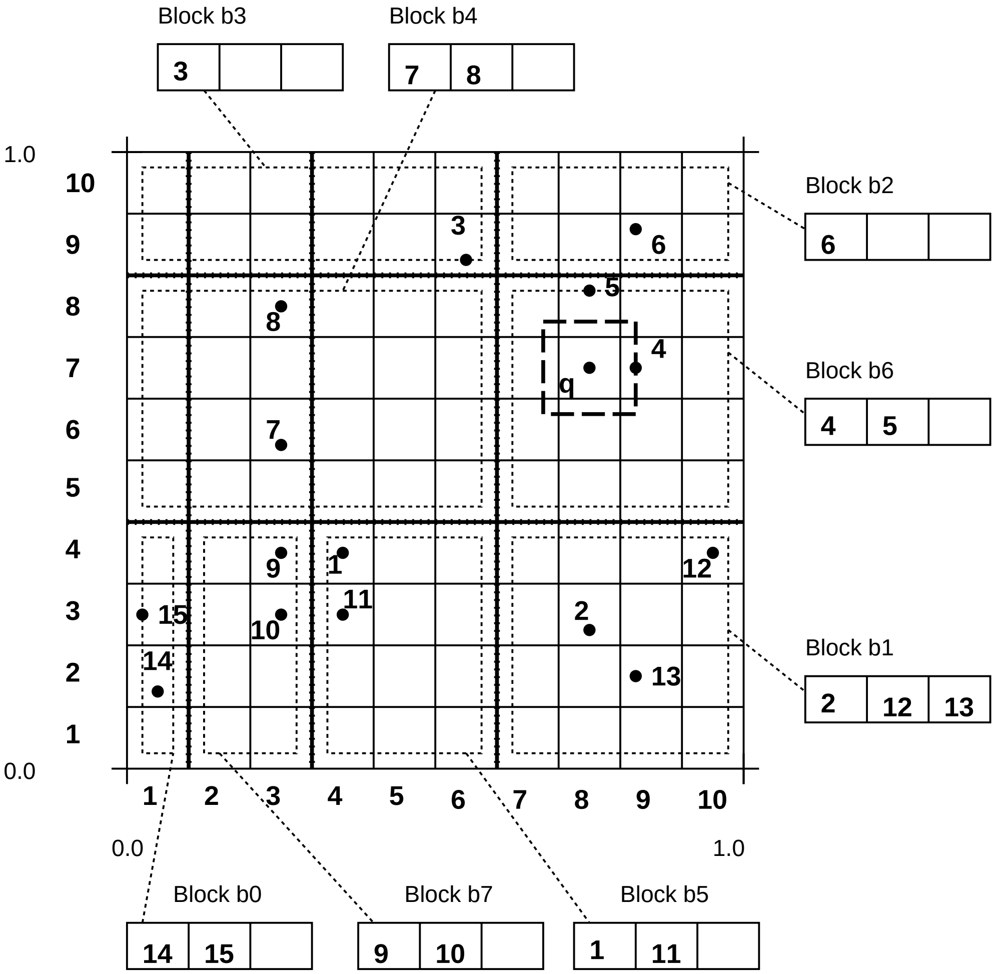
	b7
	b5
	b5
	b5
	b1
	b1
7	b1
	b1
	b4
	b4
	b4
	b4
	b4
	b4
	b6
	b6

- ∗ Example: To obtain Cell[9,4]:
  - · Compute cell number in linear order: $3 \times 10 + 9 = 39$.
  - · Compute which block: $\left\lceil \frac{39}{16} \right\rceil = $ 3rd block.
  - · 7th item in 3rd block

- Note: we need to store the positions of the partition lines
  $\Rightarrow$ can be stored in an array or linked list.

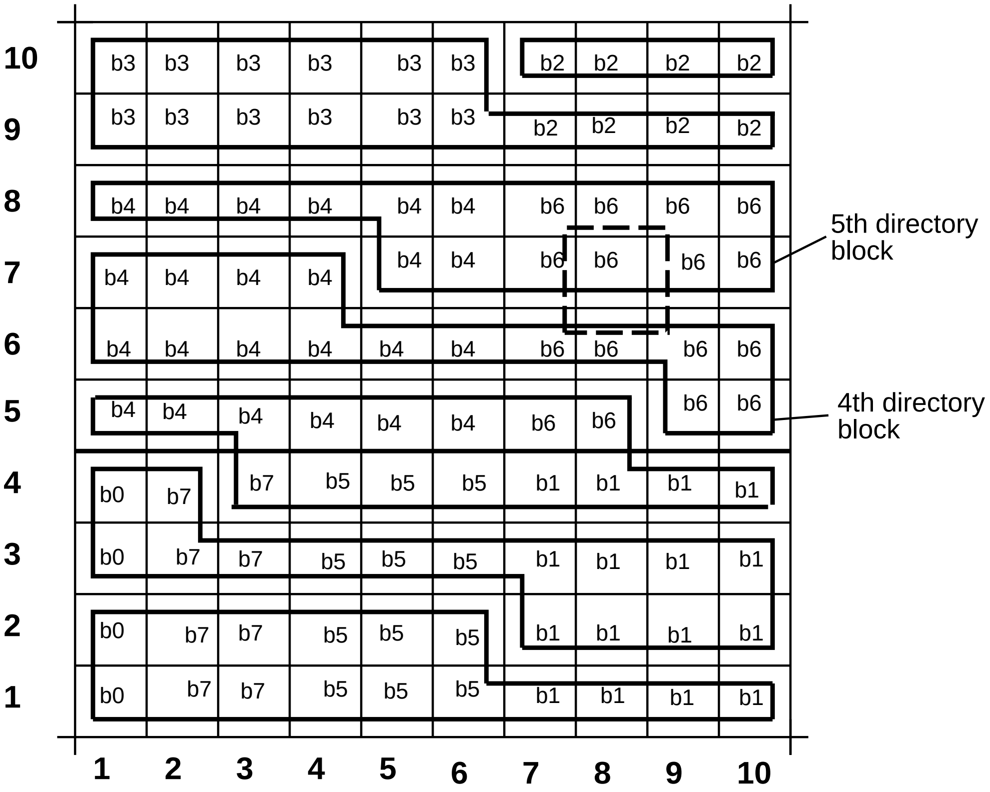
- Details: nearest neighbor query

  - Example: consider the query $q_1 = (0.75, 0.65)$.
  - First, locate cell: $\left\lceil \frac{0.75}{0.1} \right\rceil = 8$, $\left\lceil \frac{0.65}{0.1} \right\rceil = 7$
    $\Rightarrow$ Cell[8,7].



  - Retrieve grid block containing cell
    $\Rightarrow$ 7th grid directory block.
  - Retrieve cell contents
    $\Rightarrow$ pointer to block b_6.

- Retrieve block b_6 and check distance to each point
  $\Rightarrow$ point p_4 is closest.
- $d = dist(q, p_4) = \left((0.75 - 0.82)^2 + (0.65 - 0.66)^2\right)^{1/2} = 0.07.$
- Consider a square with q as center and half-side= 0.07.
- Cells covered: [7,6], [8,6], [9,6], [7,7], [8,7], [9,7], [7,8], [8,8], [9,8].
- Search cells covered.
- Note: not all cells need to be searched, since some point to the same blocks:



10	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
9	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
8	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
7	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
6	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
5	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
4	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
3	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
2	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
1	b0	b7	b7	b5	b5	b5	b1	b1	b1	b1
	1	2	3	4	5	6	7	8	9	10

- The square covers only the 4th and 5th directory blocks
  $\Rightarrow$ first consider all the square-cells in the 4th directory block
  $\Rightarrow$ fewer disk accesses.

11.13 Grid Files: Variable-Size Cells

- The fixed-size cell version has a few drawbacks:
  - Highly non-uniform point distributions can cause multiple grid refinements
    ⇒ the grid directory can occupy a lot of space.
  - A lot of cells in the grid directory point to the same block
    ⇒ unnecessary redundancy.
  - A grid refinement results in a lot of copying
    ⇒ time-consuming step.
- In the variable-size version:
  - The grid directory grows more rapidly, but does not need refinement.
  - Fewer cells point to the same block.
  - No grid refinement is needed. Instead, the grid directory only grows one column and row for each partition.
- Consider the same example as before:
  - Insert the following point data:

p_1	(0.35,0.37)	p_6	(0.82,0.87)	p_{11}	(0.31,0.24)
p_2	(0.74,0.23)	p_7	(0.25,0.53)	p_{12}	(0.94,0.35)
p_3	(0.55,0.81)	p_8	(0.24,0.77)	p_{13}	(0.82,0.15)
p_4	(0.82,0.66)	p_9	(0.23,0.35)	p_{14}	(0.05,0.12)
p_5	(0.75,0.78)	p_{10}	(0.25,0.24)	p_{15}	(0.03,0.24)

  - We will assume all the point data falls in interval [0,1]
    ⇒ i.e., assume MBR of the set of points is known.
  - Assume blocksize = 64 bytes.
  - Suppose we use

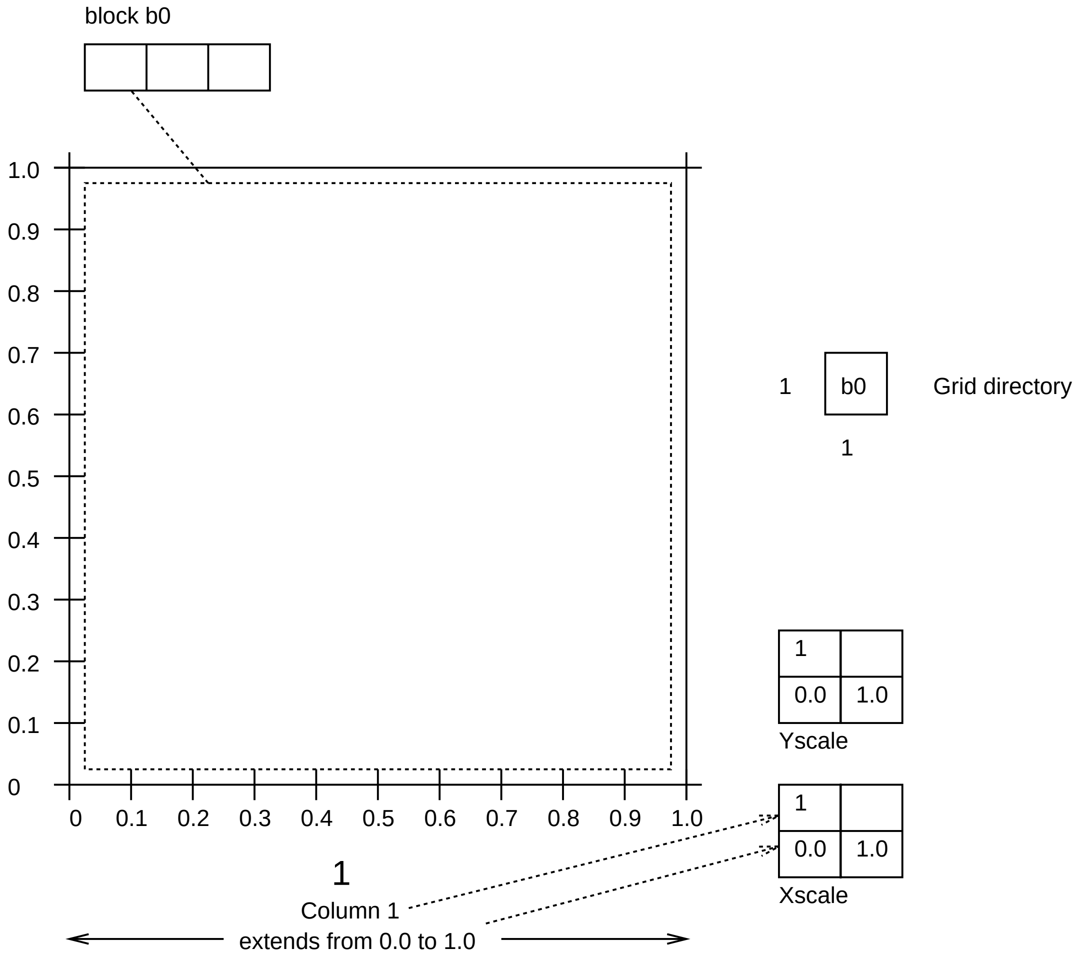
4 bytes	for x-coordinate (float)
4 bytes	for y-coordinate (float)
2 bytes	for a point ID (integer)
10 bytes	for a text label (character string)

⇒ totally 20 bytes
⇒ 3 point-tuples per block.

- Key ideas in inserting a point $p = (x_0, y_0)$:
  - Locate the cell that contains the point:
    - Use the *Xscale* to locate the x-interval containing the point.
    - Use the *Yscale* to locate the y-interval containing the point.
    - Then, retrieve the corresponding cell.
    - Next, retrieve the block pointed to by the cell.
    - If space available, insert point in block.
    - Otherwise, the block is split:
      - Try to use an existing partition.
      - If no existing partition works, create a new one.
      - Insert point in appropriate block.
      - If a vertical partition was created, update *Xscale*.
      - If a horizonal partition was created, update *Yscale*.
    - Try to alternate between vertical and horizontal partitions.
      - Use a variable *Current* to indicate which direction.
      - If a vertical cut (partition) is made (or used):
⇒ set *Current* := horizontal.
      - Initially, say, *Current* := vertical.
- The example will emphasize concepts; we will consider implementation details later.

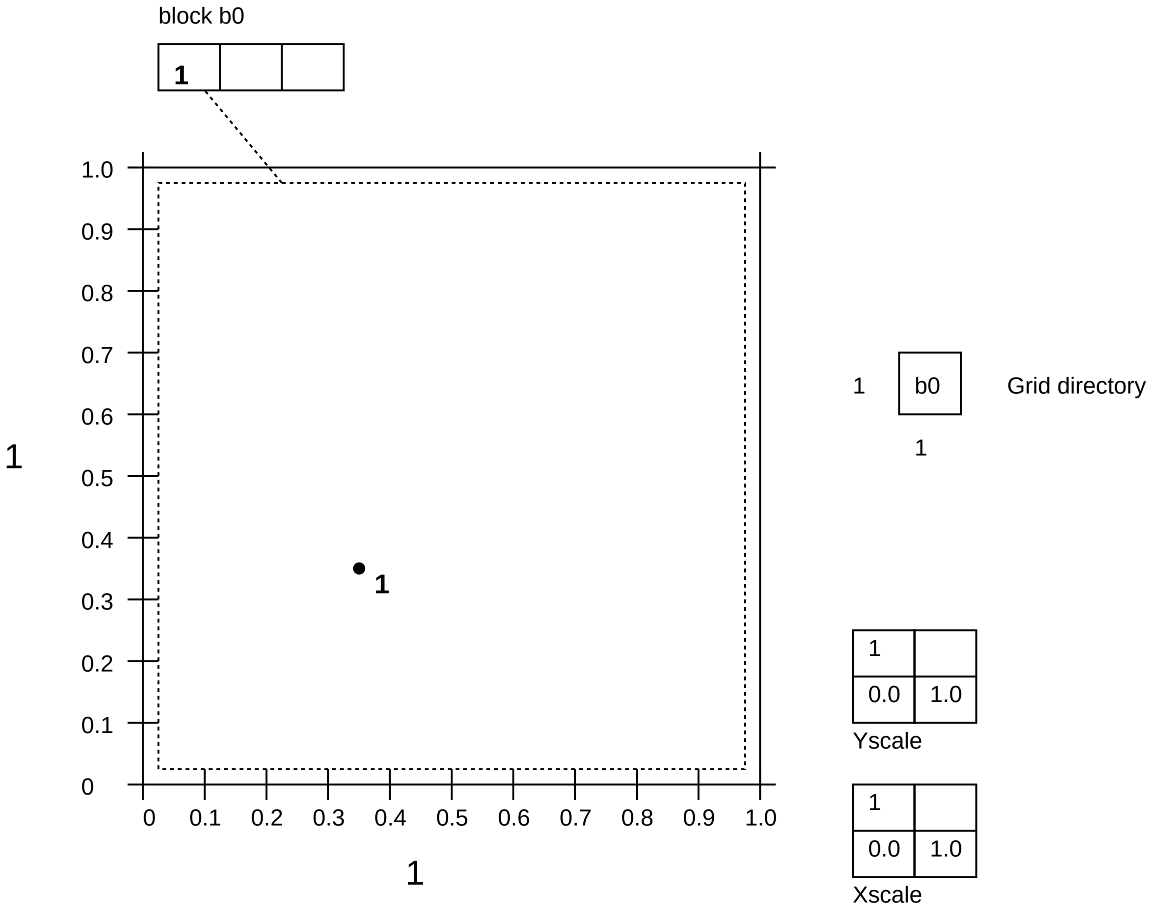
  Note: in the fixed-size cell example, each grid-picture stood for both the grid directory and the data description. In the variable-size version, the directory and data description are drawn separately.
- Initially:

- – A sufficiently large contiguous area on disk is allocated for the grid directory (to allow it to grow).
- – Initially, there is only one cell, pointing to an empty block.
- – *Current* := vertical.



- Insert $p_1 = (0.35, 0.37)$:
  - – Locate cell:
    - ∗ Take the x-value, 0.35, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to column 1.

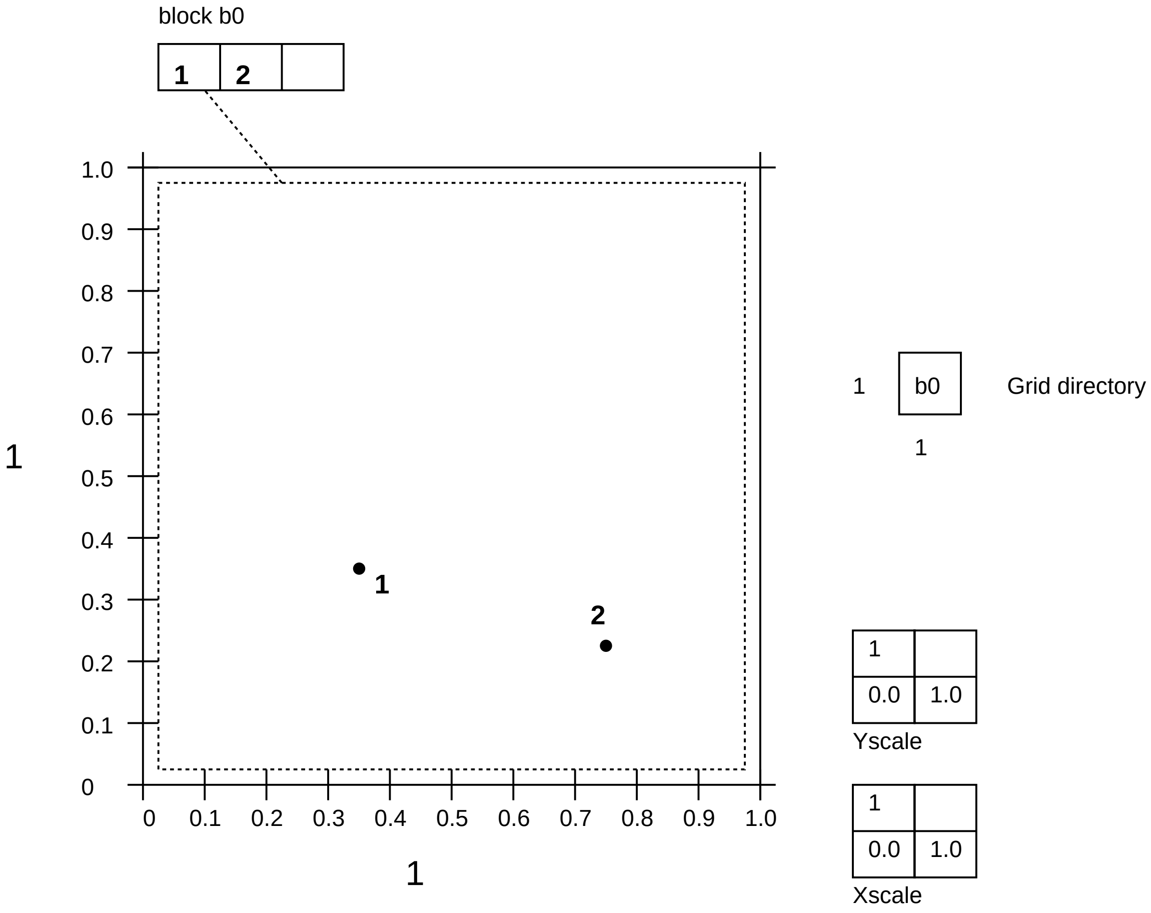
    - * Take the y-value, 0.37, and use the *Yscale* to see which interval it lies in
        ⇒ lies in the interval [0.0,1.0]
        ⇒ corresponds to row 1.
  - Retrieve Cell[1,1] of the grid directory.
  - Retrieve the block it points to
    ⇒ block b_0.
  - Block b_0 has space available
    ⇒ insert point.



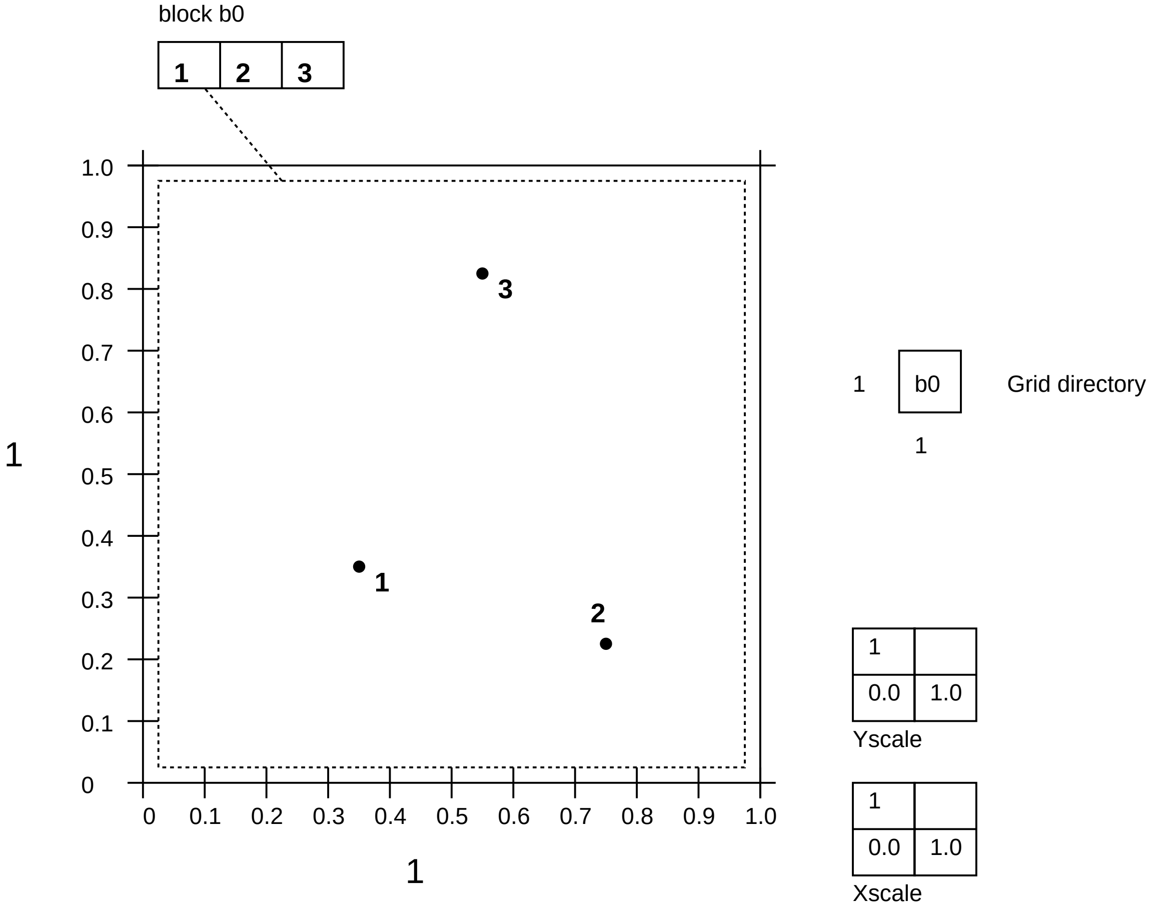
- Insert $p_2 = (0.74, 0.23)$:
  - Locate cell:

    - ∗ Take the x-value, 0.74, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to column 1.
    - ∗ Take the y-value, 0.23, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to row 1.

- – Retrieve Cell[1,1] of the grid directory.
- – Retrieve the block it points to
  ⇒ block b_0.
- – Block b_0 has space available
  ⇒ insert point.

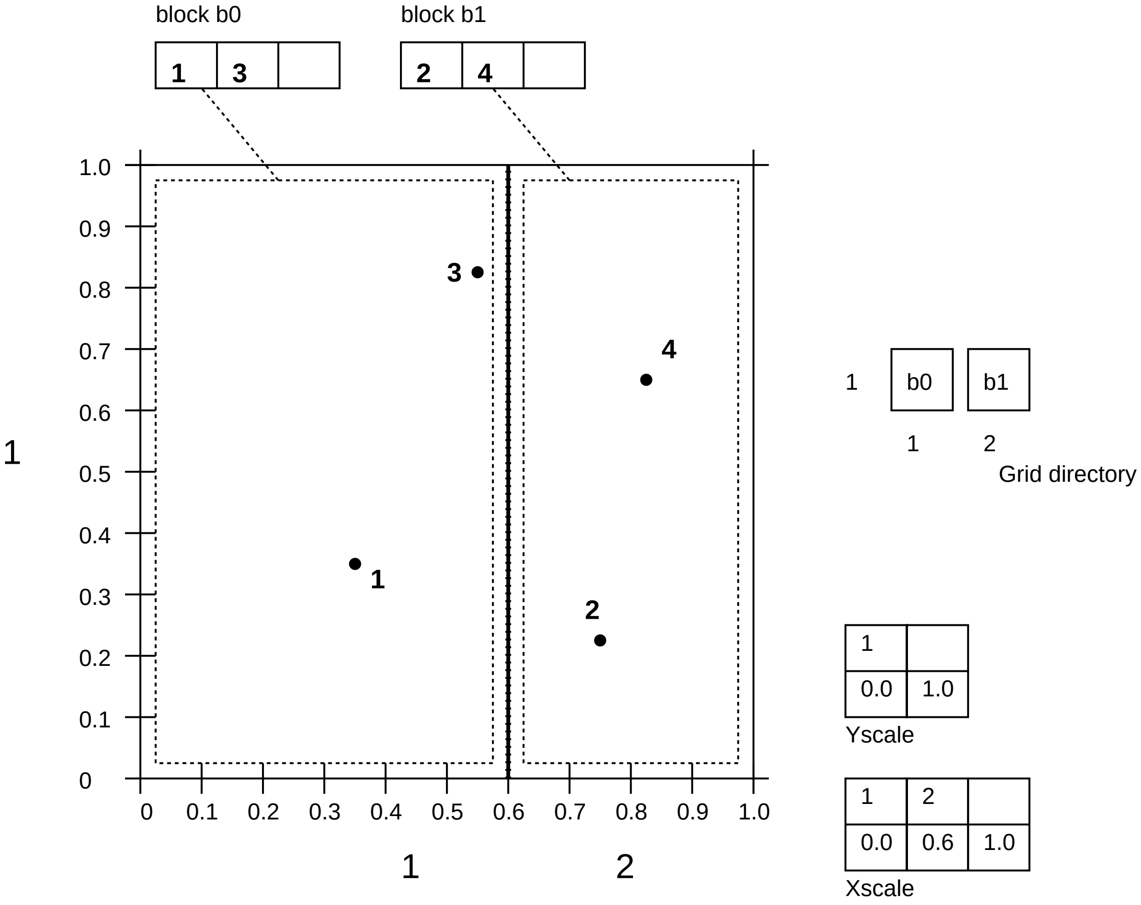

block b0
1
2
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1
1
2
1
b0
Grid directory
1
1
0.0
1.0
Yscale
1
0.0
1.0
Xscale

- Insert $p_3 = (0.55, 0.81)$:
  - Locate cell:
    - Take the x-value, 0.55, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to column 1.
    - Take the y-value, 0.81, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to row 1.
  - Retrieve Cell[1,1] of the grid directory.
  - Retrieve the block it points to
    ⇒ block b_0.
  - Block b_0 has space available
    ⇒ insert point.



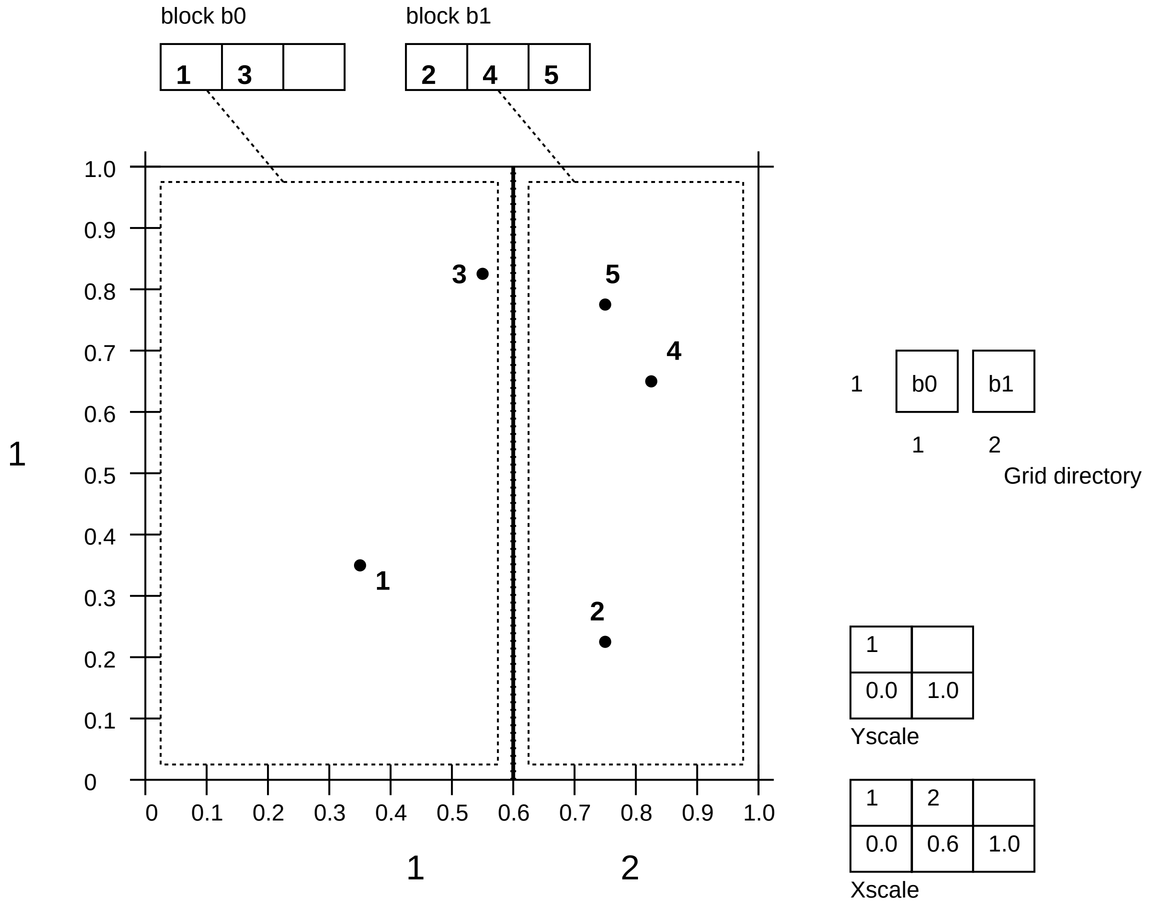
- Insert $p_4 = (0.82, 0.66)$:
  - Locate cell:
    - ∗ Take the x-value, 0.82, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to column 1.
    - ∗ Take the y-value, 0.66, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to row 1.
  - Retrieve Cell[1,1] of the grid directory.
  - Retrieve the block it points to
    ⇒ block b_0.

- Block b_0 is full
  $\Rightarrow$ split block.
- *Current* = vertical
  $\Rightarrow$ use vertical partition so that half the points are in a block.
- For example, partition at 0.6.
- Add new column to grid directory.
- Update *Xscale* to contain new interval (with correct column number).
- *Current* := horizontal.



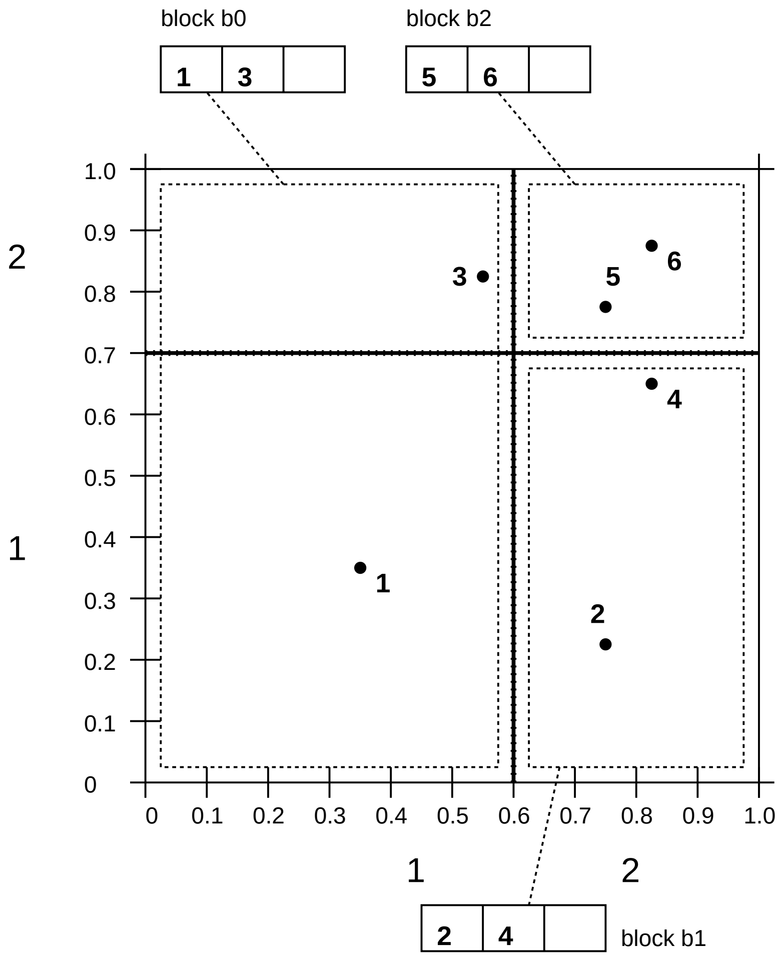
- Insert $p_5 = (0.75, 0.78)$:

- Locate cell:
  - Take the x-value, 0.75, and use the *Xscale* to see which interval it lies in
    ⇒ lies in the interval [0.6,1.0]
    ⇒ corresponds to column 2.
  - Take the y-value, 0.78, and use the *Yscale* to see which interval it lies in
    ⇒ lies in the interval [0.0,1.0]
    ⇒ corresponds to row 1.
- Retrieve Cell[2,1] of the grid directory.
- Retrieve the block it points to
  ⇒ block b_1.
- Block b_1 has space available
  ⇒ insert point.



- Insert $p_6 = (0.82, 0.87)$:
  - Locate cell:
    - ∗ Take the x-value, 0.82, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.6,1.0]
      ⇒ corresponds to column 2.
    - ∗ Take the y-value, 0.87, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,1.0]
      ⇒ corresponds to row 1.
  - Retrieve Cell[2,1] of the grid directory.
  - Retrieve the block it points to
    ⇒ block b_1.

- Block b_1 is full
  $\Rightarrow$ split block.
- No existing partition works.
- *Current* = horizontal
  $\Rightarrow$ use horizontal partition so that half the points are in a block.
- For example, partition at 0.7.
- Add new row to grid directory.
- Update *Yscale* to contain new interval (with correct row number).
- *Current* := vertical.



2	b0	b2
1	b0	b1
	1	2

Grid directory

1	2	
0.0	0.7	1.0

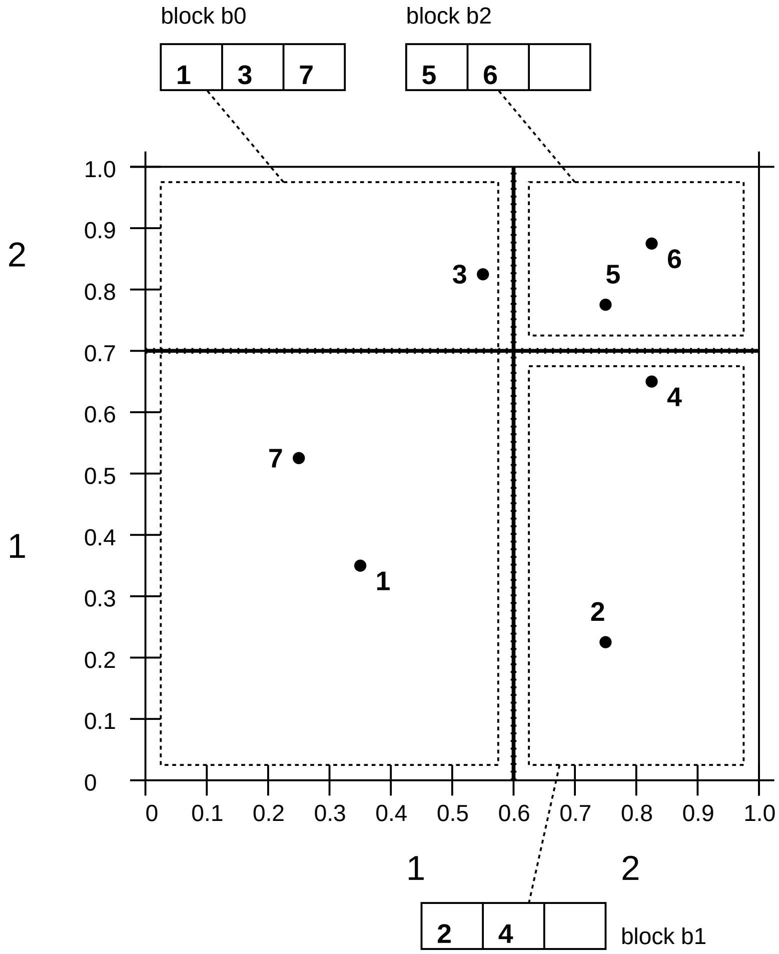
Yscale

1	2	
0.0	0.6	1.0

Xscale

- Insert $p_7 = (0.25, 0.53)$:

- Locate cell:
    - Take the x-value, 0.25, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.6]
      ⇒ corresponds to column 1.
    - Take the y-value, 0.53, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.7]
      ⇒ corresponds to row 1.
- Retrieve Cell[1,1] of the grid directory.
- Retrieve the block it points to
  ⇒ block b_0.
- Block b_0 has space available
  ⇒ insert point.



2	b0	b2
1	b0	b1
	1	2

Grid directory

1	2	
0.0	0.7	1.0

Yscale

1	2	
0.0	0.6	1.0

Xscale

- Insert $p_8 = (0.24, 0.77)$:
  - Locate cell:
    - ∗ Take the x-value, 0.24, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.6]
      ⇒ corresponds to column 1.
    - ∗ Take the y-value, 0.77, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.7,1.0]
      ⇒ corresponds to row 1.
  - Retrieve Cell[1,2] of the grid directory.

– Retrieve the block it points to
  ⇒ block b_0.

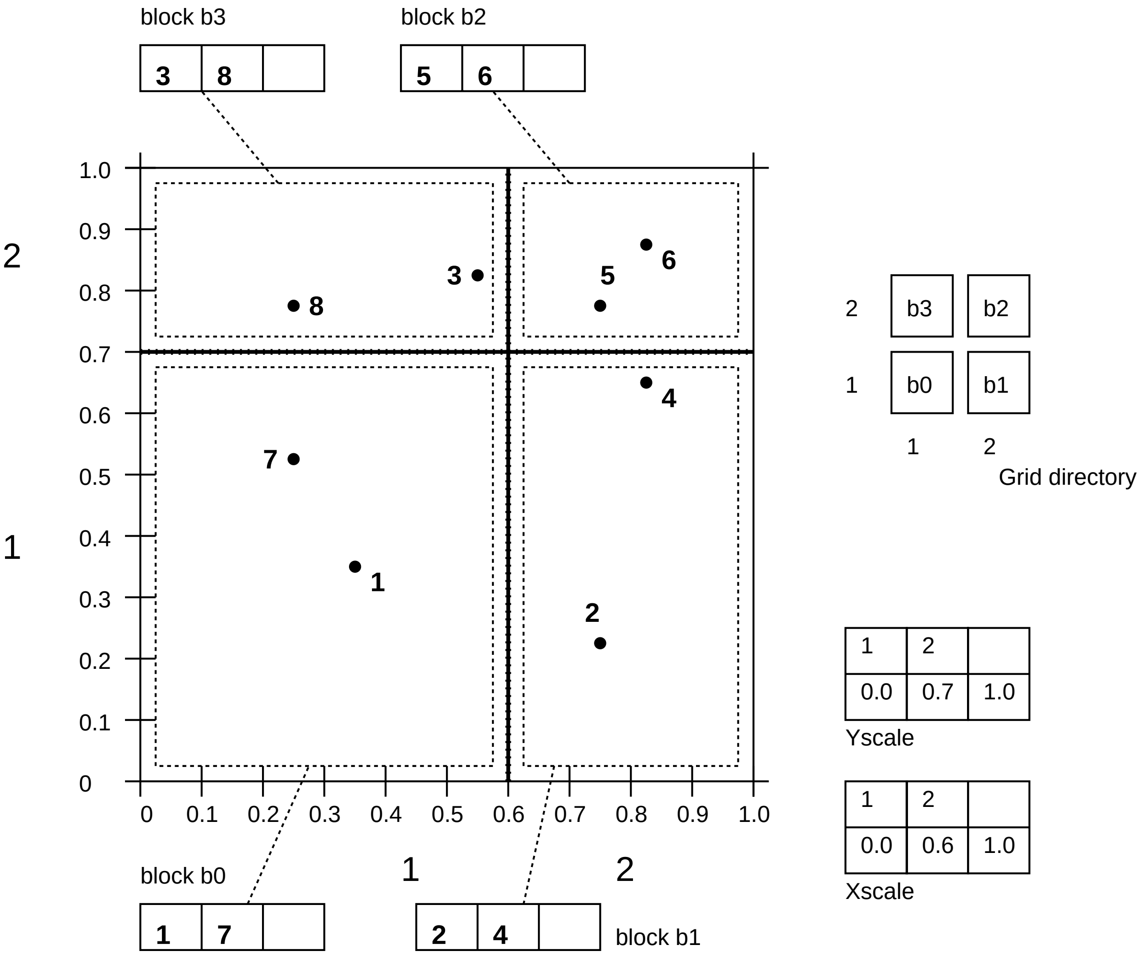
– Block b_0 is full
  ⇒ split block.

– *Current* = vertical
  ⇒ Try existing vertical partitions
  ⇒ None can be used.

– Horizontal partition at 0.7 can be used to split b_0.

– Update grid directory.

– *Current* := vertical.



2	b3	b2
1	b0	b1
	1	2

Grid directory

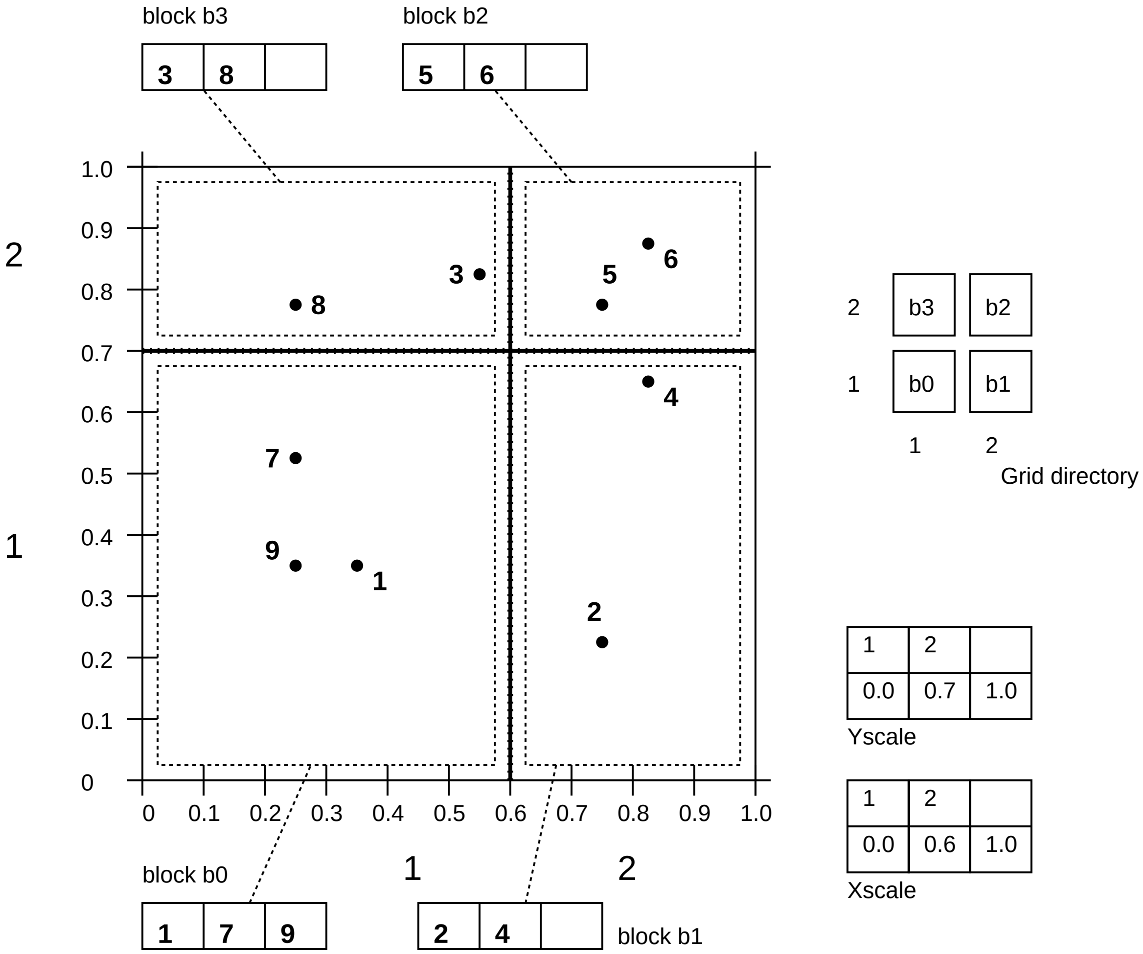
1	2	
0.0	0.7	1.0

Yscale

1	2	
0.0	0.6	1.0

Xscale

- Insert $p_9 = (0.23, 0.35)$:
  - Locate cell:
    - Take the x-value, 0.23, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.6]
      ⇒ corresponds to column 1.
    - Take the y-value, 0.35, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.7]
      ⇒ corresponds to row 1.
  - Retrieve Cell[1,1] of the grid directory.
  - Retrieve the block it points to
    ⇒ block b_0.
  - Block b_0 has space available
    ⇒ insert point.



Yscale

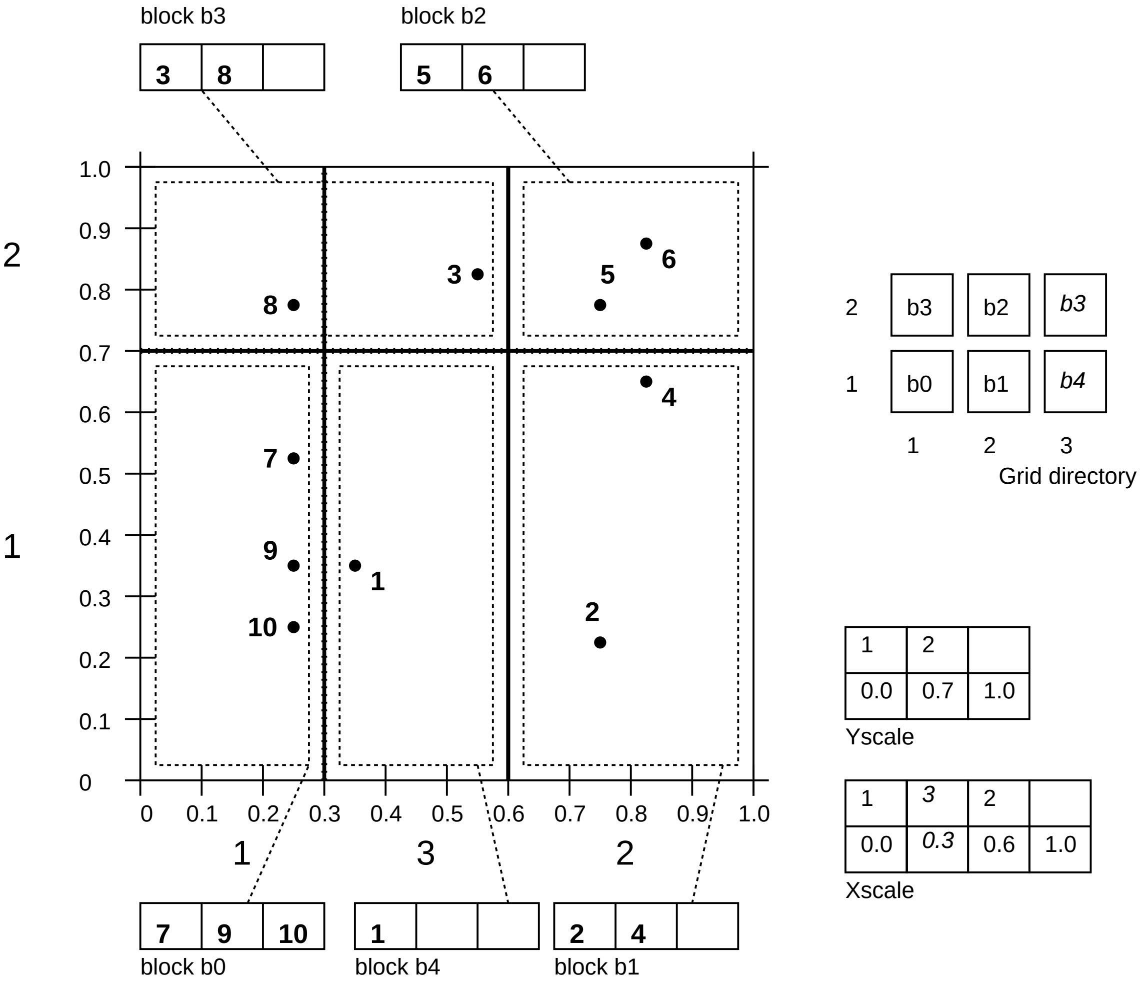
1	2	
0.0	0.7	1.0

Xscale

1	2	
0.0	0.6	1.0

- Insert $p_{10} = (0.25, 0.24)$:
  - Locate cell:
    - ∗ Take the x-value, 0.25, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.6]
      ⇒ corresponds to column 1.
    - ∗ Take the y-value, 0.24, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.7]
      ⇒ corresponds to row 1.
  - Retrieve Cell[1,1] of the grid directory.

- Retrieve the block it points to
  $\Rightarrow$ block b_0.
- Block b_0 is full
  $\Rightarrow$ split block.
- No existing partition works.
- *Current* = vertical
  $\Rightarrow$ use vertical partition so that half the points are in a block.
- For example, partition at 0.3.
- Add new column to grid directory.
- Update *Xscale* to contain new interval (with correct column number).
- **Note**: columns are out of order in *Xscale*.
- *Current* := horizontal.



2	b3	b2	*b3*
1	b0	b1	*b4*
	1	2	3

Grid directory

1	2	
0.0	0.7	1.0

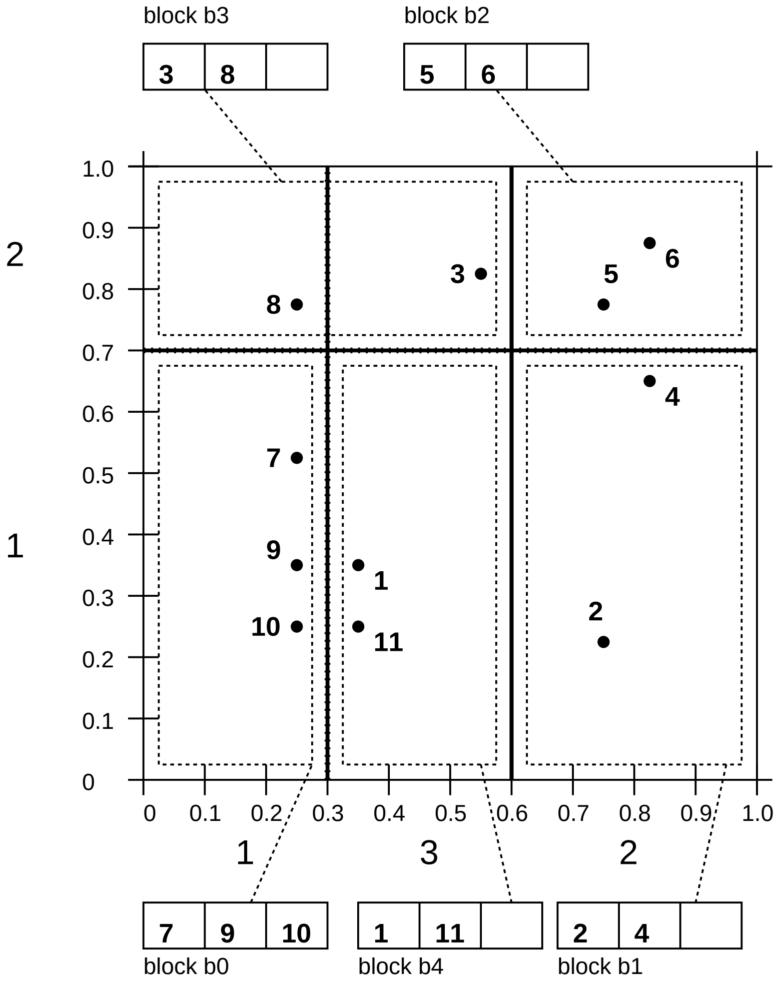
Yscale

1	*3*	2	
0.0	*0.3*	0.6	1.0

Xscale

- Insert $p_{11} = (0.31, 0.24)$:
  - Locate cell:
    - Take the x-value, 0.31, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.3,0.6]
      ⇒ corresponds to column 3.
    - Take the y-value, 0.24, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.7]
      ⇒ corresponds to row 1.
  - Retrieve Cell[3,1] of the grid directory.

- Retrieve the block it points to
  ⇒ block b_4.
- Block b_4 has space available
  ⇒ insert point.



2	b3	b2	b3
1	b0	b1	b4
	1	2	3

Grid directory

1	2	
0.0	0.7	1.0

Yscale

1	3	2	
0.0	0.3	0.6	1.0

Xscale

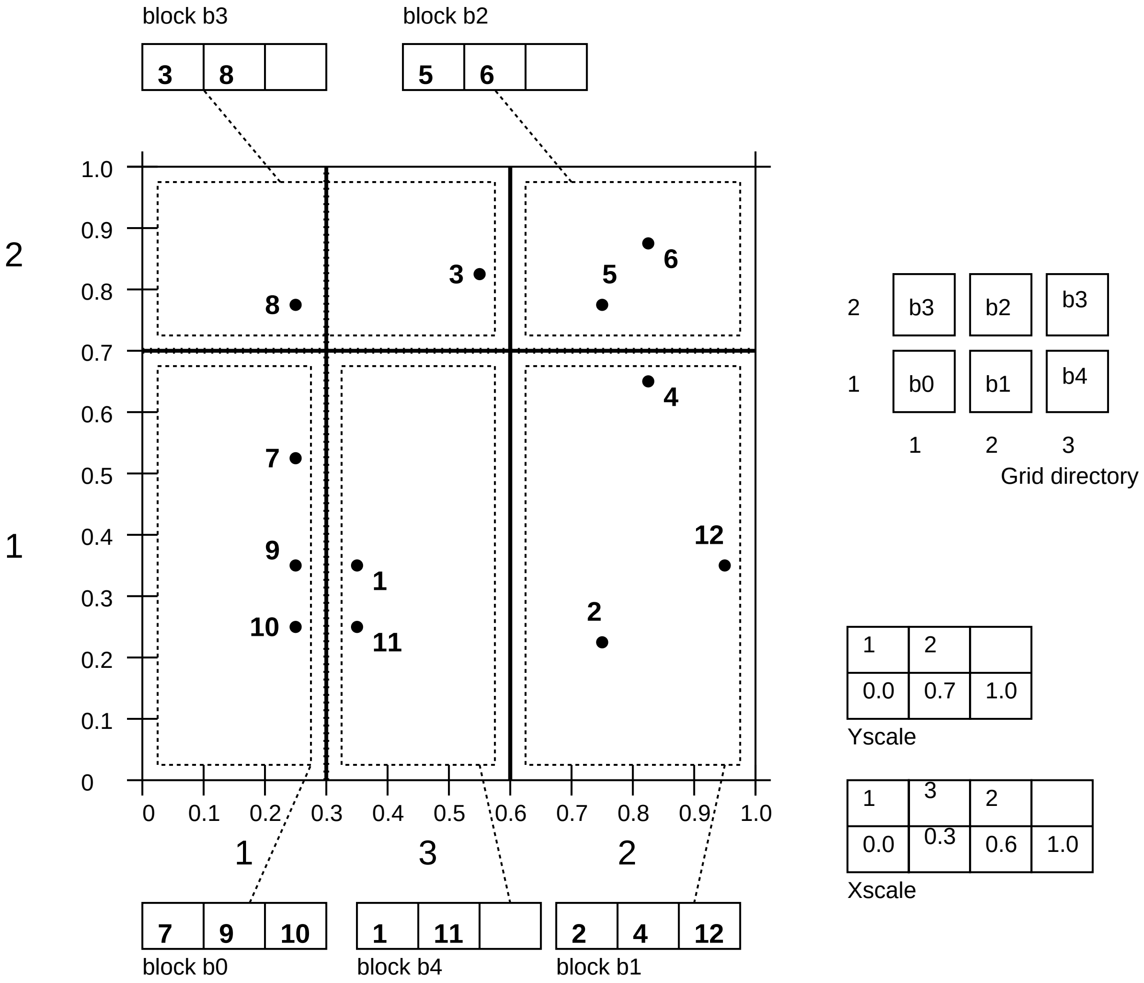
- Insert $p_{12} = (0.94, 0.35)$:
  - Locate cell:
    - ∗ Take the x-value, 0.94, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.6,1.0]
      ⇒ corresponds to column 2.

- ∗ Take the y-value, 0.35, and use the *Yscale* to see which interval it lies in
  ⇒ lies in the interval [0.0,0.7]
  ⇒ corresponds to row 1.

– Retrieve Cell[2,1] of the grid directory.

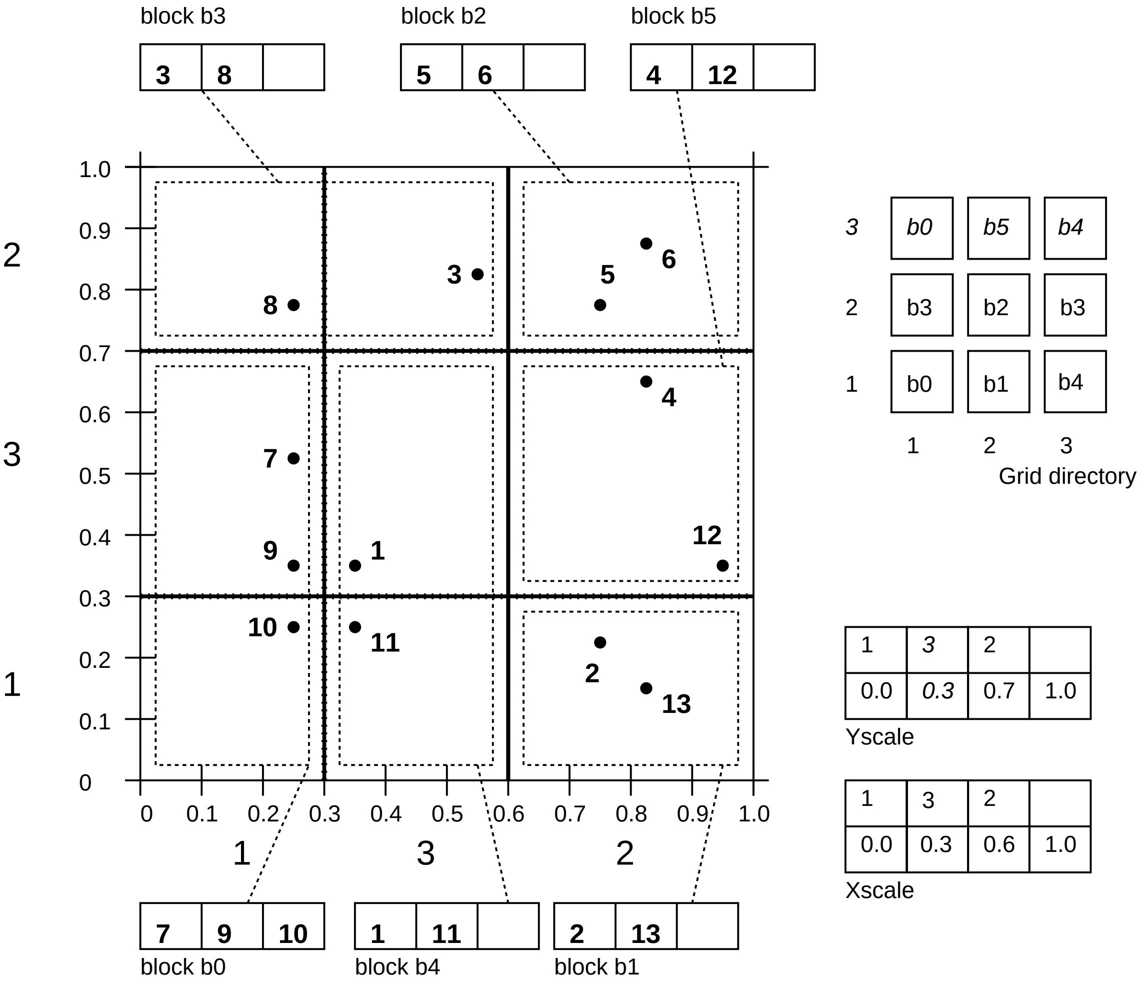
– Retrieve the block it points to
⇒ block b_1.

– Block b_1 has space available
⇒ insert point.



- Insert $p_{13} = (0.82, 0.15)$:

- Locate cell:
  - Take the x-value, 0.82, and use the *Xscale* to see which interval it lies in
    ⇒ lies in the interval [0.6,1.0]
    ⇒ corresponds to column 2.
  - Take the y-value, 0.15, and use the *Yscale* to see which interval it lies in
    ⇒ lies in the interval [0.0,0.7]
    ⇒ corresponds to row 1.
- Retrieve Cell[2,1] of the grid directory.
- Retrieve the block it points to
  ⇒ block b_1.
- Block b_1 is full
  ⇒ split block.
- No existing partition works.
- *Current* = horizontal
  ⇒ use horizontal partition so that half the points are in a block.
- For example, partition at 0.3.
- Add new row to grid directory.
- Update *Yscale* to contain new interval (with correct row number).
- **Note**: rows are out of order in *Yscale*.
- *Current* := vertical.



3	*b0*	*b5*	*b4*
2	b3	b2	b3
1	b0	b1	b4
	1	2	3

Grid directory

1	*3*	2	
0.0	*0.3*	0.7	1.0

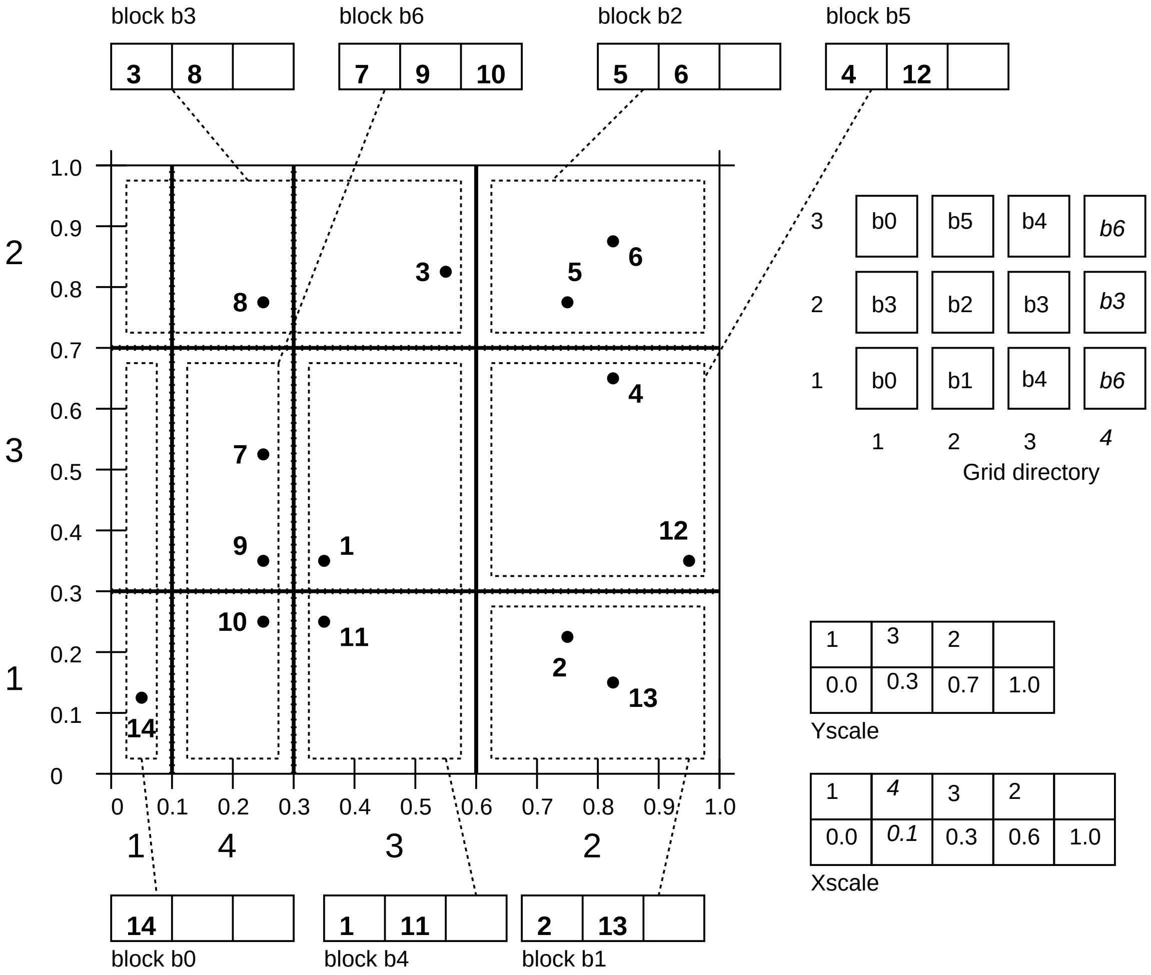
Yscale

1	3	2	
0.0	0.3	0.6	1.0

Xscale

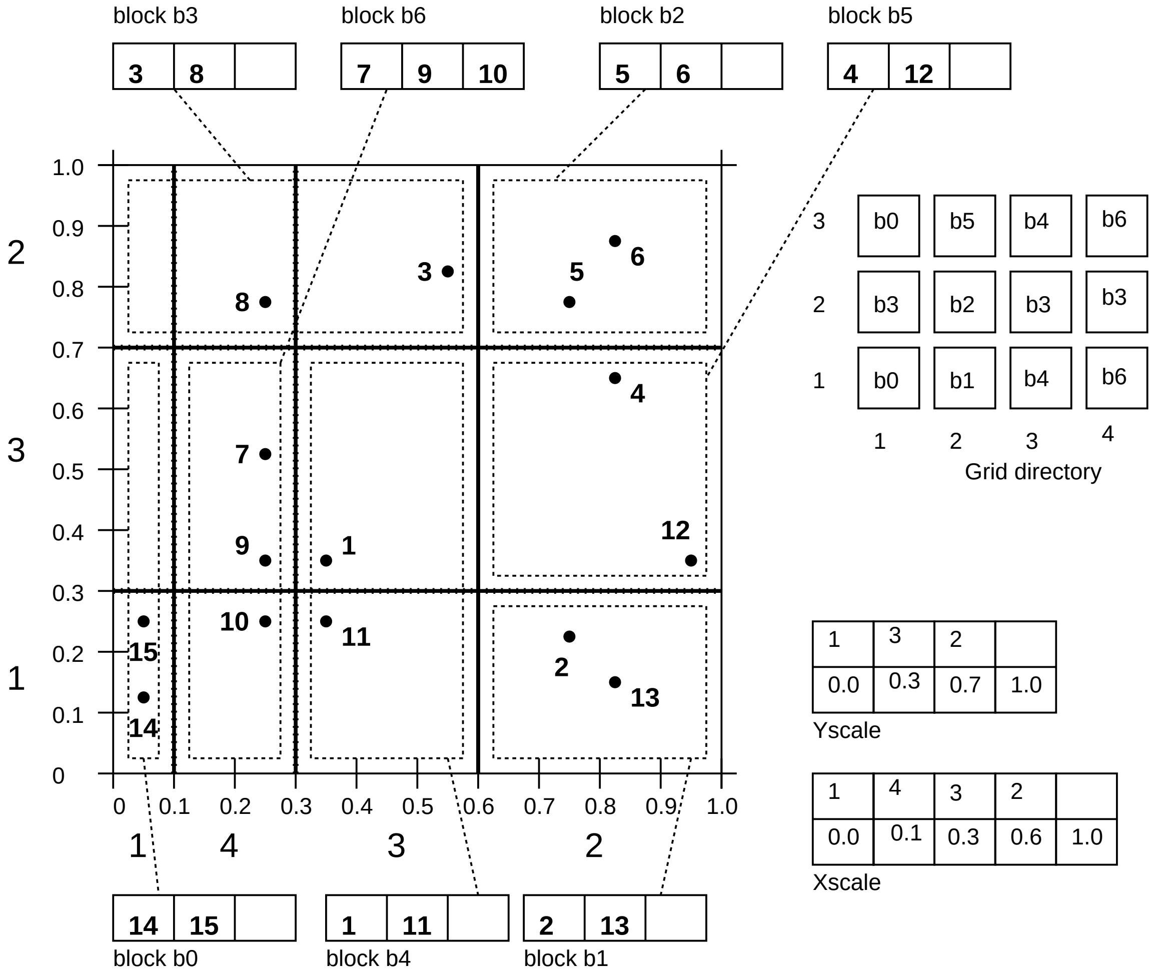
- Insert $p_{14} = (0.05, 0.12)$:
  - Locate cell:
    - Take the x-value, 0.05, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.3]
      ⇒ corresponds to column 1.
    - Take the y-value, 0.12, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.3]
      ⇒ corresponds to row 1.
  - Retrieve Cell[1,1] of the grid directory.

- Retrieve the block it points to
  ⇒ block b_0.
- Block b_0 is full
  ⇒ split block.
- The horizontal partition at 0.3 works.
- However, for the sake of creating an interesting example, we will change the policy a little and use a fresh vertical partition.
- *Current* = vertical
  ⇒ use vertical partition so that half the points are in a block.
- In this case, use partition at 0.1.
- Add new column to grid directory.
- Update *Xscale* to contain new interval (with correct column number).
- *Current* := horizontal.



- Insert $p_{15} = (0.03, 0.24)$:
  - Locate cell:
    - ∗ Take the x-value, 0.03, and use the *Xscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.1]
      ⇒ corresponds to column 1.
    - ∗ Take the y-value, 0.24, and use the *Yscale* to see which interval it lies in
      ⇒ lies in the interval [0.0,0.3]
      ⇒ corresponds to row 1.
  - Retrieve Cell[1,1] of the grid directory.

- Retrieve the block it points to
  ⇒ block b_0.
- Block b_0 has space available
  ⇒ insert point.



3	b0	b5	b4	b6
2	b3	b2	b3	b3
1	b0	b1	b4	b6
	1	2	3	4

Grid directory

1	3	2	
0.0	0.3	0.7	1.0

Yscale

1	4	3	2	
0.0	0.1	0.3	0.6	1.0

Xscale

- Implementation details: *Xscale* and *Yscale*
  - For example, consider *Xscale*.
  - Recall: *Xscale* provides the mapping between intervals on the x-axis and grid directory columns:

1	3	2	
0.0	0.3	0.7	1.0

Yscale

1	4	3	2	
0.0	0.1	0.3	0.6	1.0

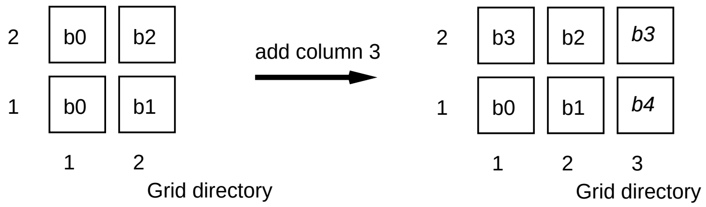
Xscale

  - Implementation options:
    1. Array implementation:
       * Each array element would contain interval endpoints and a row or column number, e.g.,

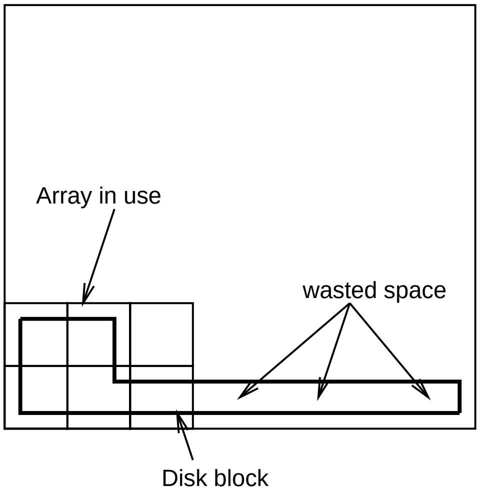
```
typedef struct scale_type {
  int index;
  double low, high;
};
scale_type *Xscale, *Yscale;
...
Xscale[2].low = 0.1;   Xscale[2].high = 0.3;
Xscale[2].index = 4;
```

       * Advantages: binary search can be used while searching.
       * Disadvantages: insertions (into array) require expansion of array
         ⇒ memory reallocation and copying overhead.
    2. Linked-list implementation:
       * Place *Xscale* elements in a linked-list.
       * Advantages: insertion is efficient.
       * Disadvantages: Search has linear cost. List needs to be created every time the scale is read from disk.

- Implementation details: the grid directory.
  - Recall: the grid directory is a 2D-array that adds columns and rows incrementally, e.g.,



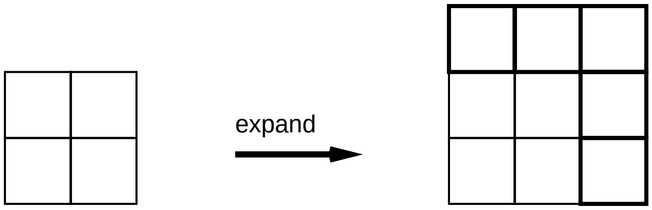
  - Implementation options:
    1. Standard row-major order array:
       - ∗ To implement row-major order, the row size has to be known ⇒ number of columns is needed in advance.
       - ∗ Initially, select a large enough array size and allocate space for it.
       - ∗ Note: when the desired array is small, space is wasted in array blocks:



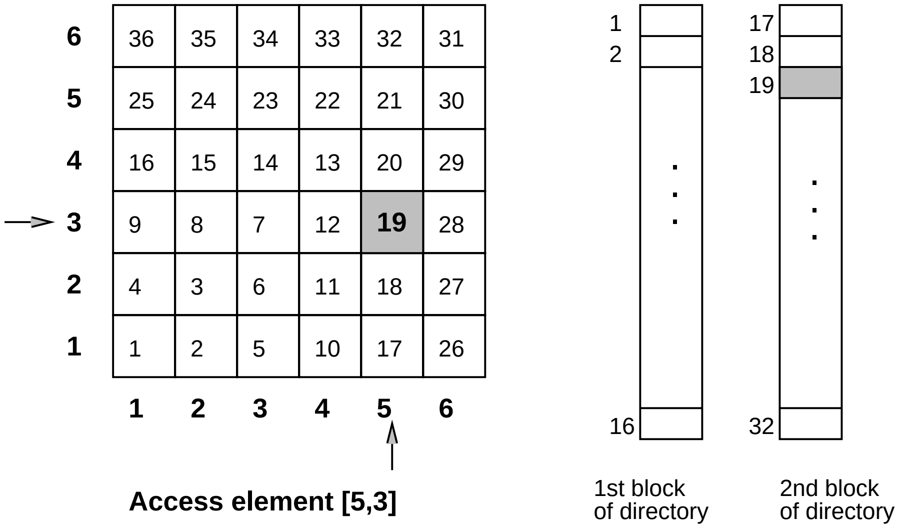
       - ∗ If array fills up, an expansion may be needed ⇒ allocate new, larger array and copy over data.
       - ∗ Advantages: straightforward implementation.

- Disadvantages: Reallocation and copying overhead during expansion. Inefficient use of space when array size is small.

2. L-order implementation:
   - Note: grid directory expansions can be quite irregular
     ⇒ several column expansions in sequence, for example.
   - The L-order implementation always adds both a row and column whenever an expansion is desired:



   - The array elements are stored in L-order:



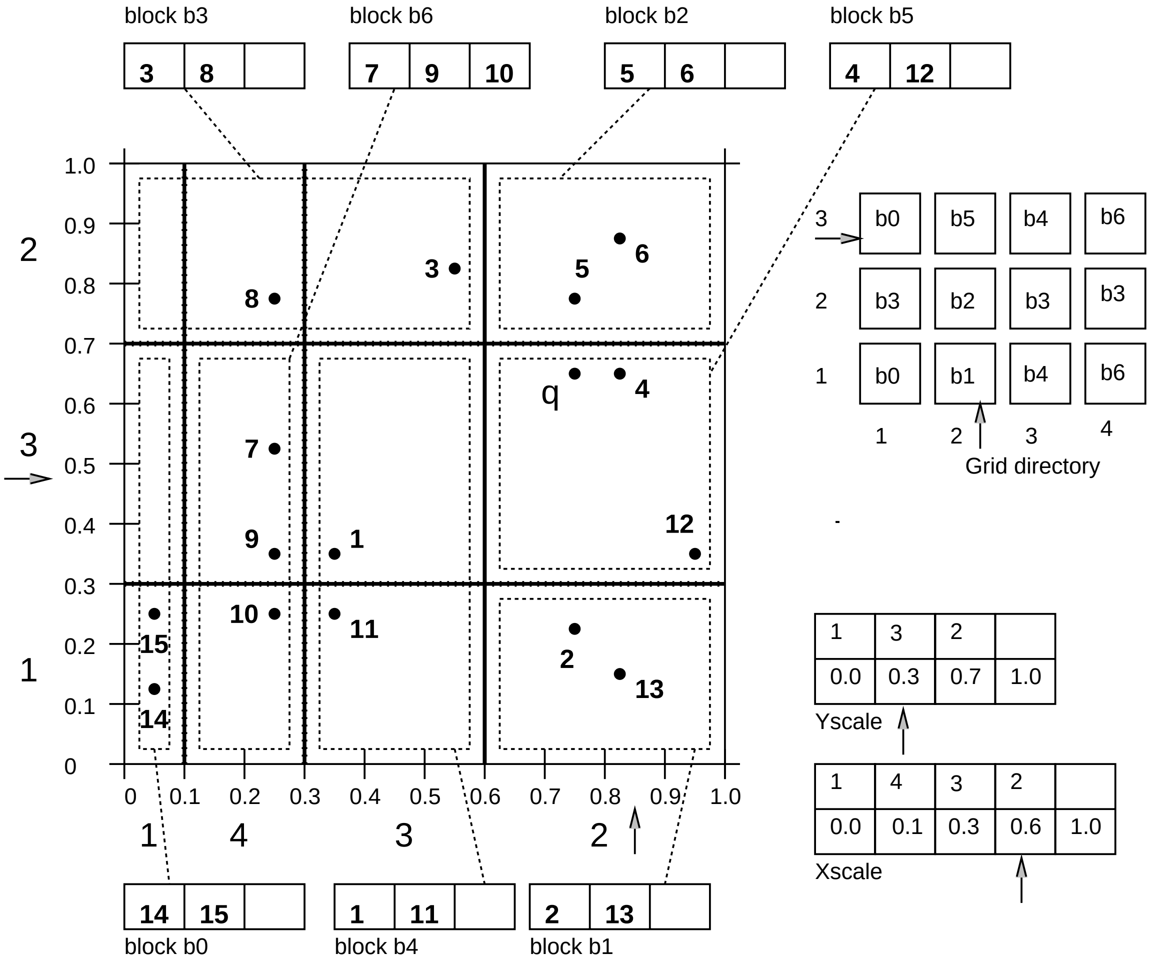
6	36	35	34	33	32	31
5	25	24	23	22	21	30
4	16	15	14	13	20	29
→ **3**	9	8	7	12	**19**	28
2	4	3	6	11	18	27
1	1	2	5	10	17	26
	1	**2**	**3**	**4**	**5**	**6**

   - To access element [5,3] of the array, we need to find the L-order number corresponding to [5,3]
     ⇒ L-order number is 19.
   - The L-order number is used to fetch the correct block and value within the block.
   - Advantages: if grid directory is small, it may occupy only a few blocks

⇒ grid directory can reside in memory.

$*$ Disadvantages: non-standard computation of array offsets.

- Next, consider a nearest neighbor query:
  - Example: consider the query $q_1 = (0.75, 0.65)$.
  - First, locate grid cell:
    - Find column using *Xscale*:
$\Rightarrow$ 0.75 $\in$ [0.6,1.0]
$\Rightarrow$ column 2.
    - Find row using *Yscale*:
$\Rightarrow$ 0.65 $\in$ [0.3,0.7]
$\Rightarrow$ row 3.
  - Retrieve grid cell [2,3]:
    - Compute L-order number of [2,3]
$\Rightarrow$ L-order number is 6.
    - Retrieve item from grid blocks.

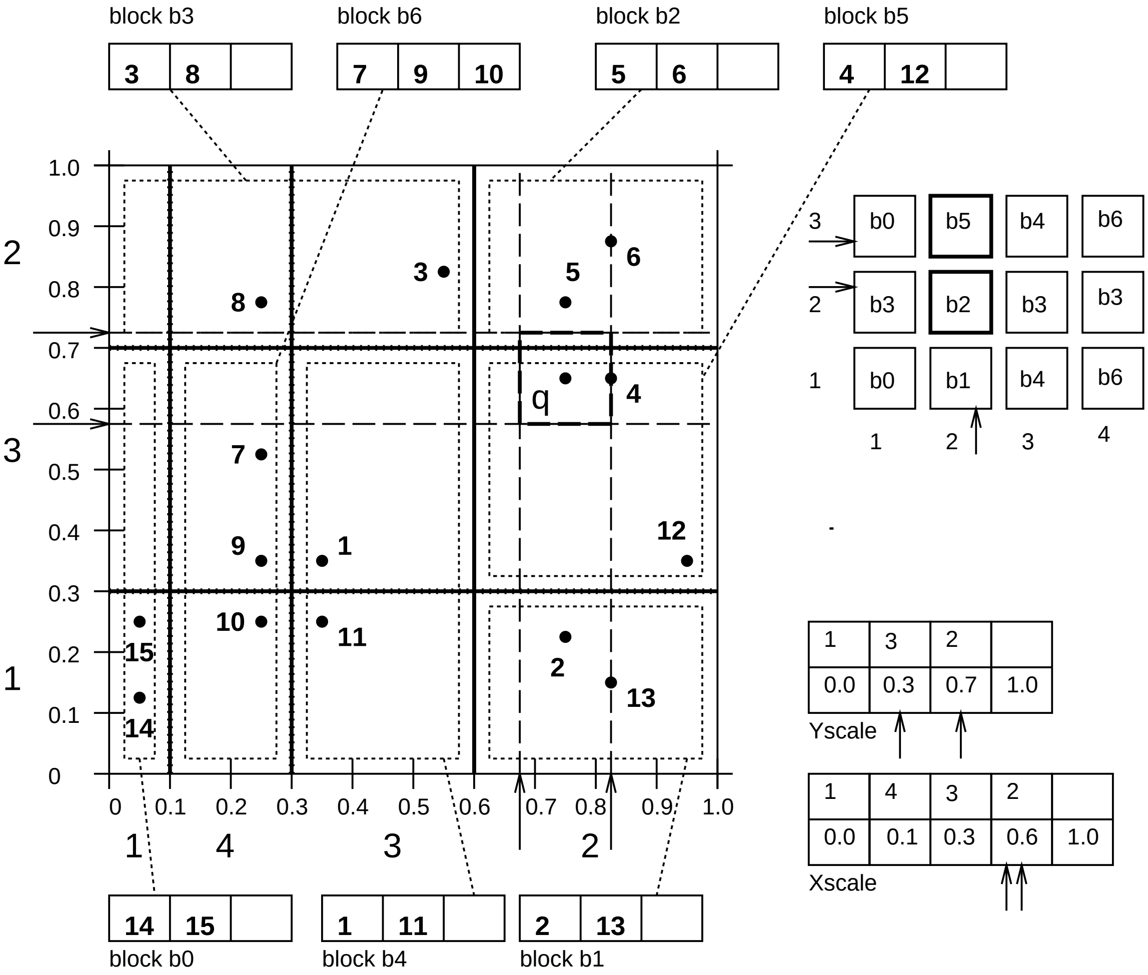


– Retrieve block pointed to by cell
  ⇒ block b_5.

– Closest point in b_5: $p_4 = (0.82, 0.66)$.

– $d = dist(q, p_4) = \left((0.75 - 0.82)^2 + (0.65 - 0.66)^2\right)^{1/2} = 0.07$.

– Consider a square with q as center and half-side$= 0.07$.



- Find all the x-intervals covered by square
  ⇒ only the [0.6,1.0] interval.
- Find all the y-intervals covered by square
  ⇒ [0.3,0.7] and [0.7,1.0].
- Retrieve corresponding grid cells
  ⇒ cells [2,3] and [2,3].
- Retrieve blocks pointed to by these cells
  ⇒ blocks b_2 and b_5.
- Test against all points in these blocks
  ⇒ p_4 is closest.

- Pseudocode for the variable-size case:
  - We will assume *Xscale* and *Yscale* are implemented with arrays (for ease of presentation).
  - The nearest-neighbor query is considered, but code for range search is omitted.
  - The following functions are used:
    - ∗ Gridfile-Create (xlow,xhigh,ylow,yhigh) – Create a new gridfile given MBR of data set.
    - ∗ Gridfile-Insert (p) – Insert the point p into the gridfile.
    - ∗ L-Order (i, j) – Find the L-order value of cell i, j.
    - ∗ Gridfile-Findcell (p) – Find the grid cell containing the point p.
    - ∗ Gridfile-Partition (S_X, S_Y) – Find a partition, using points sorted by x-value and by y-value.
    - ∗ Gridfile-Update-Directory (ptype,bnum,I,J) – Adjust directory entries for an existing partition.
    - ∗ Gridfile-Expand-Directory (ptype,pval,bnum,I,J) – Adjust directory entries for a newly created partition.
    - ∗ Gridfile-Nearest-Neighbor (q) – Locate the data point closest to the query point q.
  - We will assume the grid directory is simply a list of blocknumbers in L-order.

Algorithm: GRIDFILE-CREATE (xlow, xhigh, ylow, yhigh)

Input: MBR of region.

Output: First data block and first dir block, written to disk.

```
Xscale[1].low := xlow;
Xscale[1].high := xhigh;
Xscale[1].index := 1;
Yscale[1].low := ylow;
Yscale[1].high := yhigh;
Xscale[1].index := 1;
num_columns := 1;
num_rows := 1;
Current := vertical;
// Create new data block
bnum := DISK-NEWBLOCK();
$b$ := DISK-READBLOCK (bnum);
$b$ →num_entries := 0;
DISK-WRITEBLOCK (bnum);
Compute max_entries allowed in a block;
min_entries := $\lfloor \alpha * \text{max\_entries} \rfloor$ ; // Usually, $\alpha = 0.5$
$k$ := L-ORDER $(1, 1)$;
Set $k$-th entry of directory file to bnum and write to disk;
return;
```

Algorithm: Gridfile-Insert (p)

Input: point p.

Output: p is inserted into the gridfile.

```
// First, find cell and get appropriate data block.
$(I, J)$ := Gridfile-Findcell ($p$);
$(i, j)$ := (Xscale[$I$].index, Yscale[$J$].index);
$k$ := L-Order $(i, j)$;
bnum := $k$-th entry in grid directory;
$b$ := Disk-Readblock (bnum);
// Check if space available for direct insert.
if $b \rightarrow$num_entries < max_entries;
   Insert point in $b$ and update $b$'s header;
   Disk-Writeblock (bnum);
   return;
endif;
// Otherwise, block needs to be split.
$S$ := $\{$points in $b\} \cup \{p\}$;
$S_X$ := Sort-By-Xvalue ($S$);
$S_Y$ := Sort-By-Yvalue ($S$);
// Find the partition.
(ptype, pval, $I'$, $J'$) := Gridfile-Partition $(S_X, S_Y)$;
// Create new block, assign points between old and new blocks.
bnum2 := Disk-Newblock ();
$b2$ := Disk-Readblock (bnum2);
if ptype $\in$ {existing_vertical, new_vertical}
   for each $p \in S_X$
      if $p.x <$ pval
         put $p$ in block $b$;
      else
         put $p$ in block $b2$;
else // partition was horizontal
   // Similar to above – code omitted.
endif;
// continued...
```

Algorithm: Gridfile-Insert ... continued

```
// Write the blocks and update directory.
Disk-Writeblock (bnum);
Disk-Writeblock (bnum2);
if ptype = existing_vertical
    Gridfile-Update-Directory (vertical, bnum2, $I'$, $J$);
else if ptype = new_vertical
    Gridfile-Expand-Directory (vertical, pval, bnum, bnum2, $I'$, $J$);
else if ptype = existing_horizontal
    Gridfile-Update-Directory (horizontal, bnum2, $I$, $J'$);
else
    Gridfile-Expand-Directory (horizontal, pval, bnum, bnum2, $I$, $J'$);
return;
```

Algorithm: Gridfile-Findcell (p)

Input: point p.
Output: coordinates of the cell containing point p.

```
Find $I$ such that Xscale[$I$].low $\leq p.x <$ Xscale[I].high;
Find $J$ such that Yscale[$J$].low $\leq p.y <$ Yscale[I].high;
return ($I$, $J$);
```

Algorithm: GRIDFILE-PARTITION (S_X, S_Y)

```
Input: sets of points $S_X, S_Y$.
Output: partition type, partition, index into Xscale or Yscale.
      // Try existing partitions first.
      if Current = vertical
        for $I$ := 1 to num_columns-1 do
          pval := Xscale[$I$].high;
          $S'$ := $\{p \in S_X : p.x < \text{pval}\}$;
          if $|S'| \geq$min_entries and $|S - S'| \geq$min_entries
            // We've found a suitable partition.
            Current := horizontal;
            return (existing_vertical, pval, $I$, 0);
          endif
        endfor
        // Now try existing horizontal partitions.
        for $J$ := 1 to num_rows-1 do
          pval := Yscale[$J$].high;
          $S'$ := $\{p \in S_Y : p.y < \text{pval}\}$;
          if $|S'| \geq$min_entries and $|S - S'| \geq$min_entries
            Current := vertical;
            return (existing_horizontal, pval, $J$, 0);
          endif
        endfor
      else // Try horizontal first, then vertical;
        // Similar to above - code omitted.
      endif
      // If existing partitions didn't work, we need to create one;
      if Current = vertical
        pval := average of the two middle values in $S_X$;
        Current := horizontal;
        Find $I$ such that Xscale[$I$].low $\leq$ pval $<$ Xscale[$I$].high;
        return (new_vertical, pval, $I$, 0);
      else // Find a new horizontal partition.
        // Similar to above – code omitted.
      endif;
      return;
```

Algorithm: GRIDFILE-UPDATE-DIRECTORY (ptype, bnum, I, J)

Input: partition type, block number, Xscale or Yscale index.
Output: Change block pointer to bnum in appropriate cell.

```
if ptype = vertical
    $i$ := Xscale[$I + 1$].index;
    $j$ := Yscale[$J$].index;
    $k$ := L-ORDER $(i, j)$;
    Change $k$-th entry in directory to bnum and write to disk;
    return;
else // Horizontal partition
    // Code omitted – similar to above.
endif;
```

Algorithm: Gridfile-Expand-Directory (ptype, pval, bnum, bnum2, I, J)

Input: partition type and value, block number, Xscale or Yscale index.
Output: new row or column in directory.

```
if ptype = vertical
    // Add a new column to Xscale, shift right.
    num_columns := num_columns + 1;
    for $i$ := num_columns downto $I + 1$;
        Xscale[$i$] := Xscale[$i - 1$];
    Xscale[$I + 1$].low := pval;
    Xscale[$I$].high := pval;
    Xscale[$I + 1$].index := num_columns; //New column #
    // Next, we need to create a new column.
    $i$ := Xscale[$I$].index;
    $i'$ := Xscale[$I + 1$].index;
    for $j$ := 1 to num_rows do
        // Copy column $i$ to new column $i'$.
        $k$ := L-Order ($i, j$);
        blknum := $k$-th entry in directory file;
        // Replace each occurence of bnum with bnum2.
        if blknum = bnum
            blknum := bnum2;
        $k'$ := L-Order ($i', j$);
        Write blknum to entry $k'$ in directory file;
    endfor;
    return;
else // Horizontal partition.
    // Code is omitted – similar to above.
endif;
```

Algorithm: Gridfile-Nearest-Neighbor (q)

Input: query point q.
Output: closest point to q in data set.

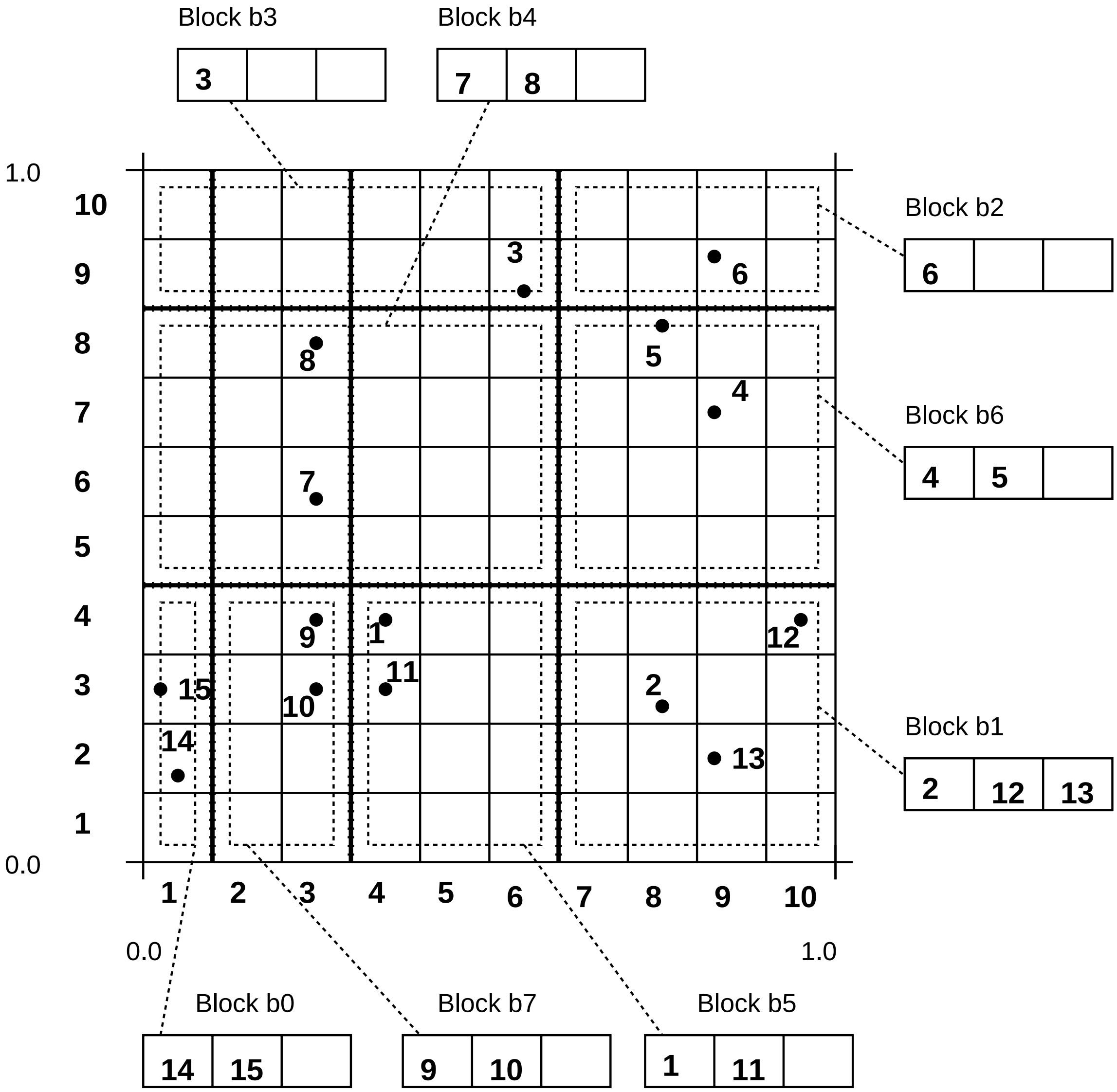
```
$(I, J)$ := Gridfile-Findcell ($p$);
$(i, j)$ := (Xscale[$I$].index, Yscale[$J$].index);
$k$ := L-Order ($i, j$);
bnum := $k$-th entry in directory file;
$b$ := Disk-Readblock (bnum);
$p$ := closest point in $b$ to query point $q$;
$d$ := distance($p, q$);
closest := $p$;
Create new blocklist $B$ and add bnum to it;
Consider square $S$ about $q$ with halfside=$d$;
Find all cells $C_1 = (i_1, j_1), \ldots, C_n = (i_n, j_n)$ that overlap $S$;
Sort list of cells by distance to $q$ (using closest corner);
for $m$ := 1 to $n$ do
   if distance($C_m, q$) $<$ $d$
      $k$ := L-Order ($i_m, j_m$);
      blknum := $k$-th entry in directory file;
      if blknum $\notin$ $B$
         Add blocknum to $B$;
         $p$ := closest point in blknum to $q$;
         $d'$ := distance($p, q$);
         if $d' < d$
            $d$ := $d'$;
            closest := $p$;
         endif;
      endif;
   endif;
endfor;
return (closest, $d$);
```

11.14 Grid Files: Deletion

- Deletion operation: given a point, delete it from the grid file.
- Two approaches to deletion:
    1. Lazy deletion:
        - Locate point in data block:
            - Identify correct grid cell.
            - Retrieve block pointed to by cell.
        - Delete point from block.
        - Leave grid directory untouched.
        - Periodically, rebuild entire grid file.
    2. Full deletion:
        - Locate point in data block:
            - Identify correct grid cell.
            - Retrieve block pointed to by cell.
        - Decide whether block underflows (too few points in block).
        - If underflow, then merge block with a suitable "neighboring" block
          ⇒ part of *merging policy.*
- Lazy deletion is straightforward.
  ⇒ we will only consider full deletion.
- A *merging policy* involves
    1. Deciding when a block underflows (and needs to be merged).
    2. Deciding which (among several neighboring blocks) to merge with.

  Note: it is not always possible to merge (see example below).

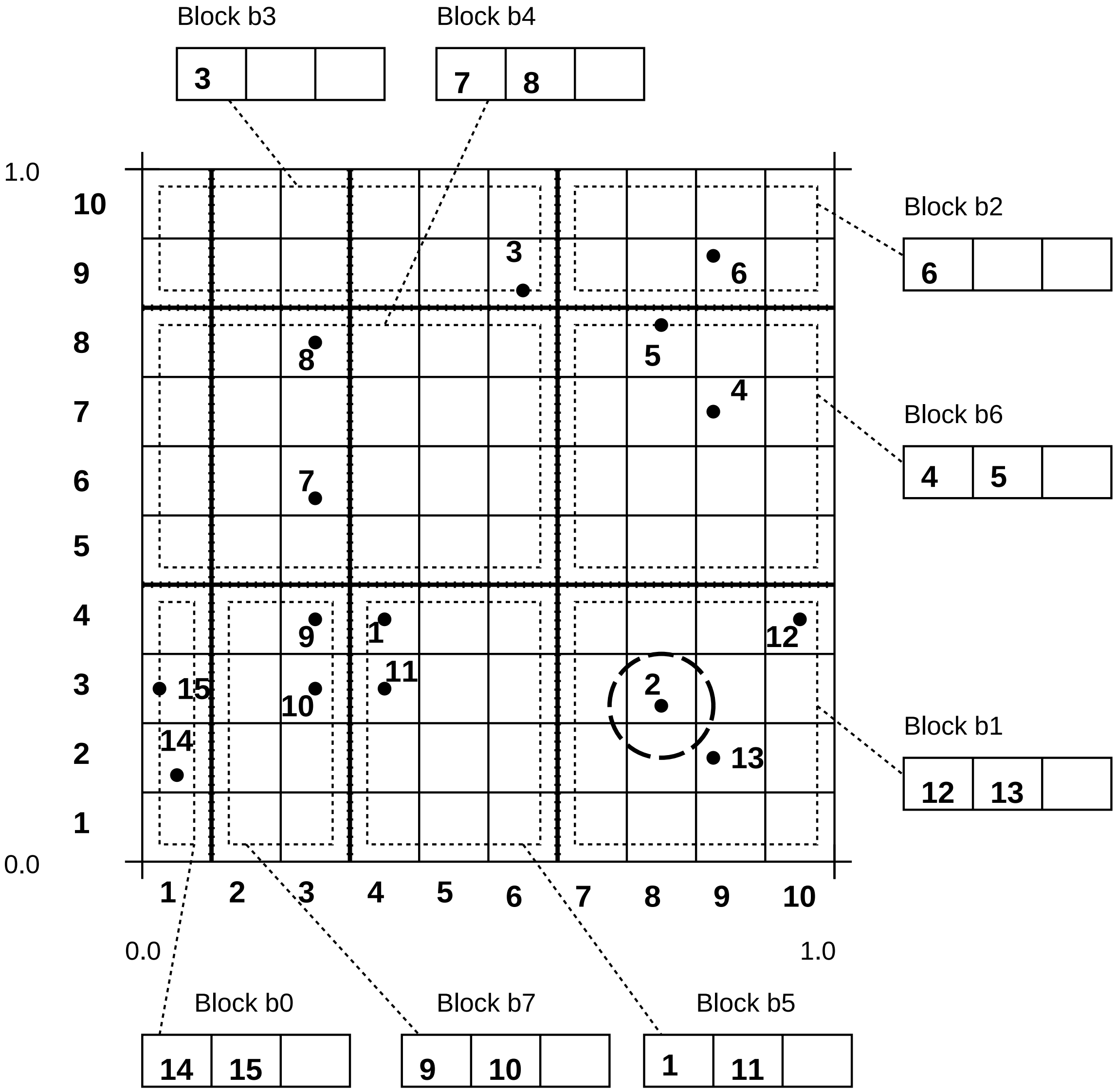
- Consider the following example:
  - Fixed-size cell version with the following points:



  - Delete following points (in order): $p_2, p_{13}, p_6, p_3, p_{14}, p_7, p_{10}, p_8, p_{11}$.
  - Merging policy:
    - ∗ Merge when a block is less than half-full
      ⇒ in this example, merge when a block has 0 or 1 points.
    - ∗ Among neighbors, pick the cell with the fewest points. If there are several, pick in clockwise order starting from North.

    - ∗ Merging is triggered only by deletes.

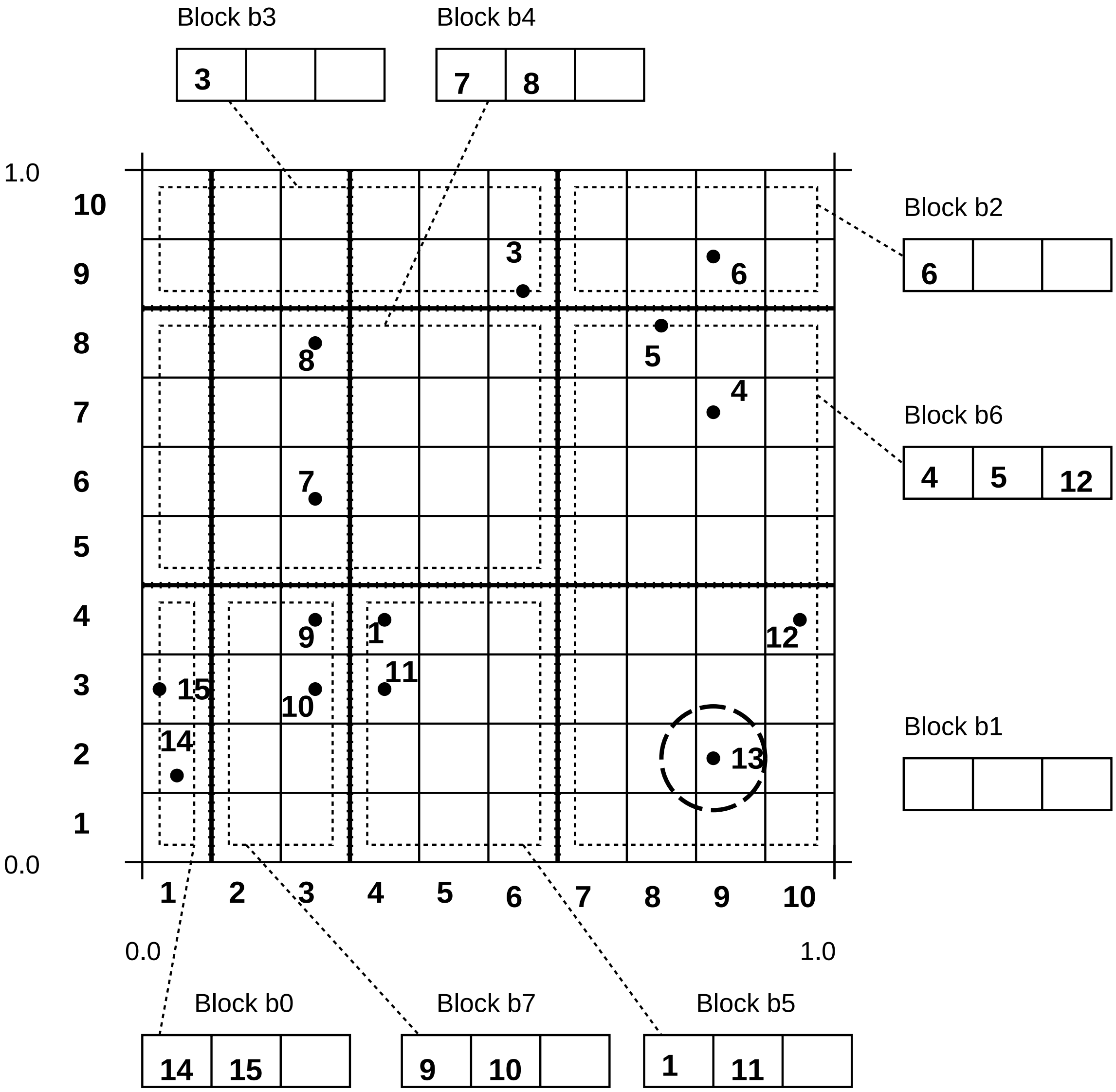
- Delete $p_2 = (0.74, 0.23)$:
  - Locate cell
    ⇒ Cell[8,3].
  - Retrieve block
    ⇒ block b_1.
  - Delete from block b_1.
  - Block does not underflow.



- Delete $p_{13} = (0.82, 0.15)$:
  - Locate cell
    $\Rightarrow$ Cell [9,2].
  - Retrieve block
    $\Rightarrow$ block b_1.
  - Delete from block
    $\Rightarrow$ causes underflow.

– Can merge with either block b_7 or block b_6.
⇒ select b_6 (priority: start from North).

– Return block b_1 to free space.



- Delete $p_6 = (0.82, 0.87)$:

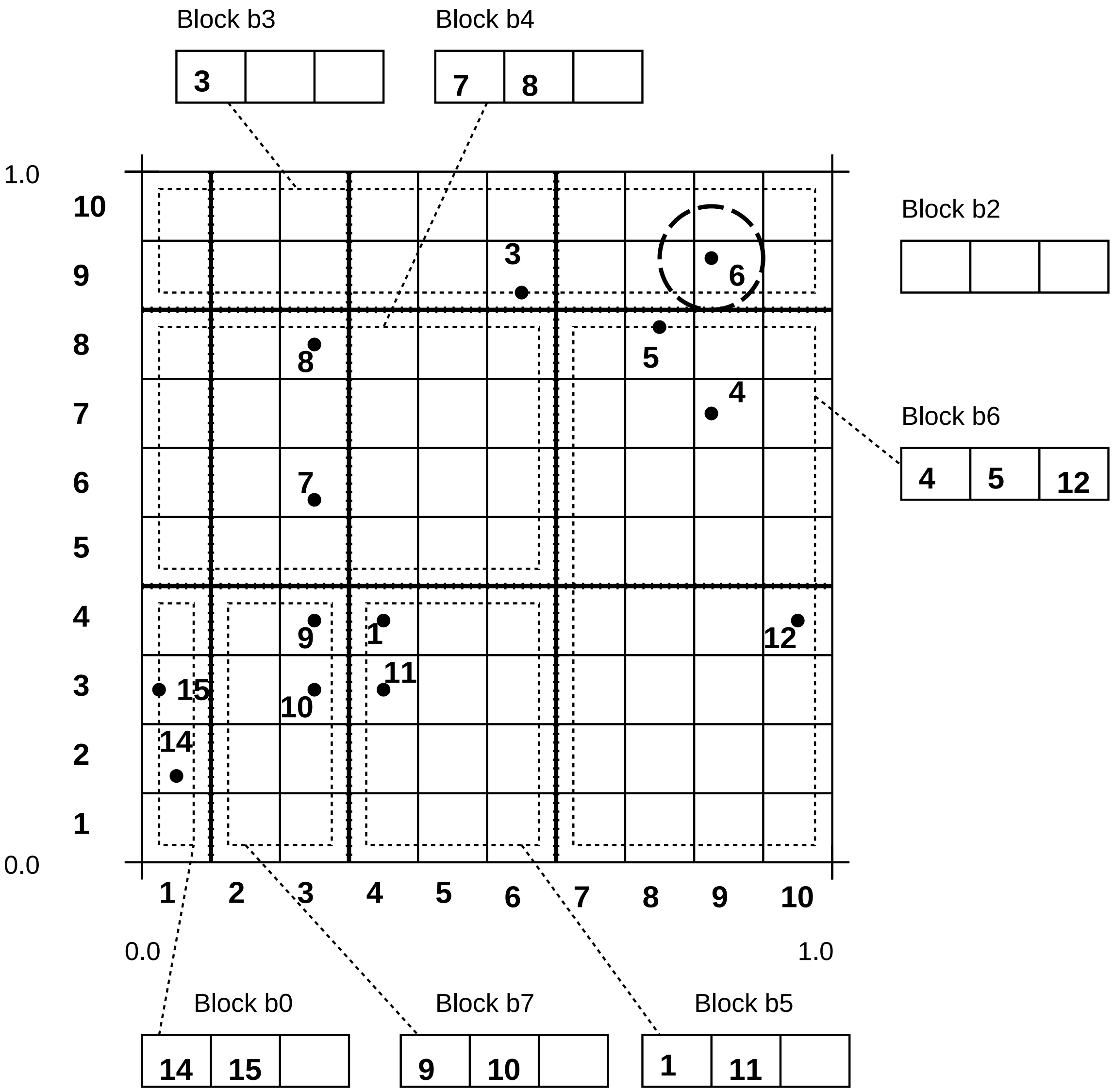
– Locate cell
⇒ Cell [9,9].

– Retrieve block
  ⇒ block b_2.

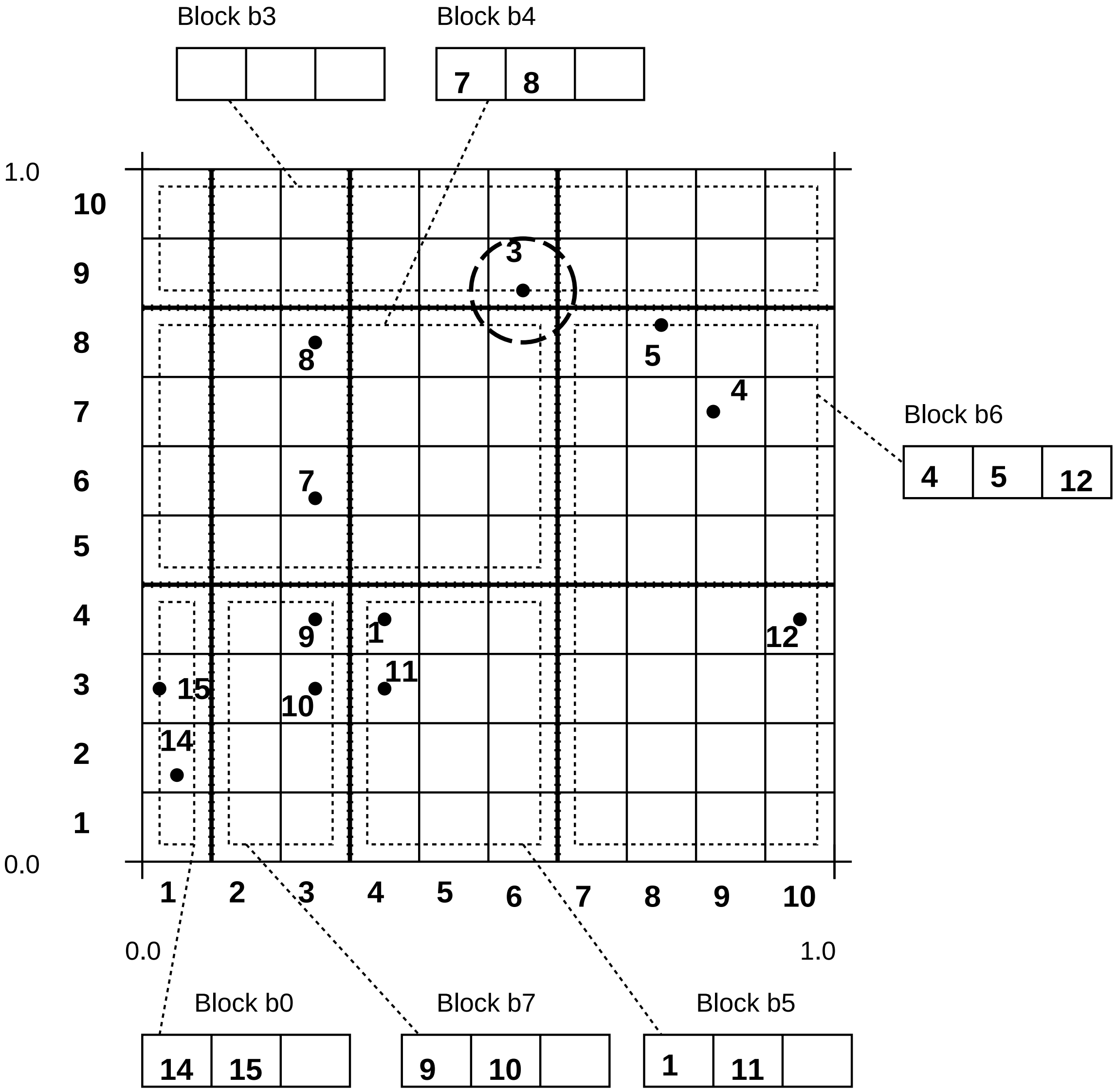
– Delete from block
  ⇒ causes underflow.

– Among neighboring blocks (b_3 and b_6), b_3 has the fewest elements
  ⇒ merge with b_3.

– Return b_2 to free space.



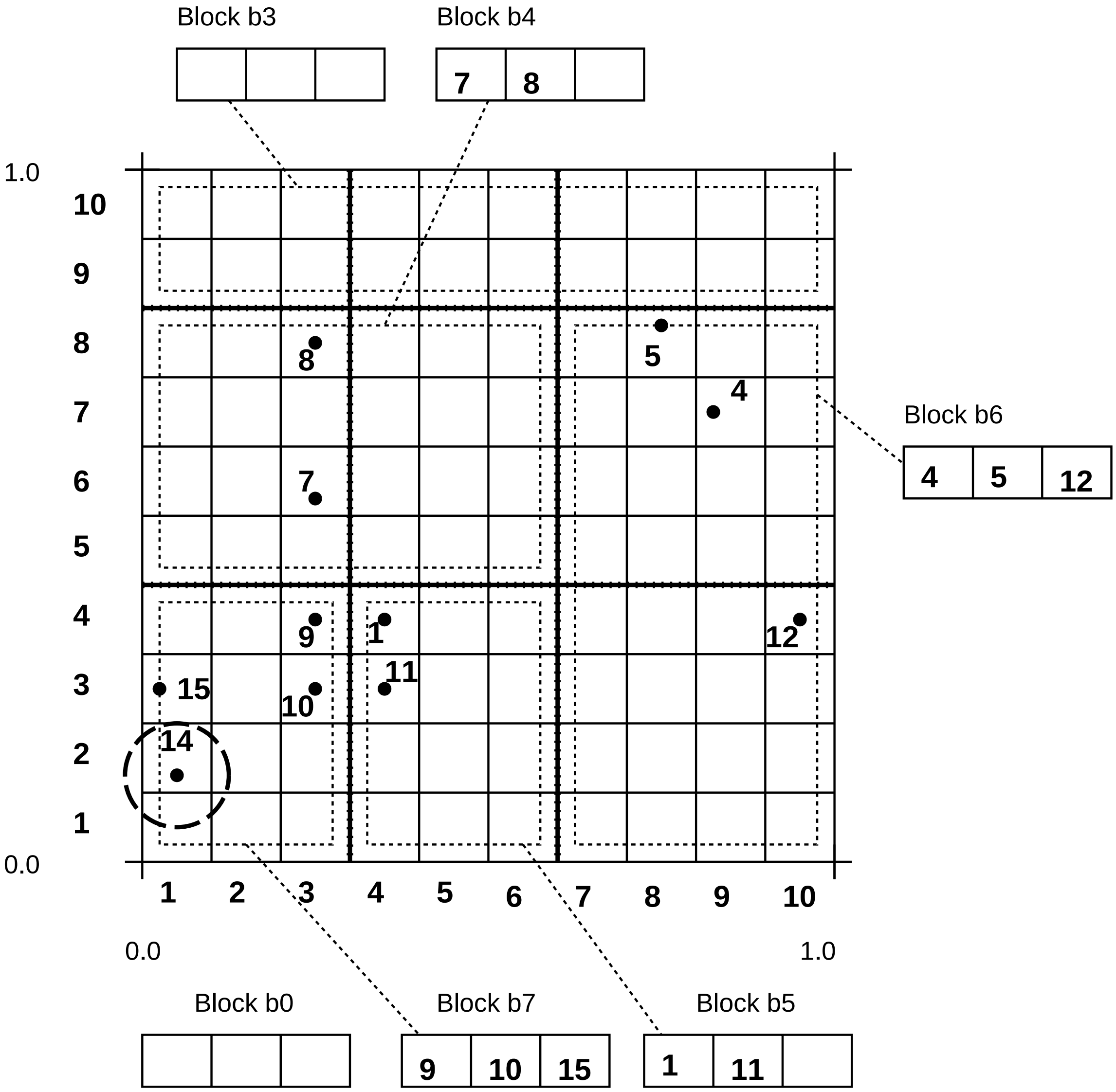
- Delete $p_3 = (0.55, 0.81)$:
    - Locate cell
      $\Rightarrow$ Cell [6,9].
    - Retrieve block
      $\Rightarrow$ block b_3.
    - Delete from block
      $\Rightarrow$ causes underflow.
    - Cannot merge with any other block
      $\Rightarrow$ retain empty block.



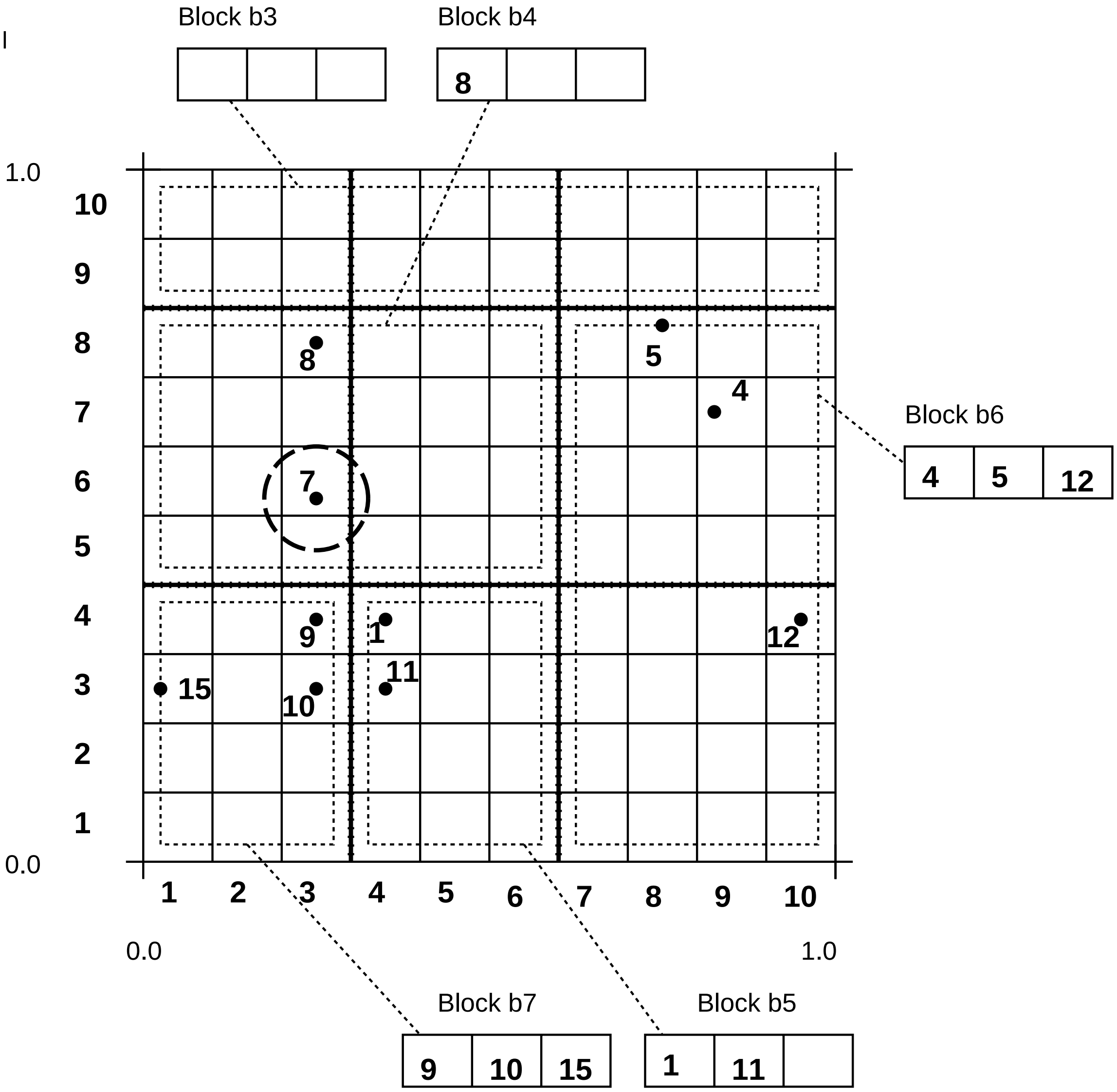
- Delete $p_{14} = (0.05, 0.12)$:
  - Locate cell
    ⇒ Cell [1,2].
  - Retrieve block
    ⇒ block b_0.
  - Delete from block
    ⇒ causes underflow.
  - Block b_0 can only be merged with b_7.

– Remove partition after column 1.

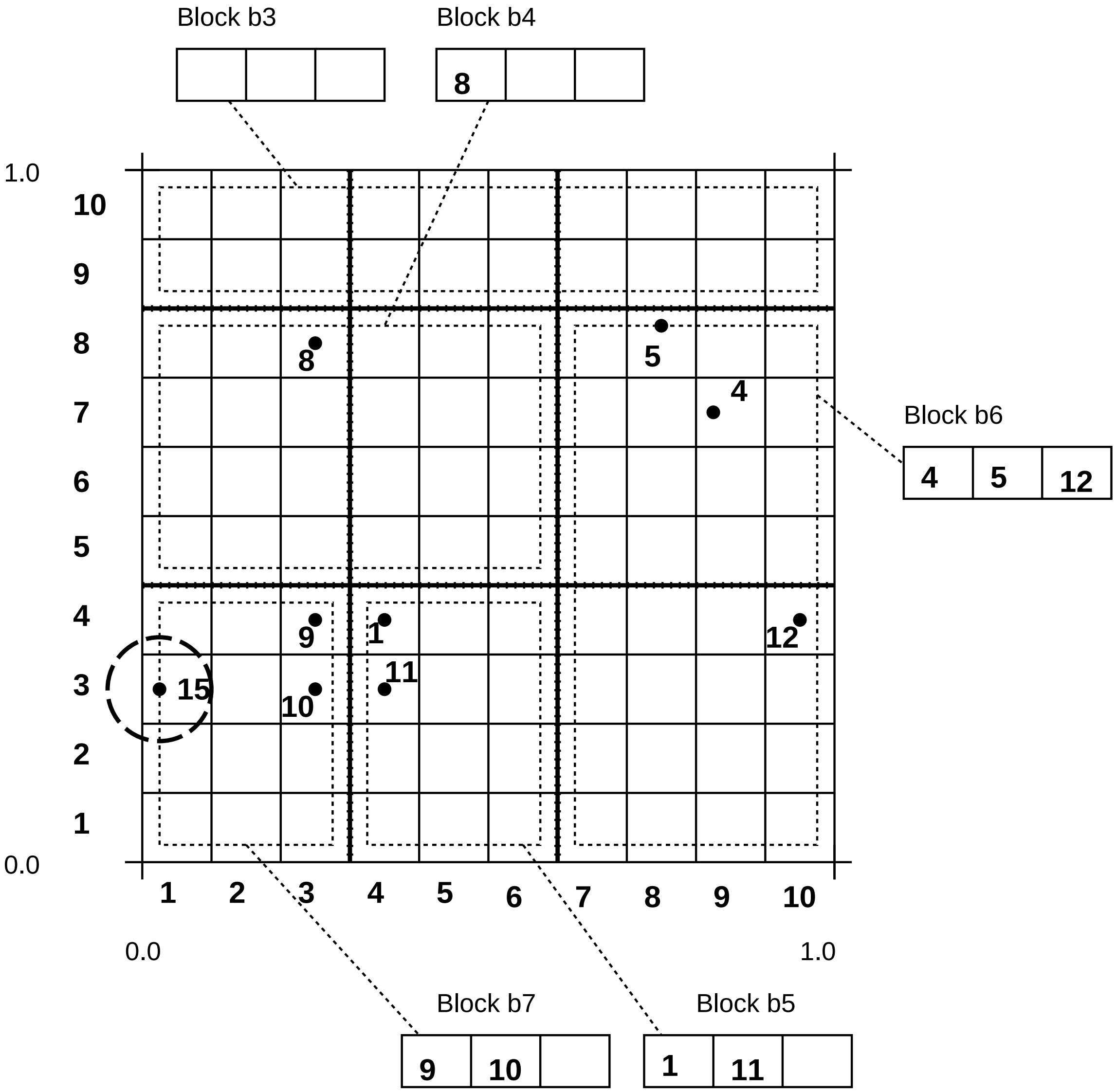
– Return b_0 to free space.



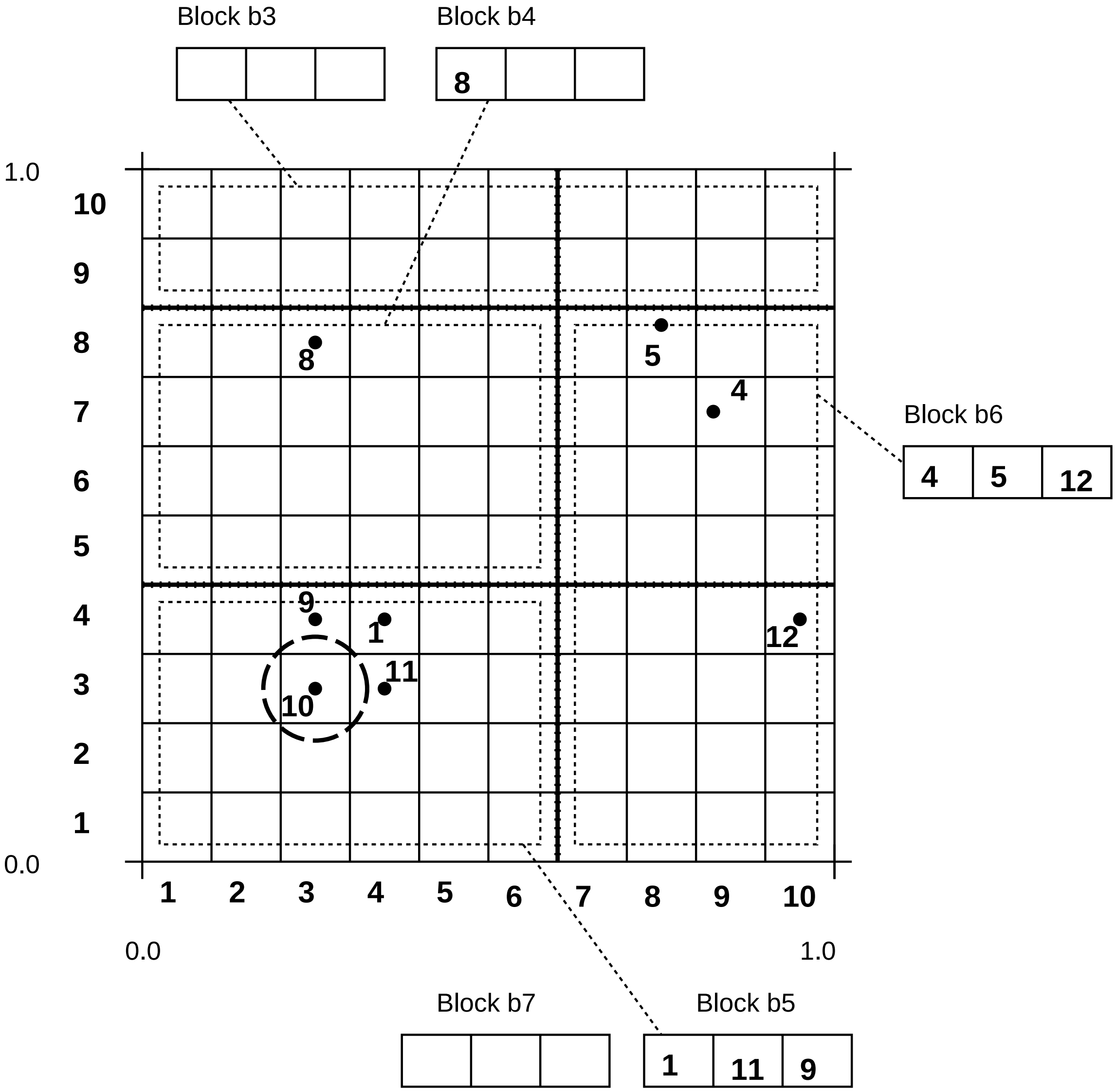
- Delete $p_7 = (0.25, 0.53)$:
  - Locate cell
    $\Rightarrow$ Cell [3,6].
  - Retrieve block
    $\Rightarrow$ block b_4.
  - Delete from block
    $\Rightarrow$ causes underflow.
  - Can't merge with any other block.



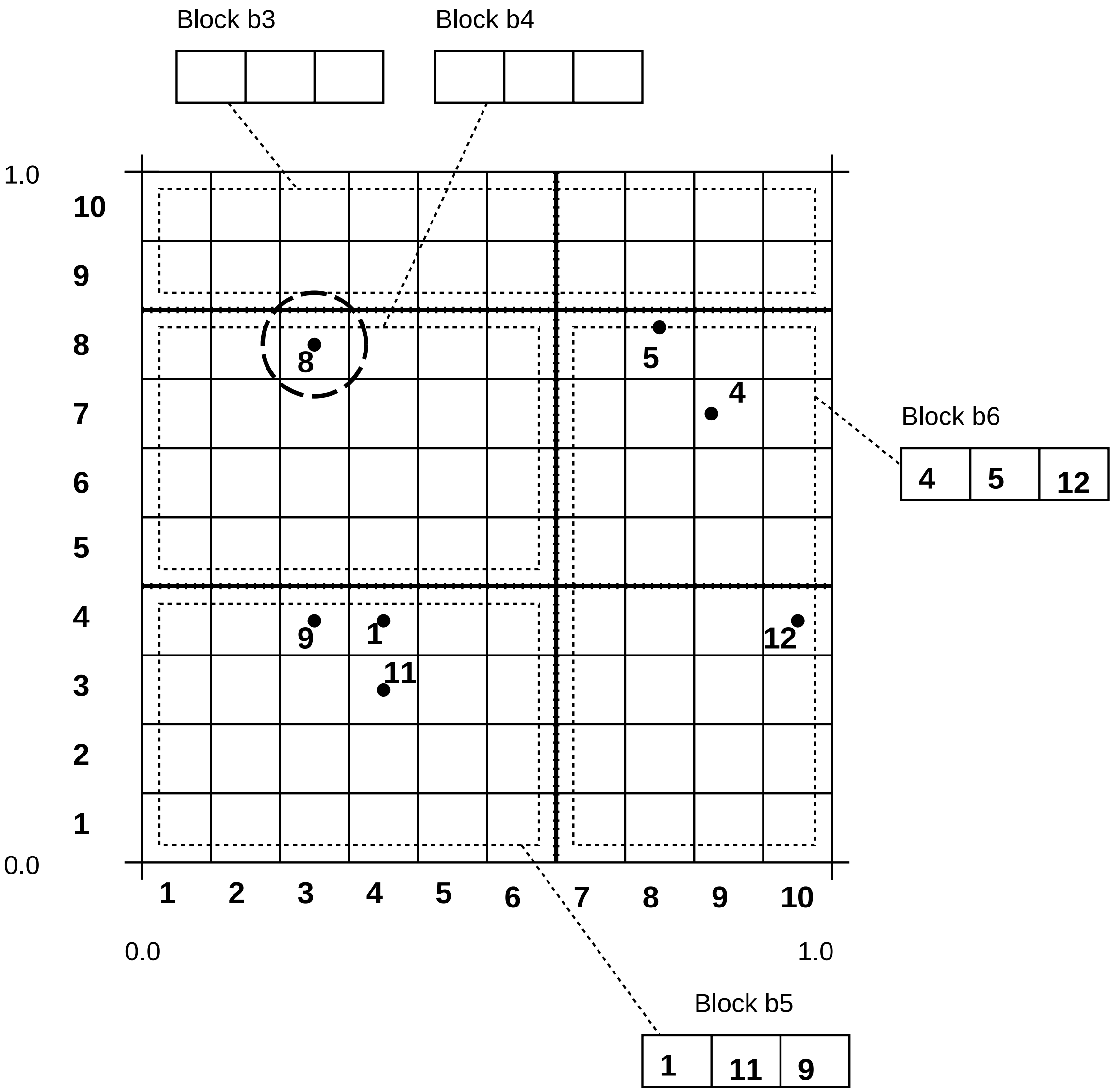
- Delete $p_{15} = (0.03, 0.24)$:
  - Locate cell
    $\Rightarrow$ Cell [1,3].
  - Retrieve block
    $\Rightarrow$ block b_7.
  - Delete from block
    $\Rightarrow$ no underflow.



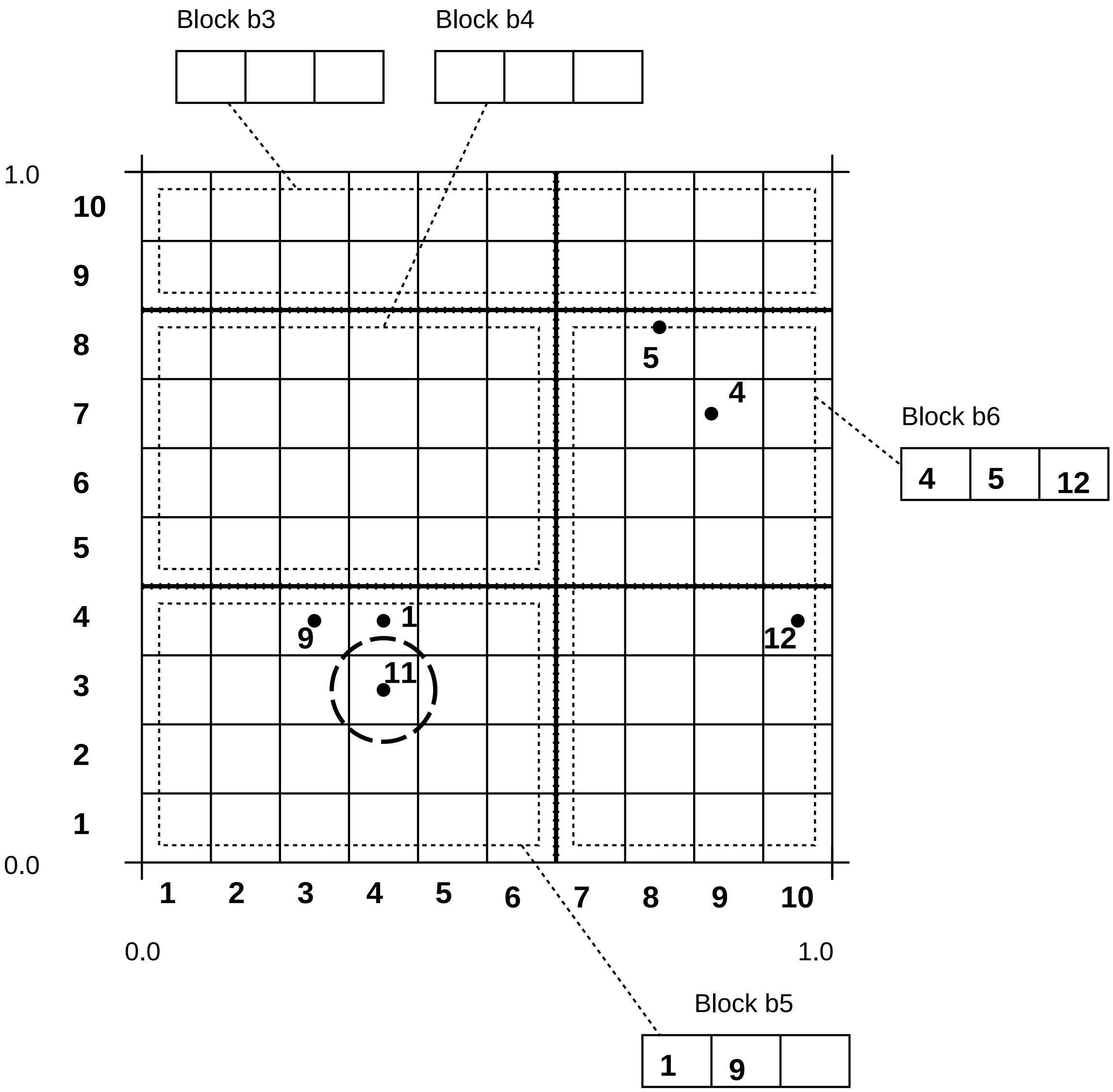
- Delete $p_{10} = (0.25, 0.24)$:
  - Locate cell
    $\Rightarrow$ Cell [3,3].
  - Retrieve block
    $\Rightarrow$ block b_7.
  - Delete from block
    $\Rightarrow$ causes underflow.
  - Can only merge with block b_5.
  - Remove partition after column 6.



- Delete $p_8 = (0.24, 0.77)$:
  - Locate cell
    ⇒ Cell [4,8].
  - Retrieve block
    ⇒ block b_4.
  - Delete from block
    ⇒ underflow.
  - Can only merge with block b_5, but b_5 is full
    ⇒ retain empty block.

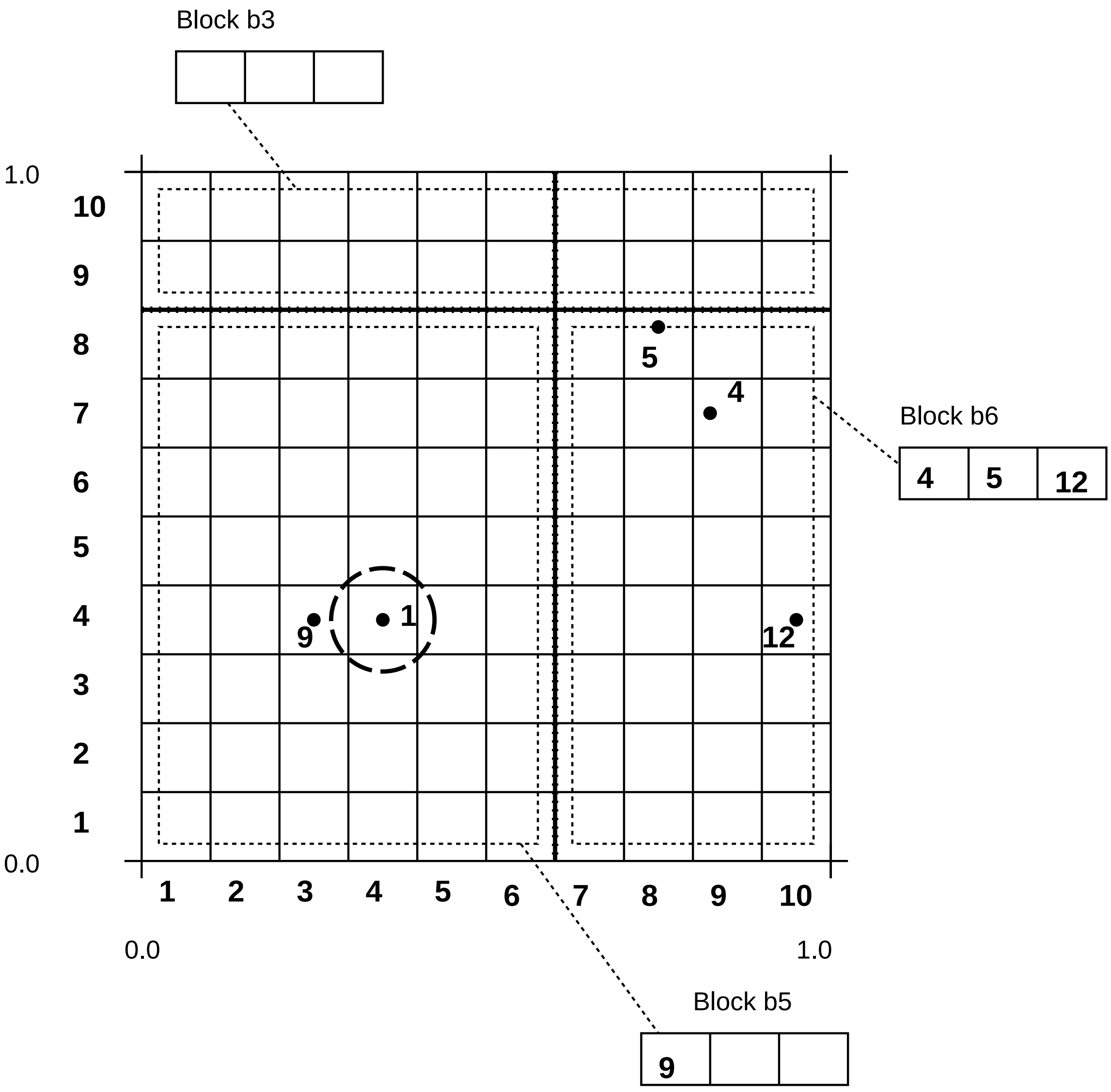


- Delete $p_{11} = (0.31, 0.24)$:
  - Locate cell
    ⇒ Cell [4,3].
  - Retrieve block
    ⇒ block b_5.
  - Delete from block
    ⇒ no underflow.



- Delete $p_1 = (0.35, 0.37)$:
  - Locate cell
    ⇒ Cell [4,4].
  - Retrieve block
    ⇒ block b_5.
  - Delete from block
    ⇒ causes underflow.

  - Merge with b_4.
  - Remove horizontal partition after row 4.
  - Return b_4 to free space.

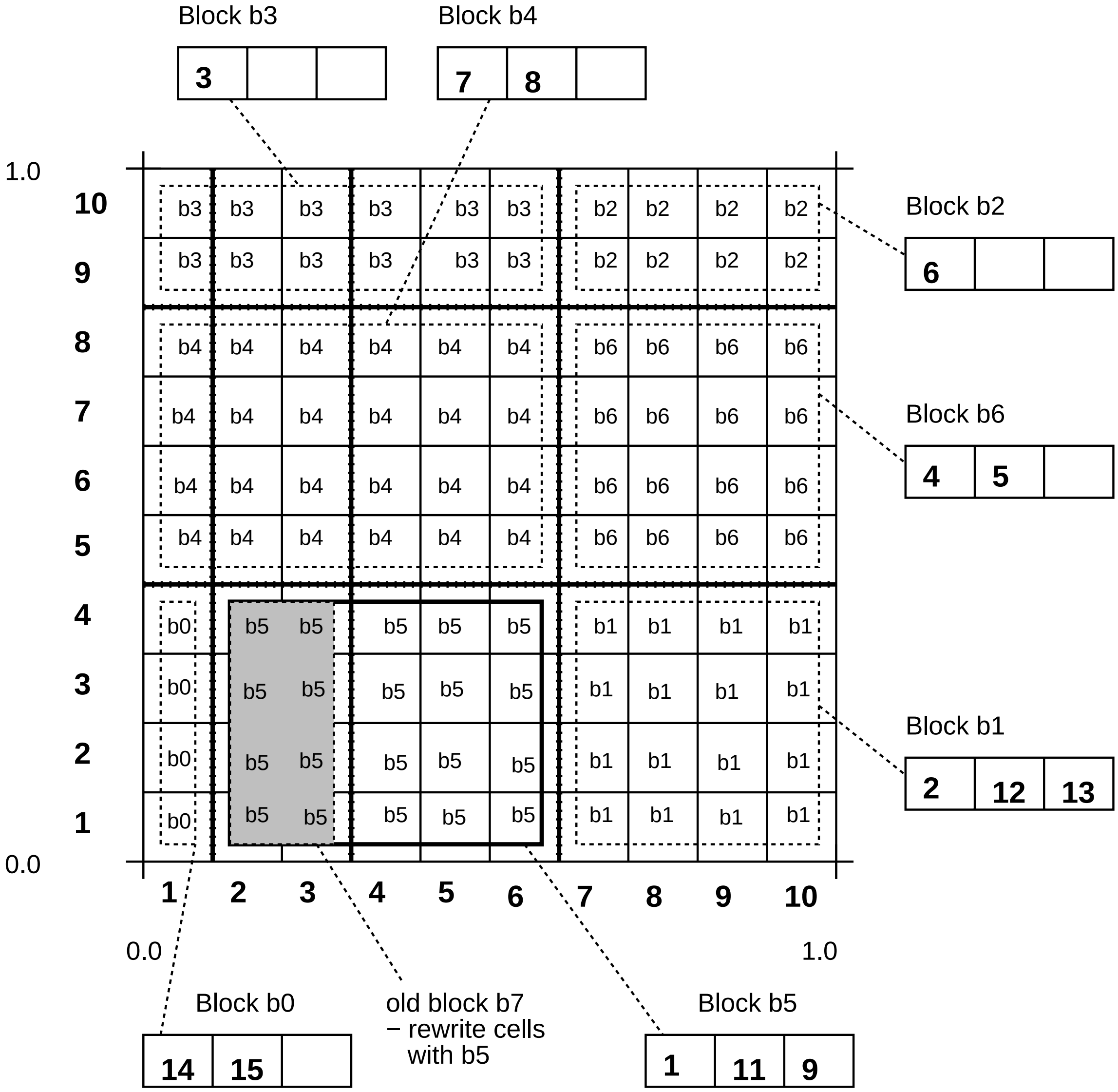


- Implementation details: deletion
  - There are two significant aspects to implementing deletion:
    1. Merging blocks.

2. Shrinking the grid directory.

These aspects differ in the two versions (fixed-size cells and variable-size cells).

- Fixed-size cell version:
  - Merging two blocks requires overwriting the cells of one of the blocks.
  - Example: merge blocks b_7 and b_5 below

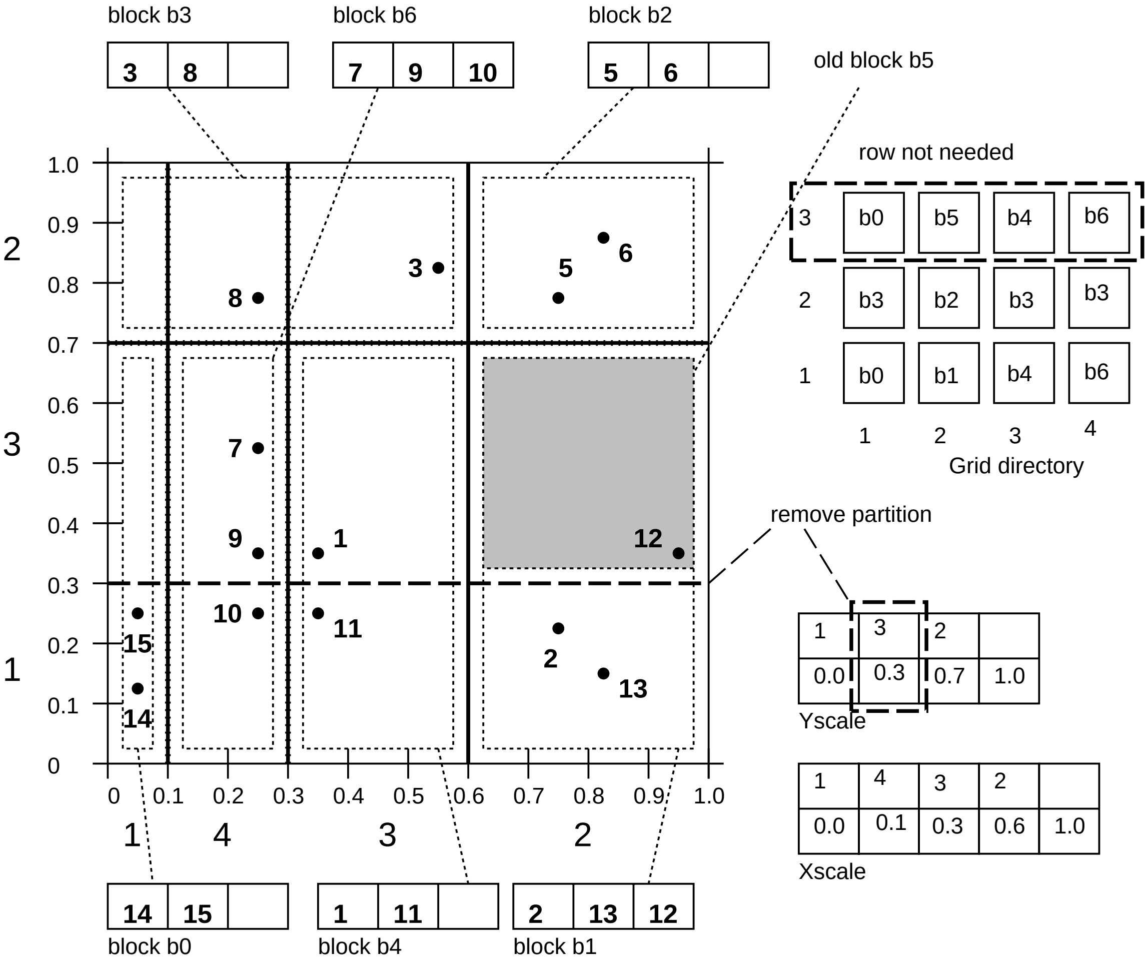


	1	2	3	4	5	6	7	8	9	10
10	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
9	b3	b3	b3	b3	b3	b3	b2	b2	b2	b2
8	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
7	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
6	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
5	b4	b4	b4	b4	b4	b4	b6	b6	b6	b6
4	b0	b5	b5	b5	b5	b5	b1	b1	b1	b1
3	b0	b5	b5	b5	b5	b5	b1	b1	b1	b1
2	b0	b5	b5	b5	b5	b5	b1	b1	b1	b1
1	b0	b5	b5	b5	b5	b5	b1	b1	b1	b1

- To remove a partition, we need to keep track of partitions
  ⇒ use array or linked-list.
- If the current set of partitions allow a coarser directory
  ⇒ grid directory can be coarsened.
- Coarsening the directory is an expensive operation
  ⇒ similar to refinement.

• Variable-size version:

- Merging requires adjusting elements in the grid directory.
- Example: merge b_1 and b_5



3	b0	b5	b4	b6
2	b3	b2	b3	b3
1	b0	b1	b4	b6
	1	2	3	4

Grid directory

1	3	2	
0.0	0.3	0.7	1.0

Yscale

1	4	3	2	
0.0	0.1	0.3	0.6	1.0

Xscale

- When a partition is removed, the grid directory can be reduced
  ⇒ expensive operation.

- Since collapsing the grid directory is expensive, it is often left in expanded form.

11.15 Grid Files: Summary

- The grid file is an index structure designed to support multikey queries.
- A grid file consists of:
  - A grid directory.
  - Grid scales (partitions).
  - Data blocks.
  - Meta-data such as the interval sizes, grid directory size etc.
- A grid file is a conceptual structure: several implementations are possible.

  We have seen two:
  - A grid file using fixed-size cells.
  - A grid file using variable-size cells.
- Several varieties of insertion are possible:
  - We alternated between vertical and horizontal partitions.
  - It might be better to use more vertical partitions if the data tends to be horizontally distributed.
- Likewise, several deletion strategies are possible.

  Shrinking the directory during deletion is an expensive operation
  ⇒ not worth implementing.

  If deletions are not common, lazy deletion is a good option.
- For general data, grid files are useful for range searches.
- For geometric point data, grid files are also useful for nearest-neighbor queries.

Chapter 12

Geometric Query Processing

Course Notes on Database Systems

12.1 Geometric Query Processing: Introduction

- Query processing in relational databases:
  - Query trees are transformed using relational algebra properties.
  - Disk I/O is the chief metric.
  - Several methods for efficiently implementing relational operators.
- Query processing in a geometric database:
  - Query processing involves implementing geometric "operators":
    - *Topological operators*: e.g., containment, overlap (intersection), disjunction
    - *Directional operators*: e.g., southwest, above, below.
    - *Search operators*: e.g., range queries, nearest neighbor queries, spatial joins.
  - Often, CPU costs are quite high
⇒ I/O is not the only cost consideration.
  - Currently, geometric query processing is only in the (early) research stages.
- We will focus on fast algorithms for certain well-defined in-memory geometric computations
⇒ problems in computational geometry.
- Examples of problems:
  - *Point location (polygon)*: given a query point q and a polygon P, is $q \in P$?
  - *Point location (polygonal map)*: given a query point q and polygonal map (region) M, which polygon in M contains q?

  - *Range search*: given a query rectangle R and a collection of points S, which points lie in R?
  - *Diameter*: given a collection of points S, find two that are farthest apart.
  - *Nearest neighbor (point):* given a collection of points S and a query point q, what is the nearest point to q in S?
  - *Nearest neighbor (set)*: given a collection of points S, find the nearest neighbor of each point.
  - *Intersection (pair)*: find whether two given geometric objects intersect.
  - *Intersection (set)*: given a collection of geometric objects find all pairs that intersect.

- Many of these problems come in two versions:

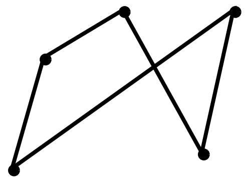
  1. *Single-shot*: only a single query is given.
  2. *Repetitive-mode*: several queries are to be handled.

  For example, consider the *point location (polygon)* problem: given a polygon P determine whether a given query point is inside P.

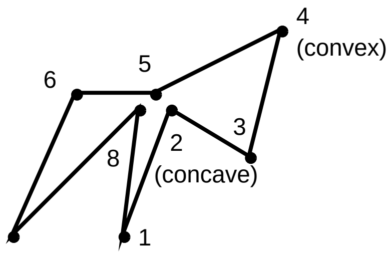
  1. *Single-shot* version:
     - Given a *single* query point q and a polygon P consisting of n points, is $q \in P$?
     - Here, options are limited
       $\Rightarrow$ must process all points in P
       $\Rightarrow$ computation is at least $O(n)$ (assume P has n points).
     - Main question of interest: is there an algorithm that performs no worse than $O(n)$?
  2. *Repetitive-mode* version:
     - Given a polygon and many query points $q_1, \ldots, q_m$, determine which points are in P?

- Here, we could treat each query point separately using the single-shot algorithm $O(n)$ times
  $\Rightarrow O(mn)$ for all the points.
- On the other hand, we can *preprocess* the polygon P (at cost $O(f(n))$).
- After preprocessing, cost of answering a point query is $O(g(n))$ per point
  $\Rightarrow$ cost for all points is $O(mg(n))$.
- Example: $f(n) = n \log n$ and $g(n) = \log n$
  $\Rightarrow O(\log n)$ per point query.
- Usually, there is also a storage cost to consider.

- Some useful definitions:
  - We will consider the standard Euclidean plane $\mathbb{R}^2$.
  - A polygon is a collection of n vertices $v_1, \ldots, v_n$ and n (directed) edges $e_1, \ldots, e_n$ where
    1. $e_i = v_i v_{i+1}$, $i = 1, \ldots, n-1$.
    2. $e_n = v_n v_1$.
  - A *simple polygon* is a polygon in which only successive edges intersect:
    $\Rightarrow e_i \cap e_j = \emptyset$ if $j \neq i+1$ or if $i = n, j \neq 1$.



A polygon (not simple)



A simple polygon

  - Assumption: we will assume that all polygons henceforth are simple.
  - Convention: vertices are numbered in counter-clockwise order.

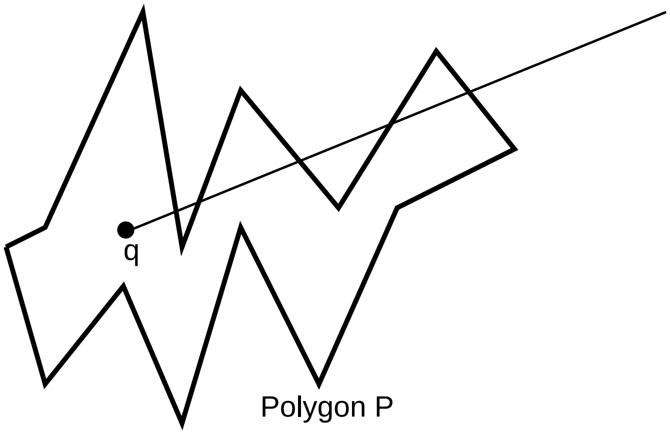
- A vertex is *convex* if its internal angle is no larger than 180°.
- A vertex is *concave* if its internal angle is larger than 180°.
- A *convex* polygon is a polygon with no concave vertices.
- ∂P denotes the boundary of polygon P.
- For a point q, we will use either $q = (x_0, y_0)$ or $(q.x, q.y)$ to denote the individual coordinates.

- Example of implementation:

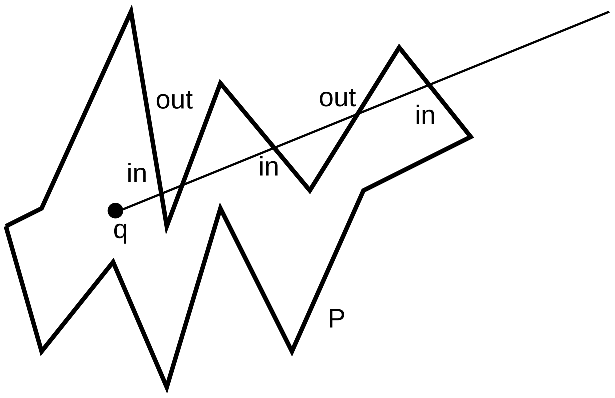
```
typedef struct point {
  double x;                // x-coordinate
  double y;                // y-coordinate
} point;

typedef struct polygon {
  long n;                  // Number of vertices
  point *v;                // Coordinates of vertices
} polygon;

...

polygon P;

P->v = (point *) malloc (sizeof(point) * (4));
P->n = 3;
P->v[1].x = 0.0;   P->v[1].y = 0.0;
P->v[2].x = 1.5;   P->v[2].y = 3.5;
P->v[3].x = 0.5;   P->v[3].y = 4.5;    // A triangle.
```

12.2 The Point Location Problem: Introduction

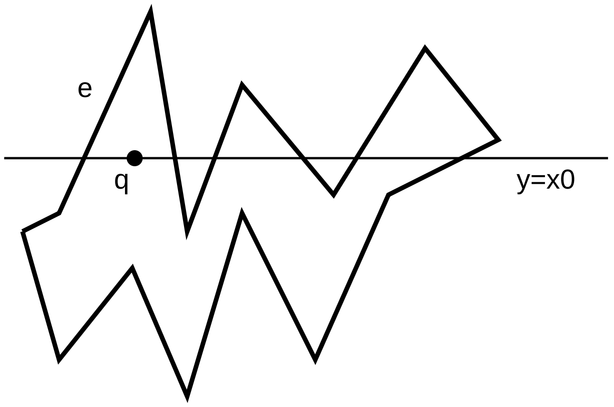
- We will consider the polygon version of the problem (the polygonal map version is not much more complex).
- Single-shot version: given a query point q and a polygon P, is $q \in P$?
- The *Ray-Crossings* method:



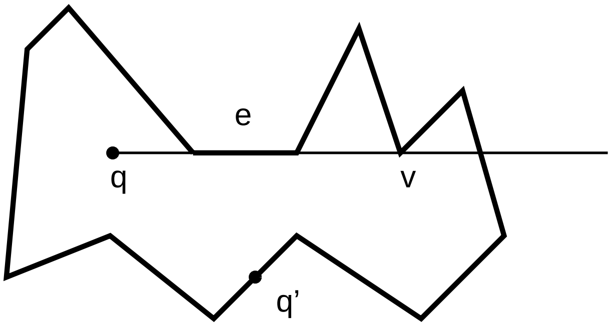
  - Draw a line (ray) from q in any one direction.
  - Count the number of intersections with edges of P.
  - If number of intersections is odd
    $\Rightarrow q \in P$
    else
    $\Rightarrow q \notin P$.
  - Why does this work?
    - Picture walking along the ray from infinity towards q.
    - When you cross an edge for the first time, you are “in”.
    - Next time you cross an edge you are “out”.
    - The next time, you are “in”, etc.



- Details:
  - Suppose the point q has coordinates $q = (x_0, y_0)$.
  - Which ray emanating from q do we use?
    $\Rightarrow$ use $y = x_0$ (horizontal line) going right of q (to $+\infty$).
  - How do we know which edges of P intersect with $y = x_0$?
    $\Rightarrow$ must test each edge against the line $y = x_0$.
  - What if intersection point lies to the left of q? e.g., intersection with edge e is to the left of q:



    $\Rightarrow$ must make sure intersection point is to the right of q.
  - What about degenerate cases?
    - ∗ ray passes through a vertex;
    - ∗ ray passes through an edge;
    - ∗ query point is on an edge.
    - ∗ query point is on an vertex.



Here, the ray from q passes through edge e and vertex v. Point q' lies on ∂P.

- Convention: $q \in P$ if $q \in \partial P$.
- If q is collinear with an edge but not contained in it
  $\Rightarrow$ ignore edge (in counting crossings).
- If ray passes through a vertex
  $\Rightarrow$ count as 2 crossings (each edge counts).

- To summarize: for each edge, make sure that at least one point is strictly above (below) while the other point is on or below (above).

- Pseudocode:

```
Algorithm: Ray-Crossings ($q$, $P$)

Input: query point $q$, polygon $P$.
Output: true, if $q \in P$; false, otherwise.
  num_crossings := 0;
  for each edge $e = (u, v)$ in $P$
    if $e$ contains $q$
      return true;
    if ($u.y > q.y$ and $v.y \leq q.y$)
        or ($u.y \leq q.y$ and $v.y > q.y$)
      // Compute intersection point with ray $y = q.x$
      $c$ := Intersection-Point ($q$, $e$);
      if $c.x > q.x$
        num_crossings := num_crossings + 1;
    endif
  endfor
  if num_crossings is odd
    return true;
  else
    return false;
```

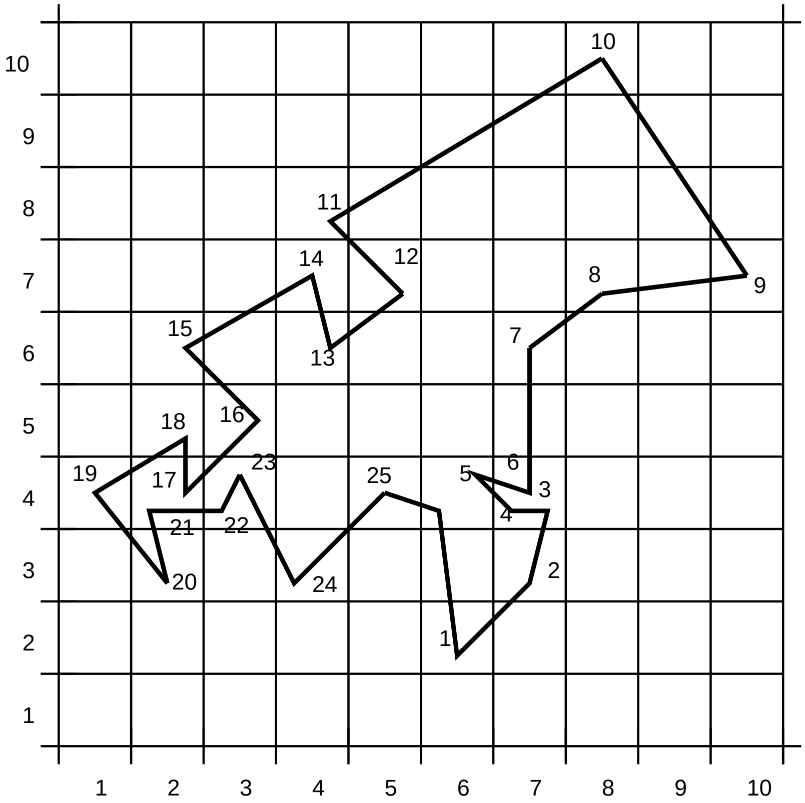
- Algorithm complexity:
  - Each edge is tested against the ray once.
  - Each intersection computation is $O(1)$
    ⇒ $O(n)$ overall.

- The ray-crossings method is suited to the single-shot type of query.

  Consider the repetitive mode, e.g., 5000 queries on a 10,000-point polygon
  ⇒ 5×10^7 intersection computations.

12.3 Point Location Using 2D-Hashing

- For repetitive-mode queries, it is often better to organize the edges in the polygon in some way
  ⇒ reduce the number of tests per point.
- Key ideas in 2D-hashing:
  - Consider an imaginary grid superimposed on the given polygon:

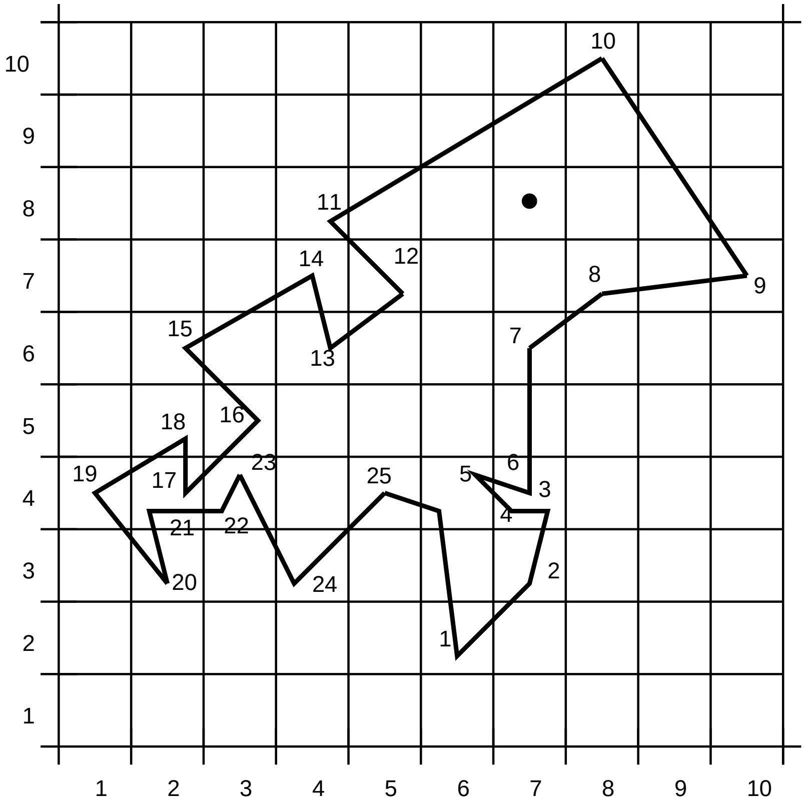


  - For each cell (bucket), identify the edges that intersect the cell, e.g.,

Cell [5,7]:	edges 11, 12.
Cell [3,4]:	edges 16, 21, 22, 23.

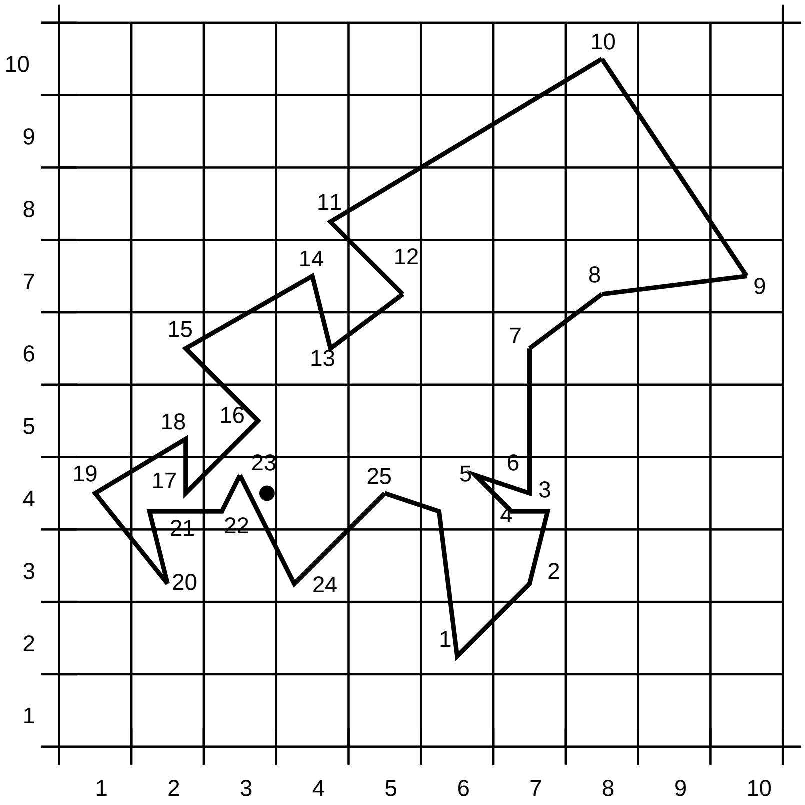
  - For each cell, store the *list* of edges that intersect the cell
    ⇒ the *cell-list* for each cell.

- For empty cells, identify whether the cell is inside the polygon or not.
- Given a query point q:
  - $*$ Identify the cell containing the point.
  - $*$ If cell is empty, determine location of point using location of cell (inside or outside).
  - $*$ Otherwise, test query point against all edges in cell-list.
  - $*$ Find closest edge and use it to determine whether $q \in P$.
- Example: $q = (7.5, 8.5)$:



  - $*$ q lies in Cell [7,8].
  - $*$ Cell [7,8] is empty.
  - $*$ Cell [7,8] is entirely contained in P
    $\Rightarrow$ q is contained in P.

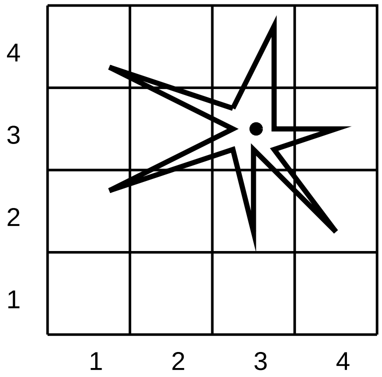
- Example: $q = (3.75, 4.5)$:



  - ∗ q lies in Cell [3,4].
  - ∗ Cell [3,4] is not empty.
  - ∗ Cell [3,4] has edges 16, 21, 22 and 23.
  - ∗ Edge 23 is closest.
  - ∗ q is to the left of edge 23
    $\Rightarrow q \in P$.

- Preprocessing cost:
  - Suppose we use a $k \times k$ grid.
  - Each polygon edge intersects at most $O(k)$ cells.
    $\Rightarrow O(k)$ insertions per edge
    $\Rightarrow O(nk)$ insertions overall.

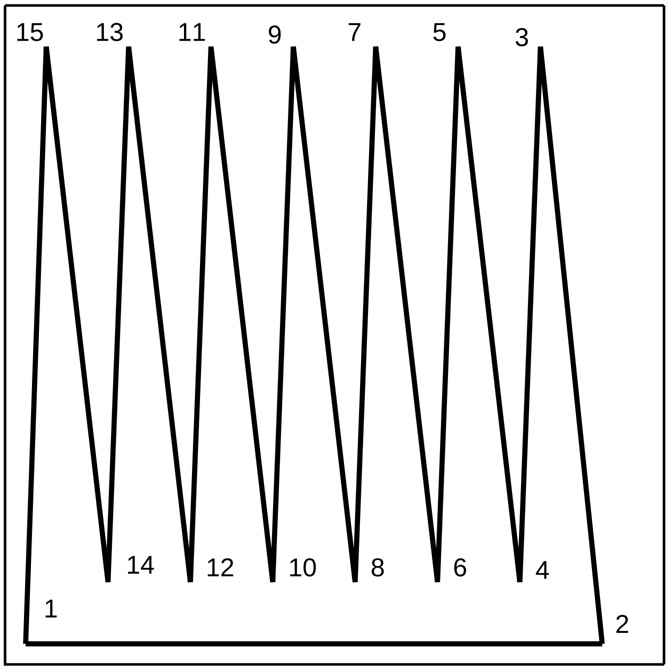
- Storage cost:
  - $O(k^2)$ for the cells.
  - $O(nk)$ for the edges in the cell-lists.
- Query cost per point query:
  - Worst-case, all edges hash to a single cell:



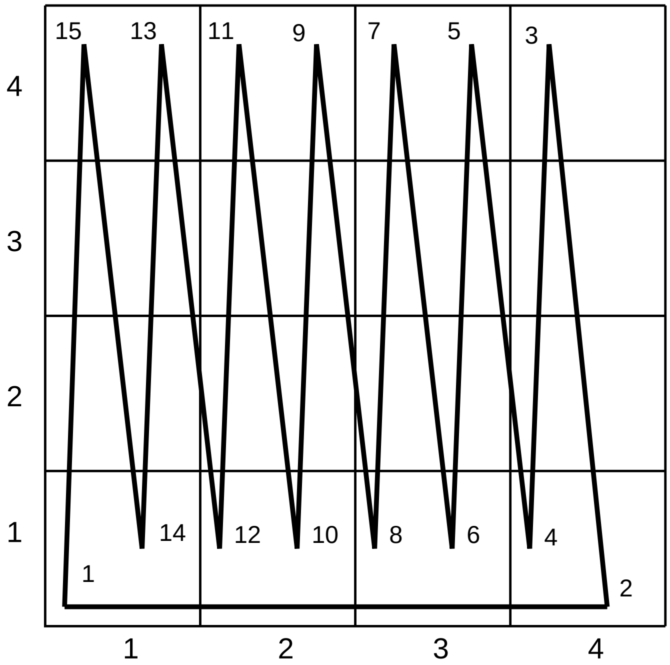
    Here, Cell [3,3] has all edges.
    $\Rightarrow q$ must be tested against all edges
    $\Rightarrow O(n)$ worst-case.

  - If the edges are somewhat uniformly distributed, then hopefully each cell has approximately $O(\frac{n}{k^2})$ edges.
- Choosing k:
  - To make $O(\frac{n}{k^2}) = O(1)$, we need $k = O(\sqrt{n})$.
  - Preprocessing cost: $O(n\sqrt{n})$.
  - Storage cost: $O(n)$.

- Sometimes it is better to use different partitions along x and y axes:
  - Consider this example:

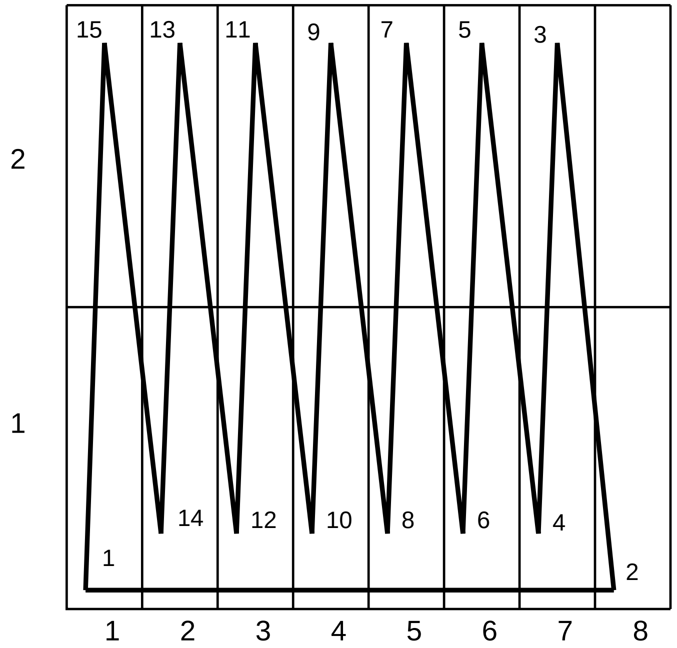


    - $n = 15$ edges.
    - $k = \sqrt{n} \approx 4$
      $\Rightarrow$ a 4×4 grid:



Each cell as at least 4 edges
$\Rightarrow O(\sqrt{n})$ edges per cell
$\Rightarrow$ query cost is $O(\sqrt{n})$.

- Instead, suppose we use different x and y partitions:
  - Let k_x = number of x divisions.
  - Let k_y = number of y divisions.
  - Example: take $k_x = 8$ and $k_y = 2$ (in example above):



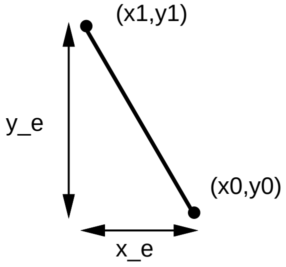
- $*$ 2 edges per cell in most cells ⇒ better!

– Intuition:

- $*$ Consider walking along ∂P.
- $*$ You tend to walk a lot along the y-direction
- $*$ Thus, to separate out the edges thin long slabs are needed ⇒ more divisions along x direction.

– A heuristic for choosing k_x and k_y:

- $*$ Consider an edge e with endpoints (x_0, y_0) and (x_1, y_1):



- $*$ Define the projections:

$$
\begin{aligned}
x_e &\triangleq |x_1 - x_0| \\
y_e &\triangleq |y_1 - y_0|
\end{aligned}
$$

- Define the total horizontal (projected) length:

$$D_x \triangleq \sum_e x_e$$

- Define the total vertical (projected) length:

$$D_y \triangleq \sum_e y_e$$

- Let L_x and L_y be the span of the polygon in the x and y directions, respectively.
- Thus, there will be more "y-travel" per unit-length if

$$\frac{D_y}{L_y} > \frac{D_x}{L_x}$$

⇒ we want more x-partitions if

$$\frac{D_y}{L_y} > \frac{D_x}{L_x}$$

⇒ take

$$\frac{k_x}{k_y} \propto \frac{D_y/L_y}{D_x/L_x}.$$

- But we still want $k_x k_y = O(n)$.
- Let α be a parameter associated with the algorithm.
- Then, pick k_x and k_y such that

$$k_x k_y = \alpha n$$

and

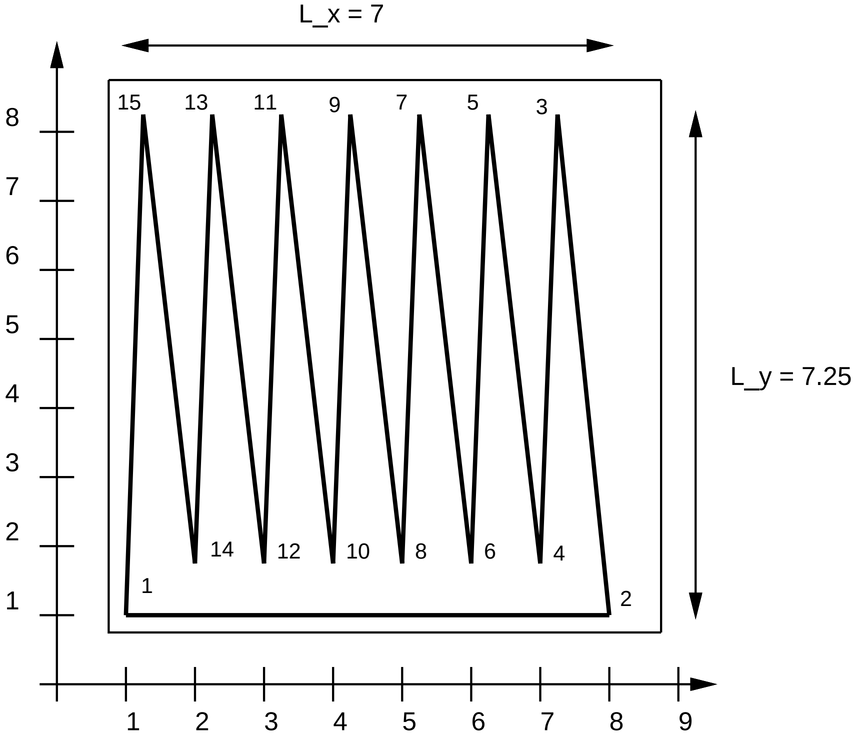
$$\frac{k_x}{k_y} \propto \frac{D_y/L_y}{D_x/L_x}.$$

∗ This gives

$$k_x = \left\{\alpha n \frac{L_x D_y}{L_y D_x}\right\}^{\frac{1}{2}}$$

$$k_y = \left\{\alpha n \frac{L_y D_x}{L_x D_y}\right\}^{\frac{1}{2}}$$

∗ Previous example (with $\alpha = 1.0$):



· $L_x = 7.0$ and $L_y = 7.25$.

· To compute D_x:
edge 1 has length 7
7 odd-numbered edges have length 0.25
$\Rightarrow 1.75$
7 even-numbered edges have length 0.75
$\Rightarrow 5.25$
$\Rightarrow D_x = 7.0 + 1.75 + 5.25 = 14.00.$

· To compute D_y:
edge 1 has length 0

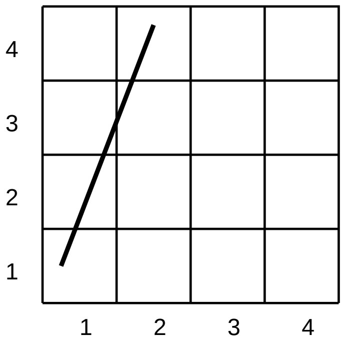
edges 2 and 15 have length 7.25
$\Rightarrow$ 14.5
12 remaining edges have length 6.5 each
$\Rightarrow$ 78
$\Rightarrow D_y = 0 + 14.5 + 78 = 92.5.$

· This gives

$$\begin{aligned} k_x &= \left\{15 \times \frac{7.0 \times 92.5}{7.25 \times 14.0}\right\}^{\frac{1}{2}} = 9.78 \\ k_y &= \left\{15 \times \frac{7.25 \times 14.0}{7.0 \times 92.5}\right\}^{\frac{1}{2}} = 1.53 \end{aligned}$$

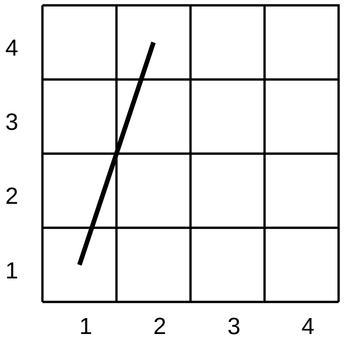
· Thus, $k_x = 10$ and $k_y = 2$ are appropriate choices.

- Some implementation details:
  - Use a 2D-array for the buckets (cells).
  - Use a linked list to store the edges associated with each bucket.
  - To create the hashtable:
    - Find the minimum and maximum x values among all vertices.
    - Find the minimum and maximum y values among all vertices.
    - Create the boundary points for the grid
      $\Rightarrow$ essentially the MBR of the polygon.
  - Process edges one by one.
  - For each edge:
    - Find out which cells are traversed by the edge.
    - Add the edge to each such cell (bucket).
    - For example:



Edge traverses cells [1,1], [1,2], [1,3], [2,3] and [2,4].

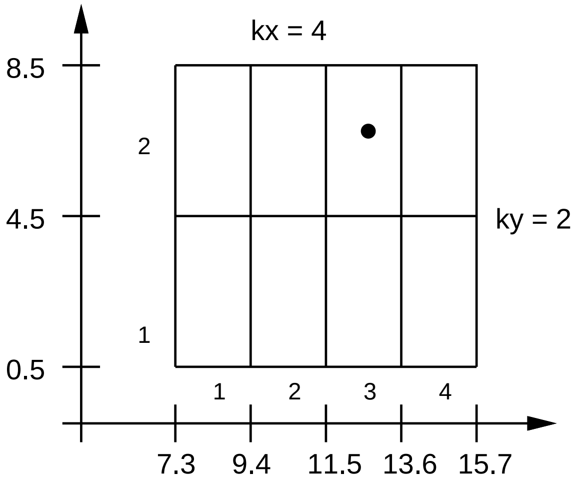
- What about edges that pass through a cell crosspoint?



Insert in the edge lists of all adjoining cells.
Above, insert edge into cells [1,1], [1,2], [1,3], [2,2], [2,3] and [2,4].

– To process a query point:

- ∗ Find the cell containing the query point.
- ∗ Example:



In this example ($q = (12.6, 6.4)$):

- · The grid boundaries are [7.3,15.7] (x-axis) and [0.5,8.5] (y-axis).
- · $k_x = 4$
  ⇒ x-length of cell is 2.1.
- · $k_y = 2$
  ⇒ y-length of cell is 4.0.
- · 4 x-intervals: [7.3,9.4], [9.4,11.5], [11.5,13.6], [13.6,15.7].
- · 2 y-intervals: [0.5,4.5], [4.5,8.5].
- · To get x-interval: divided by interval length

$$\frac{12.6 - 7.3}{2.1} = 2.52$$

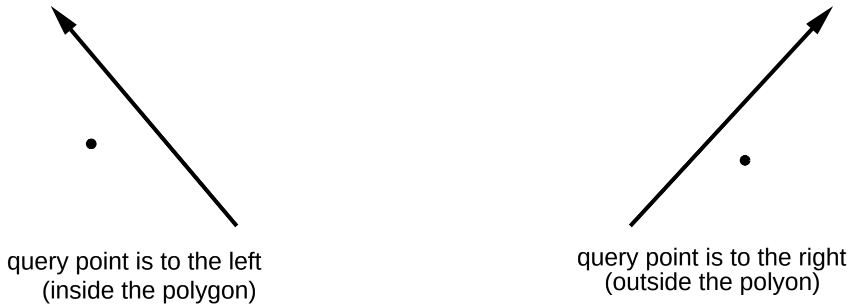
⇒ 3rd column.

- · Similarly, the y-interval is given by

$$\frac{6.4 - 0.5}{4.0} = 1.475$$

⇒ 2nd row.

    - Once the correct bucket (cell) is identified, there are two cases to consider.
    - CASE 1: The bucket has edges
      - Find the closest edge to point.
      - Recall: each edge is directed.
      - Find out which side of the edge (left or right) the point lies.
      - If point is to the left, it is inside the polygon. Otherwise, it is outside.
      - Why does this work?



      Recall: the points are in counter-clockwise order.

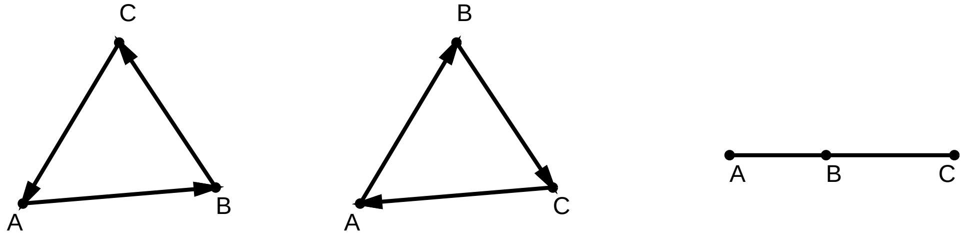
    - CASE 2: the bucket has no edges
      - In this case, the bucket is either completely inside or completely outside the polygon.
      - Use this to determine the location of the point.

    Note: locations of empty buckets are identified during preprocessing.

- Additional implementation details:
  - How do we know whether a point is to the *left* of a directed edge?
  - First, consider finding the area of a triangle:
    Given 3 points $A = (a_x, a_y), B = (b_x, b_y), C = (c_x, c_y)$, the area of triangle ABC is given by:

$$Area(A, B, C) = \frac{1}{2}(a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - b_y c_x)$$

– It turns out that if A, B and C are in counter-clockwise order, then $Area(A, B, C) > 0$. In general:



Area(A,B,C) > 0 Area(A,B,C) < 0 Area(A,B,C) = 0

– The area calculation can be used to see if point C is to the *left* of edge (A, B)
$\Rightarrow C$ is left of (A, B) if $Area(A, B, C) > 0$.

– Hence if we have a function `double area (point A, point B, point C)` we can write

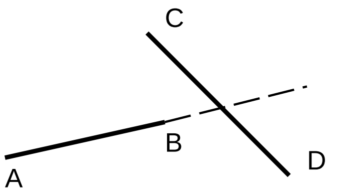
```
boolean left (point A, point B, point C)
{
  return area (A,B,C) > 0;
}

boolean lefton (point A, point B, point C)
{
  return area (A,B,C) >= 0;
}

boolean collinear (point A, point B, point C)
{
  return area (A,B,C) == 0;
}
```

– Note: using integer coordinates will give exact results.

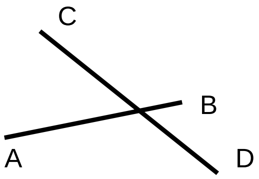
– A related issue: how do we check whether a point C is *on* the segment (A, B)?

    - See if C is collinear with A, B.
    - If so, check to see if C's x-value is between A's x-value and B's x-value.

- How does one check edge intersections?
    - It is not enough to simply use line equations and analytic geometry. For example:

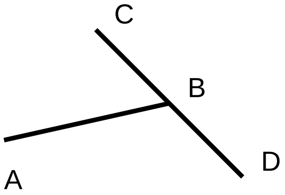


Equation of line AB intersects but the segment does not.



Segments intersect

    - For proper intersection:
        - Segment CD forces A and B on different sides of CD.
        - Segment AB forces C and D on different sides of AB.
    - Suppose we have a function called `area_sign()` that returns the sign of the area of a triangle.
    - Then, for proper intersection we want:
        1. `area_sign(A,B,C) * area_sign(A,B,D) < 0` and
        2. `area_sign(C,D,A) * area_sign(C,D,B) < 0`.
    - Note: the above conditions don't allow collinearity:



    Here, B is collinear with C, D.

    - Thus, to include collinearity of one endpoint as part of proper intersection, a separate test is needed.

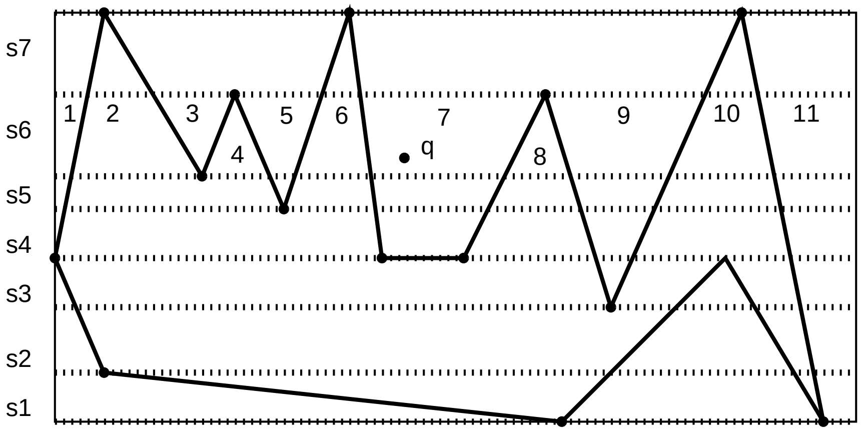
12.4 Point Location: The Slab Method

- One of the most effective unidimensional searching methods is *binary search*.

  Binary search requires organizing the data (sorting) to suit binary search.

- Main idea behind the slab method: organize the data (polygon vertices and edges) to facilitate binary search.

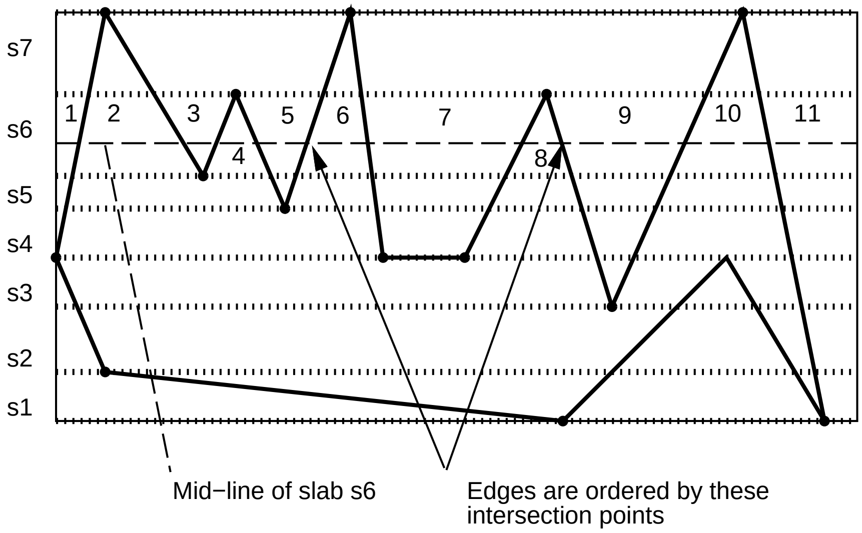
- Key ideas:
  - Construct the polgon's MBR.
  - Pass a horizontal line through each polygon vertex.



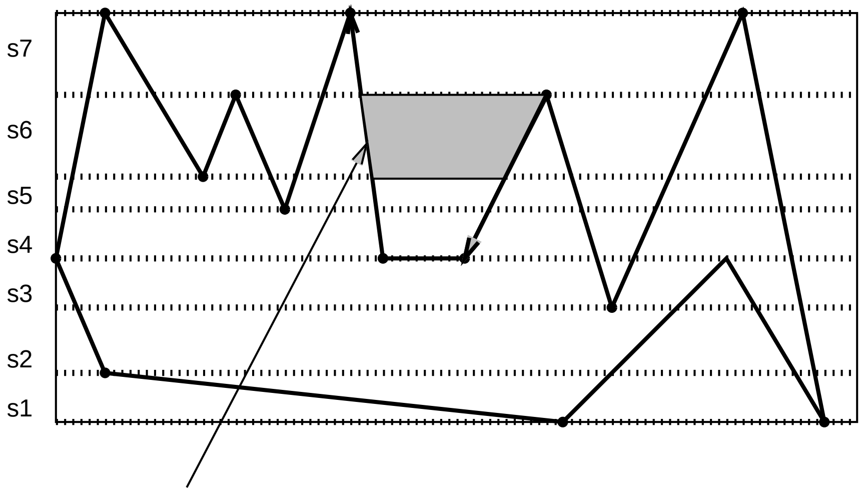
  - The lines partition the MBR into horizontal slabs.
  - Each slab is a collection of trapezoids ordered left to right.
  - The trapezoids are *horizontal trapezoids*: the two parallel edges are parallel to the x-axis.
  - Some trapezoids are triangles: one side has zero length.
  - Each trapezoid is either completely inside or completely outside the polygon.
  - Suppose we have identified the status (inside or outside) of each trapezoid.

- To process a point query $q = (x_0, y_0)$:
  - ∗ Identify the correct slab.
  - ∗ Search for the proper trapezoid within the slab.
  - ∗ Return status of trapezoid (inside or outside).

- Details:
  - Use binary search to identify the correct slab
    ⇒ slab intervals must be stored in an array.
  - To construct slab array, the vertices of P need to be sorted by y-value.
  - For each slab, we need to know which polygon edges cut through the slab:
    - ∗ Maintain an array of edges for each slab.
    - ∗ Process polygon edges one by one.
    - ∗ For each edge, check which slabs it intersects.
    - ∗ Add edge to that slab's array.
  - Edges within a slab have to be ordered left to right
    ⇒ sorting needed.
  - In sorting edges, we need to "compare" two edges. How?
    - ∗ While placing edges in a slab's array, find intersection point with mid-line of slab.

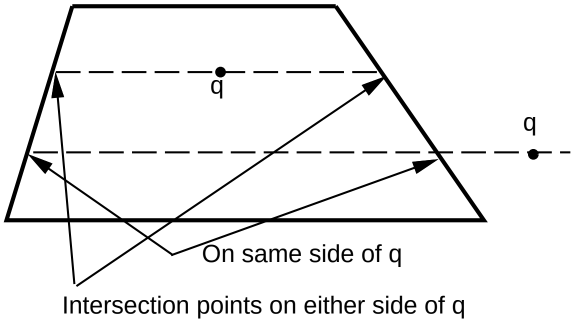


  - ∗ While processing edges in a slab, sort by these intersection points.
- – Use any non parallel edge to determine if a trapezoid is inside or outside.



Trapezoid is outside (left edge goes up)

- – For each trapezoid, store left and right edges.
- – To determine if a point is inside a particular trapezoid:
  - ∗ Check intersections of trapezoid edges with the line $y = y_0$.
  - ∗ If intersection points are on either side of point
    ⇒ point is inside.



- – Given a query point $q = (x_0, y_0)$:
  - ∗ Use binary search with y_0 to find correct slab.
  - ∗ Use binary search with x_0, y_0 to find correct trapezoid.
  - ∗ Return status of trapezoid.

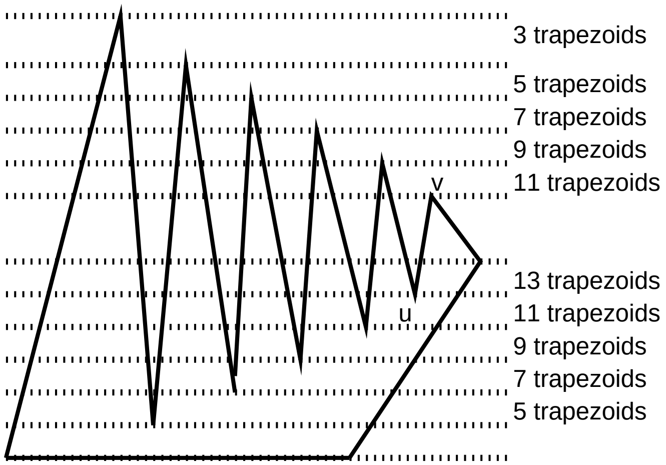
- Preprocessing time:
  - $O(n \log n)$ to sort vertices by y-value.
  - Each edge may pass through $O(n)$ slabs
    $\Rightarrow O(n^2)$ insertions into slab arrays.
  - Each slab may have $O(n)$ trapezoids
    $\Rightarrow O(n \log n)$ to sort within a slab
    $\Rightarrow O(n^2 \log n)$ to sort all slabs.
  - Identification for each trapezoid takes $O(1)$ time
    $\Rightarrow$ Overall, $O(n^2)$.

  Thus, preprocessing time is $O(n^2 \log n)$.

- Storage cost:
  - $O(n)$ slabs
    $\Rightarrow O(n)$ intervals.
  - Each slab may have $O(n)$ trapezoids
    $\Rightarrow O(n^2)$ for trapezoids.

  Thus, $O(n^2)$ storage is needed.

  Worst-case example:



By adding more zig-zags between u and v, additional slabs can be added. Total number of trapezoids is of the order

$$\begin{aligned}(\text{const})\ \ & (3+5+7+\ldots) \\ & = O(n^2)\end{aligned}$$

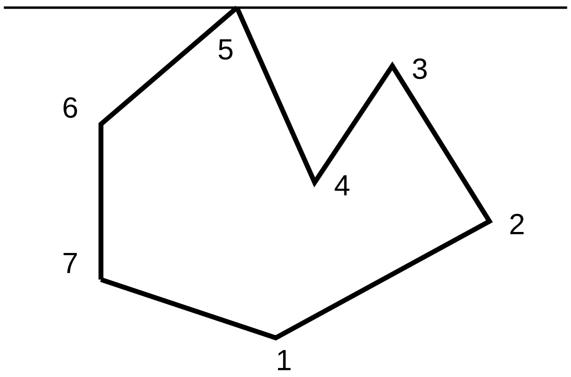
- Query cost:
  - $O(\log_2 n)$ for binary search among slabs.
  - $O(\log_2 n)$ for binary search within a slab.

  $\Rightarrow O(\log_2 n)$ overall.

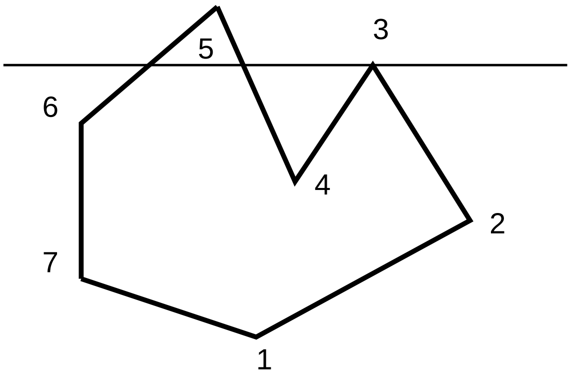
  Note: the constant is small: $2\lceil \log_2 n \rceil$ steps for search.
- Summary:
  - Preprocessing: $O(n^2 \log n)$.
  - Storage: $O(n^2)$.
  - Query: $O(\log n)$.
- Can we do better?
  - It's unlikely that query time can be improved (Improvement would result in faster unidimensional search).
  - Storage cost cannot be improved for this method.
  - What about preprocessing?
    - $*$ Consider two adjacent slabs.
    - $*$ Once we've sorted one slab (takes $O(n \log n)$ time), how much time does it take to sort the next one?
    - $*$ Currently, $O(n \log n)$.
    - $*$ However, how different are the two slabs?
    - $*$ If they are similar it should be possible to use that information to reduce sorting time.

12.5 Plane Sweep

- Plane sweep is a fundamental method in computational geometry
  ⇒ useful as part of several algorithms.
- For simplicity of presentation, we'll first consider polygons with unique vertex y-values
  ⇒ no two points lie on the same horizontal line.
- Key ideas:



  - Consider a horizontal line above the polygon.
  - Picture moving the line towards the lowest point of the polygon.
  - Stop the line (temporarily) at each vertex the line encounters.
  - Some processing occurs at each stop.
  - Example: above the line starts by sweeping down from vertex 5.

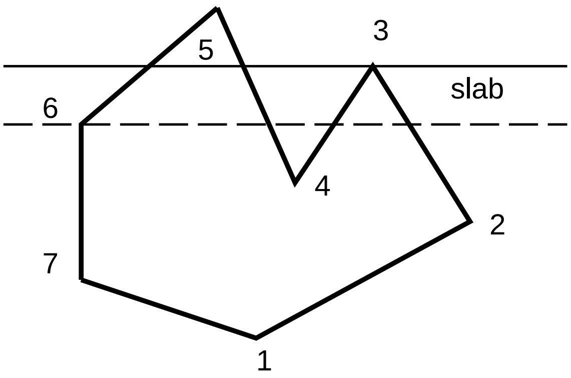


Next stop in sweep is vertex 3.
  ⇒ vertices are visited in the order: 5,3,6,4,2,7,1.

– Notation: let $v_{[i]}$ denote the vertex encountered in the i-th stop of the plane sweep.
  ⇒ $v_{[1]} = 5, v_{[2]} = 3, \ldots, v_{[7]} = 1$.

– Each stop is a slab boundary.

– At each stop: create the correct order of edges (left-to-right) in the slab immediately below.

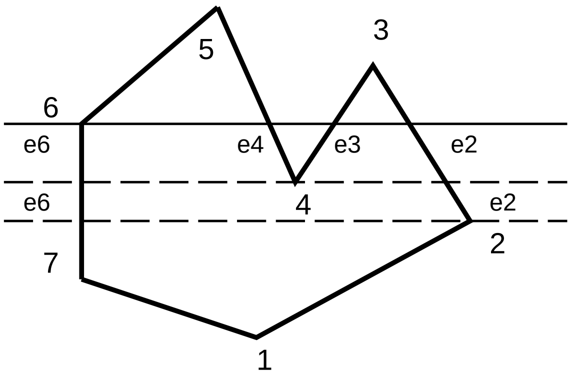


– Example: when the sweep is at vertex 3, the second slab is processed.

– Denote the ordered list of edges for slab i by $L_{[i]}$.
  ⇒ e.g., $L_{[2]} = e_5, e_4, e_3, e_2$.

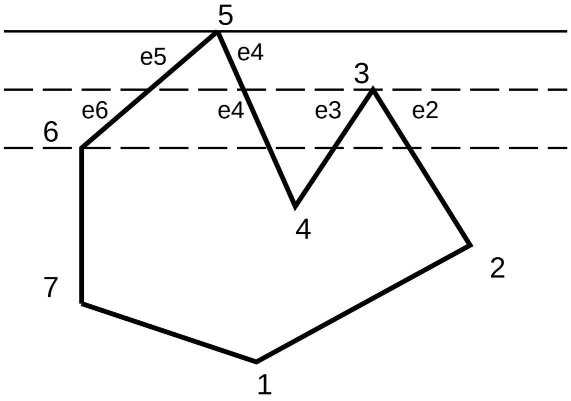
– The key observation:
  * Suppose we have created $L_{[i]}$ at the i-th stop in the sweep.
  * Next, we want to create $L_{[i+1]}$.

- Question: how different is $L_{[i+1]}$ from $L_{[i]}$?
- Answer: not much!



– Example:

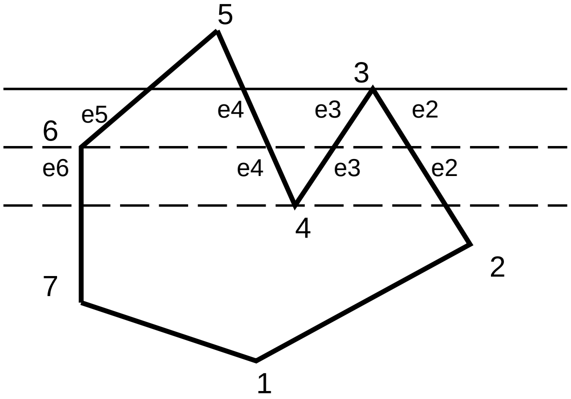
- $L_{[3]} = e_6, e_4, e_3, e_2$ (in that order).
- Now sweep down to $v_{[4]} = 4$.
- $L_{[4]} = e_6, e_2$.
- Observe: to get $L_{[4]}$ from $L_{[3]}$, remove e_4, e_3.
  $\Rightarrow e_4, e_3$ are the edges incident at $v_{[4]}$.
- Note: both e_4 and e_3 go "up".



– Example: Consider the sweep from $v_{[1]} = 5$ to $v_{[2]} = 3$.

- $L_{[1]} = e_5, e_4$.
- $L_{[2]} = e_5, e_4, e_3, e_2$.

- Here, we *added* e_3, e_2.
- e_3, e_2 are incident to $v_{[2]}$.
- Note: both e_3 and e_2 point "down".

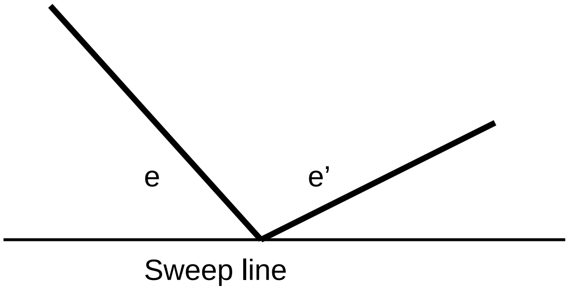


– Example: Consider the sweep from $v_{[2]} = 3$ to $v_{[3]} = 6$.

- $L_{[2]} = e_5, e_4, e_3, e_2$.
- $L_{[3]} = e_6, e_4, e_3, e_2$.
- Here, we deleted e_5 and added e_6.
- Note: e_6 points up, e_5 points down.

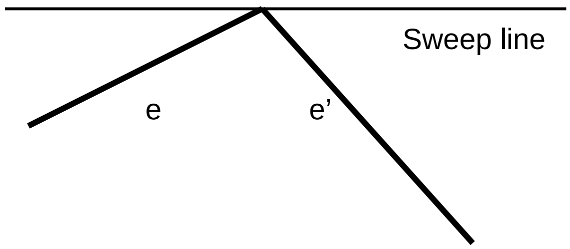
– Thus, at each step in the sweep, one of three cases occur:

1. Both incident edges are above the vertex:



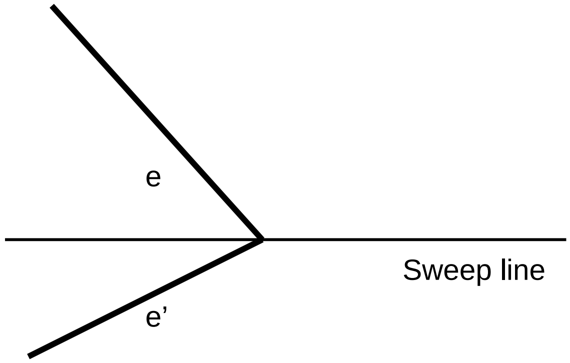
$\Rightarrow$ delete e, e' from current list.

2. Both incident edges are below the vertex:



⇒ add e, e' to current list.

3. One edge is above and one is below:



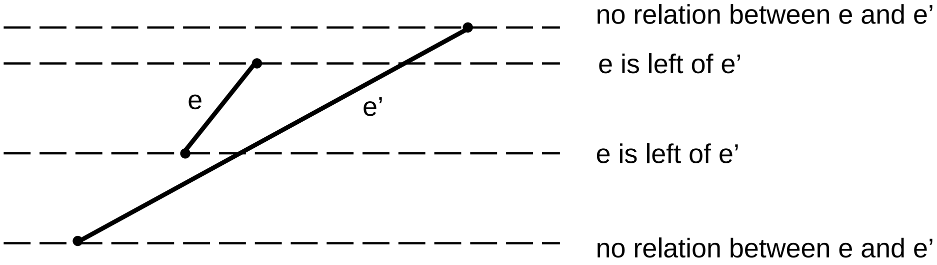
⇒ delete e, add e'.

- Note: when edges are added, they have to be added at the right place (to maintain left-to-right order).

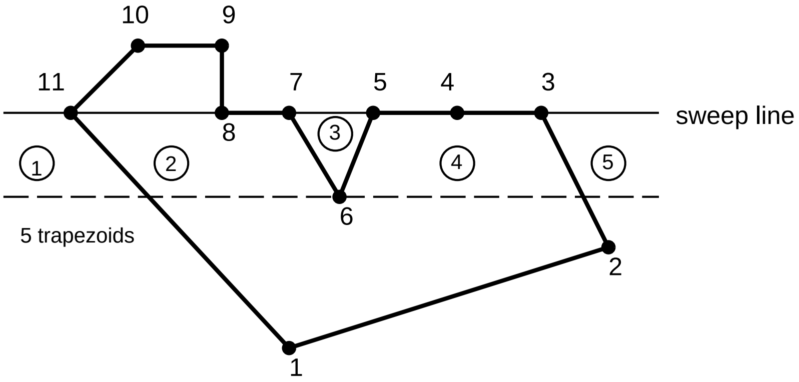
- Complexity:
  - At each step in the sweep, we need to find the right place in current list
    ⇒ $O(n)$ work at most (since list can have $O(n)$ elements).
  - Once we've found the right place, adding and deleting takes $O(1)$ time.
  - n vertices in sweep
    ⇒ $O(n^2)$ overall.
  - Thus, using plane sweep speeds up preprocessing for the Slab Method from $O(n^2 \log n)$ to $O(n^2)$.
  - Note:
    * By using a height-balancing tree to represent the current list, the plane sweep can be done in $O(n \log n)$ time.

  - ∗ Insertions and deletions cost $O(\log n)$ in a height-balanced tree.
  - ∗ Examples of height-balanced trees: AVL trees, red-black trees, 2-3 trees.
- – However, it does not help in the Slab Method, since $O(n^2)$ work is needed to build the slab arrays.
- – Implementation using a height-balanced tree is more complex than implementation with a simple list.

- Some details:
  - – How do we decide whether e is left of e'?



  - – To decide the relation between two edges
    ⇒ compare the x-values of the points of intersection with sweep line.
  - – Note: intersection computation is $O(1)$.
  - – Handling horizontally collinear points is more complex:

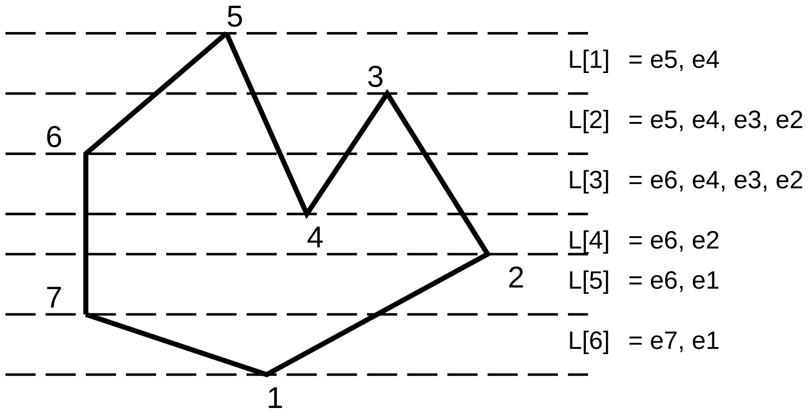


In the example above:

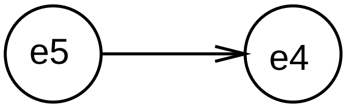
  - ∗ Each collinear point is handled separately.

    - ∗ Point v_{11}: delete e_{10}, add e_{11}.
    - ∗ Point v_8: delete e_8.
    - ∗ Point v_7: add e_6.
    - ∗ Point v_5: add e_5.
    - ∗ Point v_4: no action.
    - ∗ Point v_3: add e_2.
    - ∗ Observation: horizontal edges are ignored (they are not needed for the trapezoids).

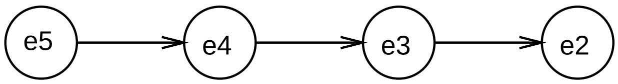
- Plane sweeps and topological sorts.
  - Suppose we use the notation (e_i, e_j) to denote e_i is left-of e_j.
  - Note that the *left-of* relation is a *partial order*.



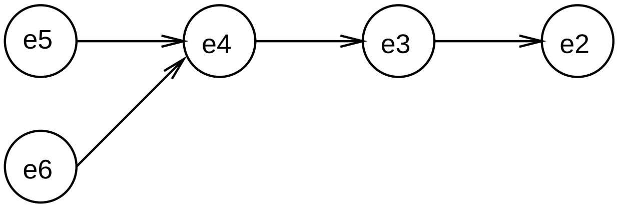
  - Example: for the polygon above
    - ∗ In creating $L_{[1]} = e_5, e_4$, the pair (e_5, e_4) is added to the relation:



    - ∗ In creating $L_{[2]} = e_5, e_4, e_3, e_2$, the pairs (e_3, e_2) and (e_4, e_3) are added to the relation:

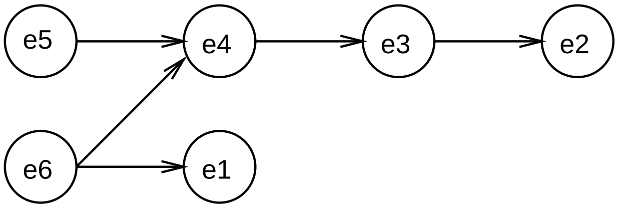


    - ∗ In creating $L_{[3]} = e_6, e_4, e_3, e_2$, the pair (e_6, e_4) is added to the relation:

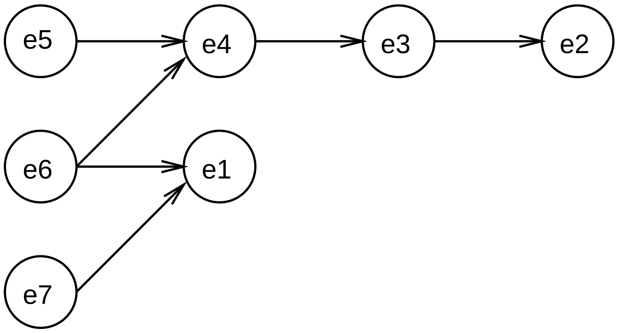


∗ In creating $L_{[4]} = e_6, e_2$, nothing is added.

∗ In creating $L_{[5]} = e_6, e_1$, the pair (e_6, e_1) is added:



∗ In creating $L_{[6]} = e_7, e_1$, the pair (e_7, e_1) is added:



– The Directed Acyclic Graph (DAG) thus created can be processed in $O(n)$ time to create a *topologically sorted list*.

– How?

1. Visit (any) leftmost node.
2. Print the node and delete it from the DAG.
3. If DAG is not empty, go to 1.

– Example: $e_5, e_6, e_7, e_1, e_4, e_3, e_2$.

– The list is not unique.

– Property of list: if e_i precedes e_j in the list and a sweep line passes through both, then e_i is left-of e_j.

– Topologically sorted lists are useful in other geometric algorithms.

- Note: it takes $O(n \log n)$ time to create a topologically sorted list ($O(\log n)$ per tree operation, $O(n)$ for topological sort).

12.6 The Trapezoid Tree

- Recall: the slab method takes up $O(n^2)$ space
  $\Rightarrow$ too much for some applications.

- Example: $n = 10,000$
  $\Rightarrow n^2 = 10^{10}$
  $\Rightarrow$ may not fit into memory.

- The trapezoid tree

  – uses $O(n \log n)$ space;

  – requires $O(n \log n)$ preprocessing time;

  – handles queries in time $O(\log n)$.

  This is close to optimal:

  – $O(n)$ space;

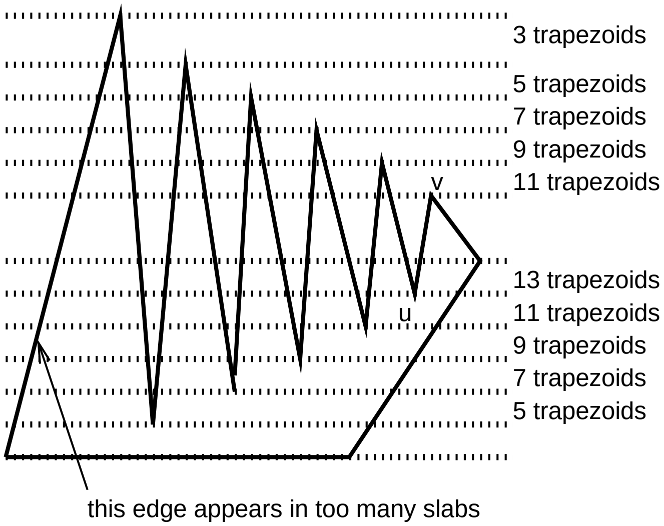
  – $O(n \log n)$ preprocessing time;

  – $O(\log n)$ per query.

  However, creating a trapezoid tree is a bit complicated.

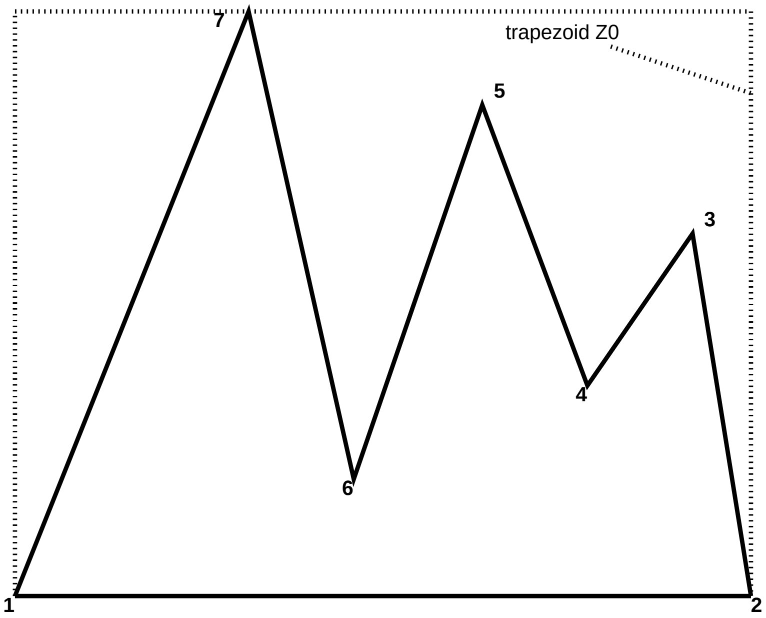
- Key ideas:

  – Consider the drawback of the slab method:



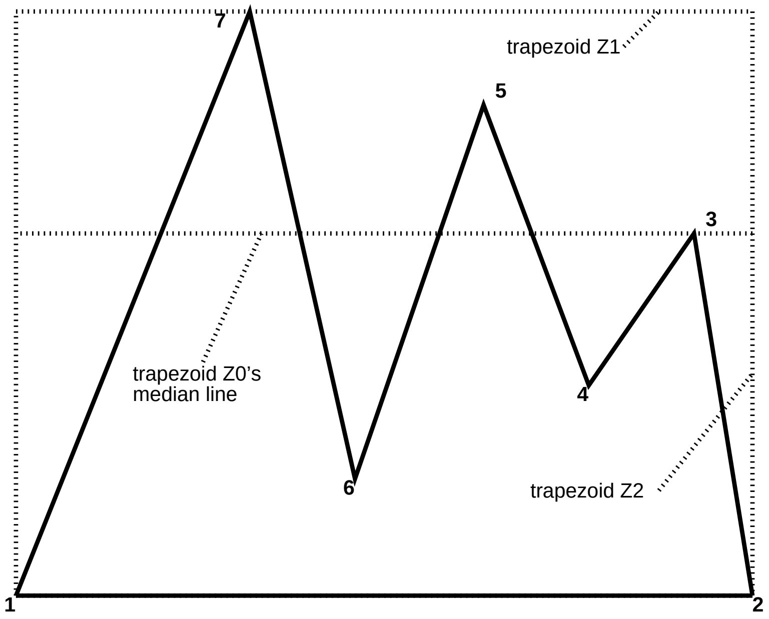
Long edges span several slabs.

- The slab decomposition is a *flat* one.
- The trapezoid tree uses a *hierarchical* decomposition:
  * The polygon is broken down into smaller and smaller trapezoids.
  * It avoids having the same edge show up in too many trapezoids.
- Example (to give the general idea):
  * First, the polygon is enclosed in an MBR:

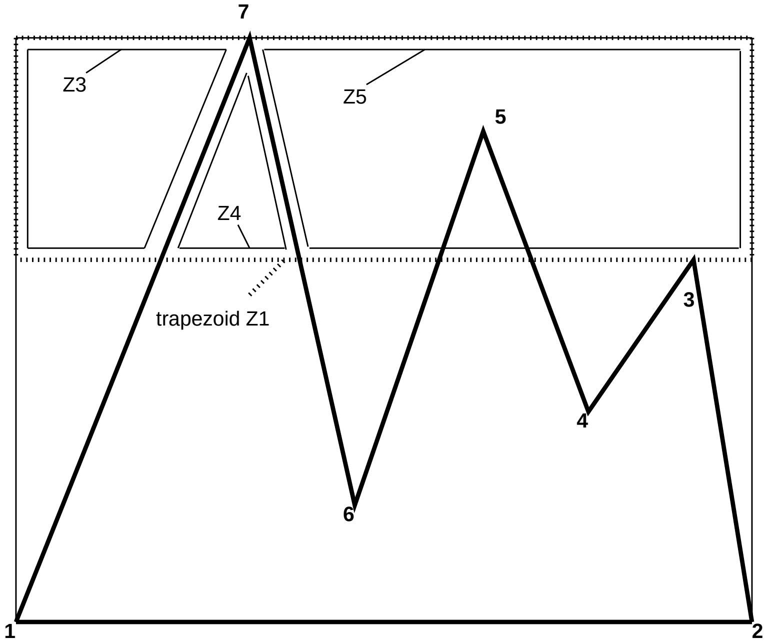


  * The MBR is itself a trapezoid - we will call this Z_0.

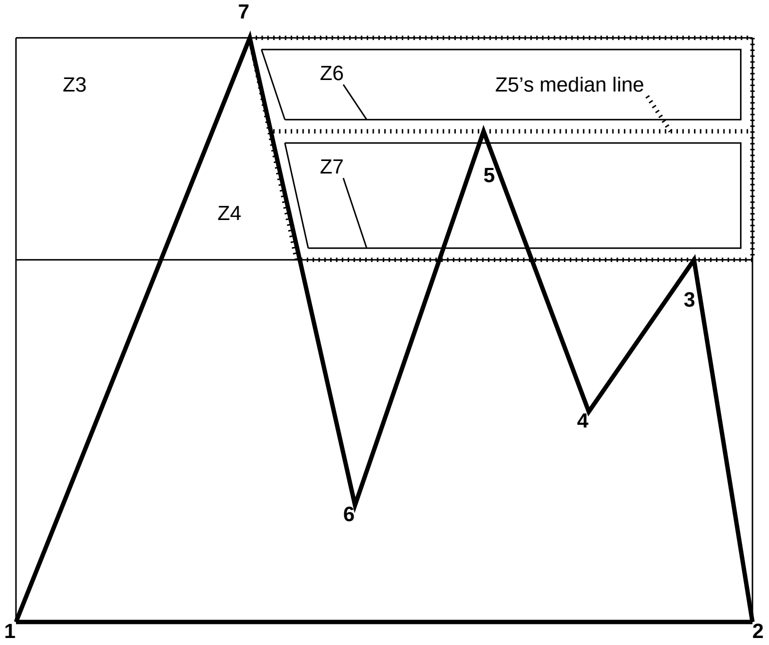
- Trapezoid Z_0 is partitioned via a horizontal line into two trapezoids, Z_1 and Z_2.
- The horizontal line goes through the median y-value in Z_0
  $\Rightarrow$ vertex 3.



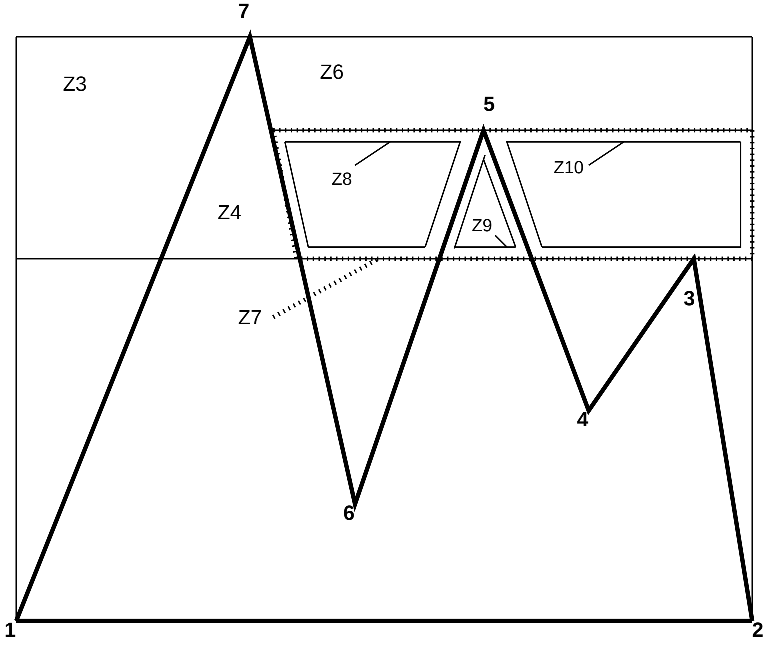
- Each sub-trapezoid is processed recursively.
- To process the upper trapezoid Z_1:
  - Look for edges that span the whole trapezoid.
  - Edges e_7 and e_6 span Z_1.
  - Use these edges to vertically partition Z_1
    $\Rightarrow$ creates 3 sub-trapezoids: Z_3, Z_4, Z_5:



- $*$ Now process each of Z_3, Z_4, Z_5 in turn.
- $*$ Z_3 is empty $\Rightarrow$ no further processing needed
  $\Rightarrow$ simply determine whether Z_3 is inside or outside P
  $\Rightarrow$ Z_3 is outside.
- $*$ Z_4 is empty $\Rightarrow$ no further processing needed
  $\Rightarrow$ simply determine whether Z_3 is inside or outside P.
  $\Rightarrow$ Z_4 is inside.
- $*$ Z_5 is not empty $\Rightarrow$ process it recursively.
- $*$ Note: the trapezoids are drawn slightly inside their actual boundaries for emphasis.
- $*$ Split Z_5 horizontally by passing a line through the y-value of its median point.

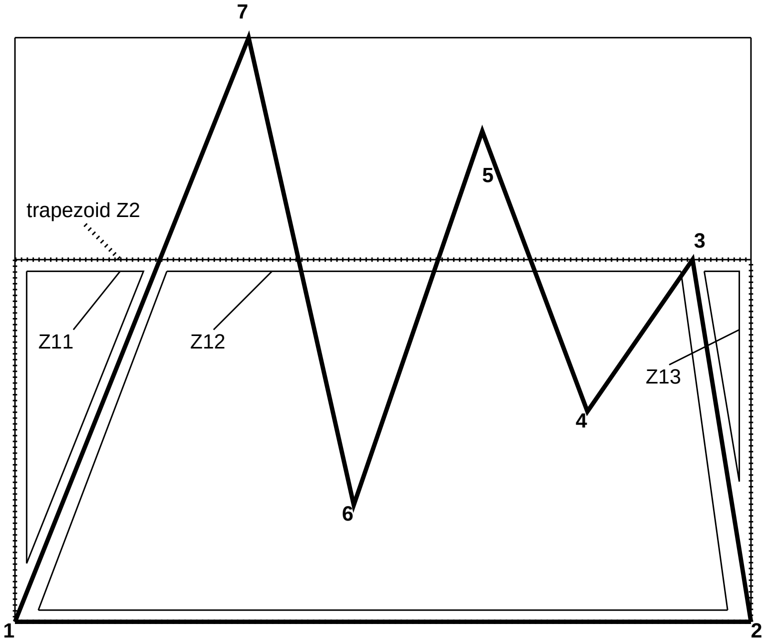


- $*$ This creates trapezoids Z_6 and Z_7.
- $*$ Z_6 is empty
  $\Rightarrow$ it is outside P.
- $*$ Z_7 is first vertically partitioned by all the edges that span Z_7 vertically.

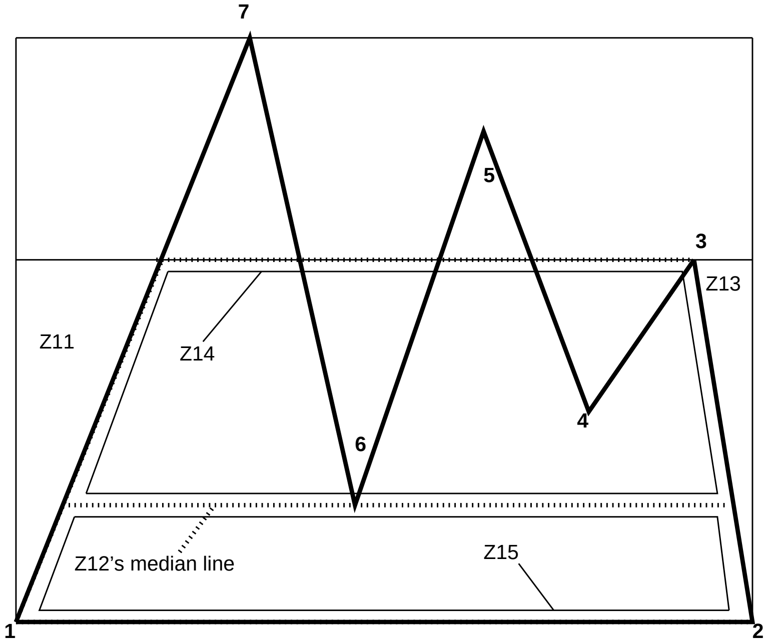


- $*$ This results in trapezoids Z_8, Z_9, Z_{10}.

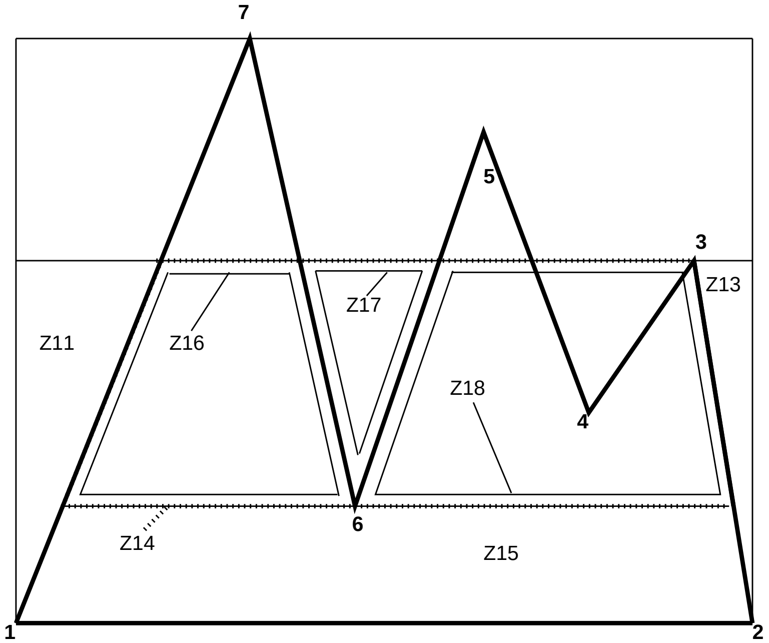
- Z_8 is empty and outside.
- Z_9 is empty and inside.
- Z_{10} is empty and outside.
- This completes the processing of the first upper trapezoid, Z_1.
- The original lower trapezoid, Z_2 remains to be processed.
- Z_2 is spanned by edges e_7 and e_2
  $\Rightarrow Z_2$ is vertically partitioned into three trapezoids Z_{11}, Z_{12}, Z_{13}.



- Of these, Z_{11} and Z_{13} are empty.
- Z_{11} is outside.
- Z_{13} is outside.
- Z_{12} is partitioned horizontally by its median y-value
  $\Rightarrow$ upper trapezoid Z_{14}, lower trapezoid Z_{15}.

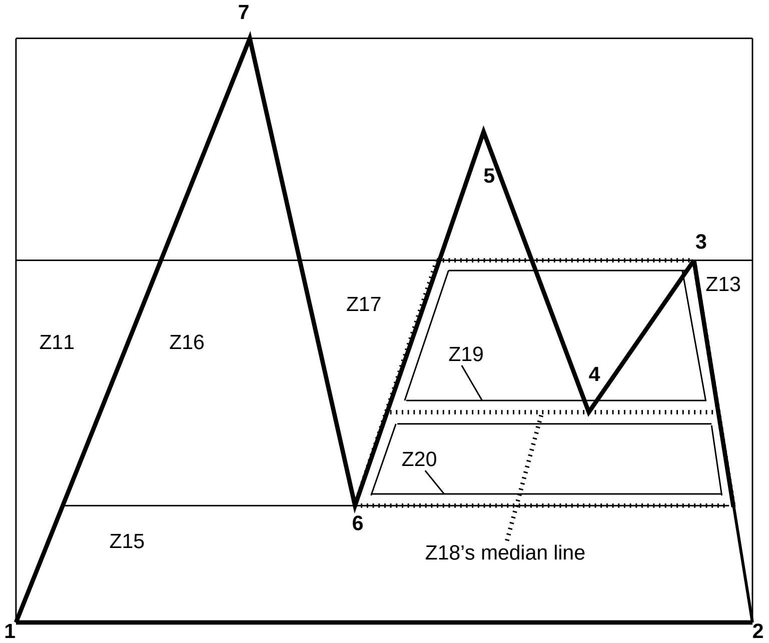


- Z_{15} is empty and inside.
- Z_{14} needs further processing.
- Z_{14} is spanned by edges e_5 and e_6
  $\Rightarrow$ vertically partition into Z_{16}, Z_{17}, Z_{18}.

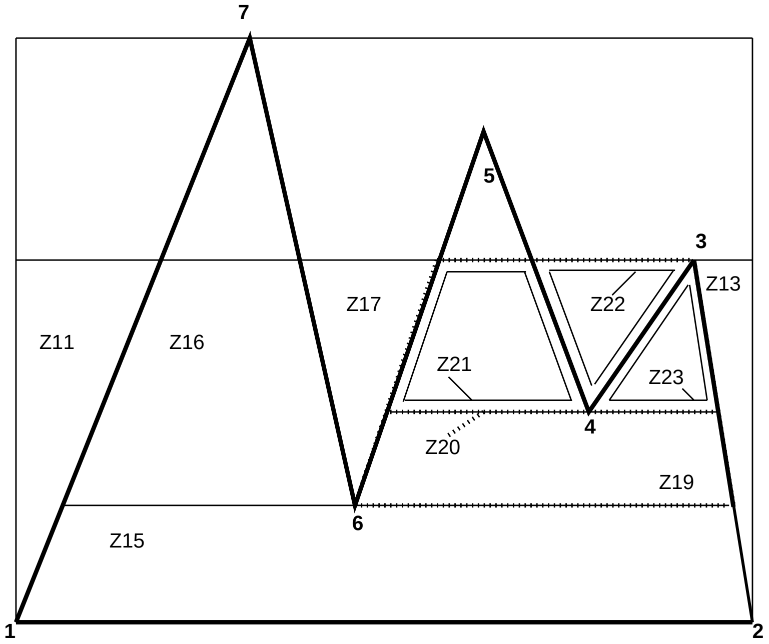


- Z_{16} is empty and inside.
- Z_{17} is empty and outside.

- Z_{18} is horizontally partitioned
  $\Rightarrow$ upper trapezoid Z_{19}, lower trapezoid Z_{20}.



- Z_{19} is empty and inside.
- Z_{20} is spanned by edges e_3 and e_4
  $\Rightarrow$ vertically partition into Z_{21}, Z_{22} and Z_{23}:

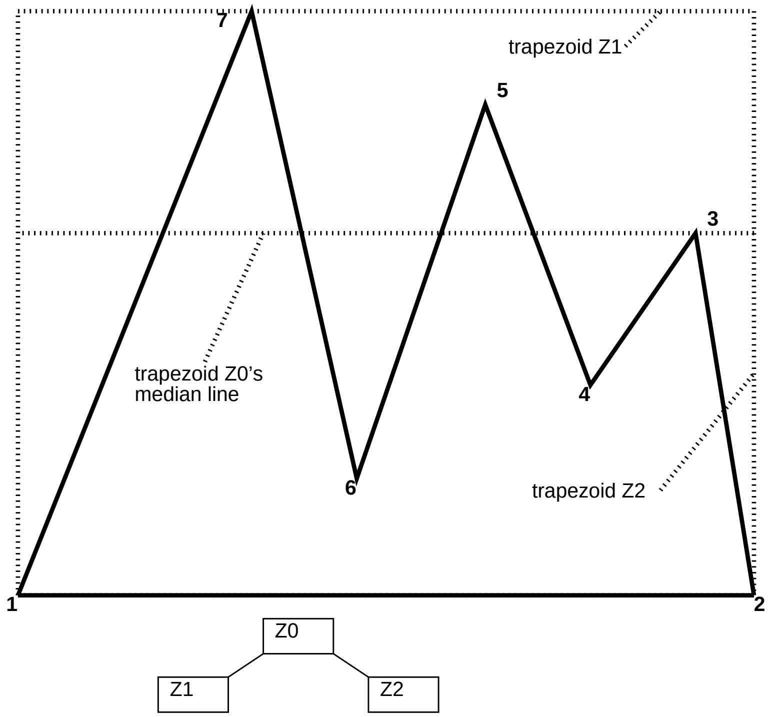


– What good is it to decompose a polygon into a trapezoid hierarchy?

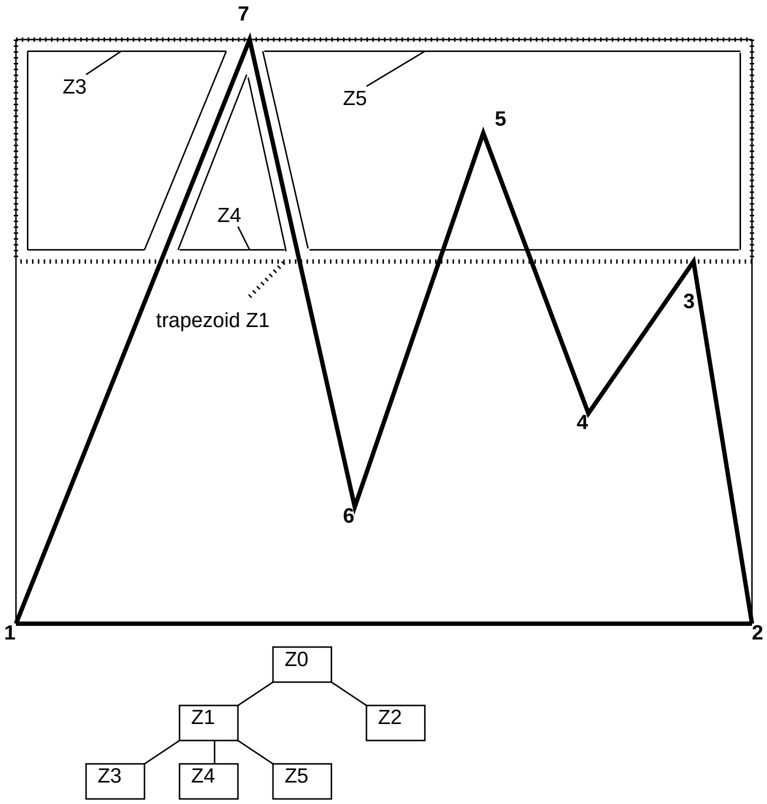
– A decomposition is useful if it helps with searching
  ⇒ try to represent decomposition in a tree.

– First, let's consider a straightforward "containment" tree:

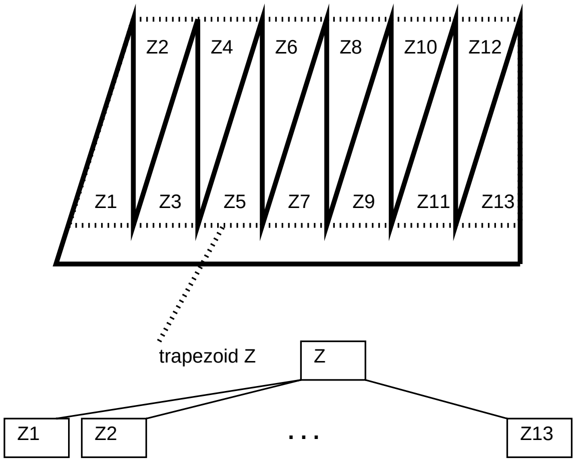
  $*$ Initially, Z_0 was decomposed into Z_1 and Z_2
    ⇒ create a root Z_0 with children Z_1 and Z_2:



  $*$ Z_1 is partitioned (vertically) into Z_3, Z_4, Z_5:



- ∗ We could continue in this fashion until the whole tree is built.
- ∗ Unfortunately, this approach does not bound the degree of a tree node, for example:

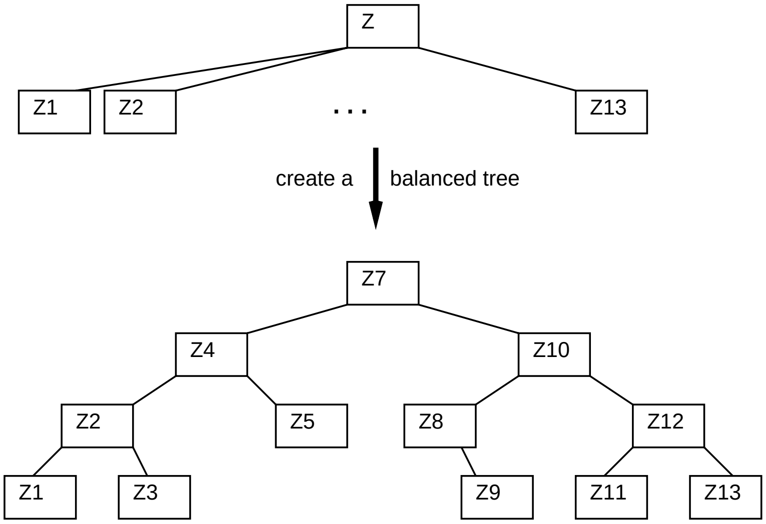


In this case, the trapezoid shown decomposes into too many "child" trapezoids

⇒ a node can have $O(n)$ children in the tree

⇒ cannot provide $O(\log n)$ search time.

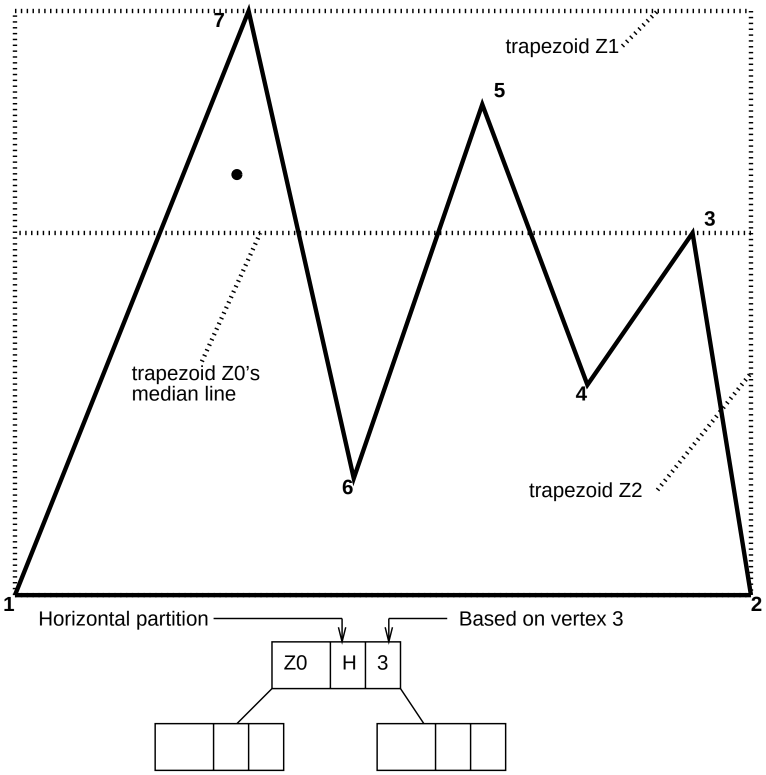
- The solution is to limit the degree of each node, e.g., at most two ⇒ binary tree.
- To handle cases like the above one, simply create a balanced tree:



- Once a tree has been created, how are queries processed?
  * The coordinates of a query point are used to navigate down the tree.
  * The leaf level contains empty trapezoids.
  * Each trapezoid is either inside or outside the polygon.
  * The status of a query point is the status of the leaf-trapezoid it lies in.

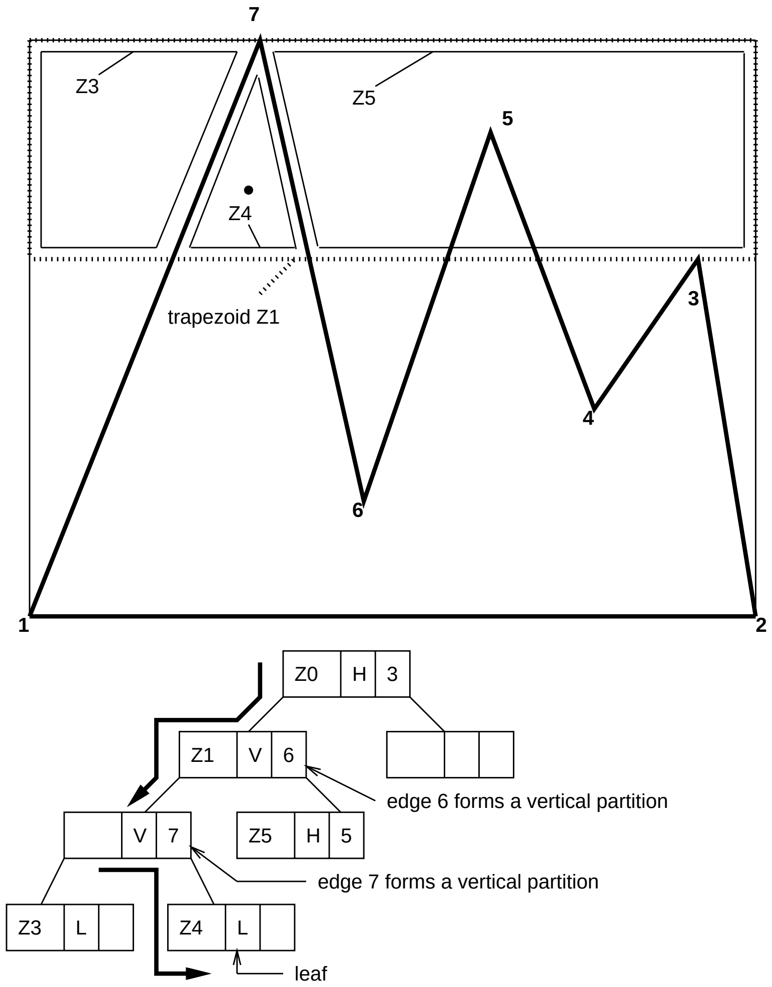
– Note:

- Some trapezoids were partitioned vertically; others horizontally.
- Consider a horizontal partition:



- To decide which subtree
  ⇒ compare y-value of point with y-value of partition line
  ⇒ use y-value of vertex 3.

∗ Next, consider a vertical partition:

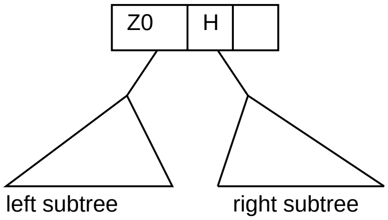

7
Z3
Z5
5
Z4
trapezoid Z1
3
4
6
1
2
Z0
H
3
Z1
V
6
edge 6 forms a vertical partition
V
7
Z5
H
5
edge 7 forms a vertical partition
Z3
L
Z4
L
leaf

∗ To decide which subtree, see if point lies on left or right of edge segment.

- Implementation: key ideas
  - Recall:
    - The polygon is hierarchically subdivided into trapezoids.
    - A division is either horizontal (using a median y-value) or vertical (using a spanning edge).
    - If more than one edge spans a trapezoid, a balanced subtree needs to be constructed.
    - The initial trapezoid is the MBR.
    - The initial trapezoid is the root of the tree.
  - We will write a recursive function

    $$\text{root} := \textsc{Make-Trapezoid-Subtree}\ (Z_0)$$

    such that the tree is obtained using the initial trapezoid Z_0.



  - This function will create the left and right subtrees recursively.
  - Data needed for each trapezoid Z (to process it):
    - Z's boundary:

Z.top	top parallel line of Z
Z.bottom	bottom parallel line of Z
Z.leftedge	the polygon edge that forms Z's left edge
Z.rightedge	the polygon edge that forms Z's right edge

    - Z.n – The number of polygon points in Z.
    - Z.v[1],...,Z.v[Z.n] – the polygon points in Z.
    - Z.edgelist – polygon edges that pass through the interior of Z, topologically sorted.

- Version 1.0 of MAKE-TRAPEZOID-SUBTREE:

```
MAKE-TRAPEZOID-SUBTREE ($Z$)
  if $Z$ is empty
    – create a leaf node
    – check containment in $P$
    – return leaf node pointer
  endif

  // Otherwise, proceed with partition
  Find median vertex in $Z$
  Create upper and lower trapezoids $Z_U$ and $Z_L$
  // Process edges in $Z$'s edge list
  for each edge $e = (a, b)$ in $Z$.edgelist
    // Test $e$ with respect to $Z_U$:
    if $a$ or $b$ is in $Z_U$
      insert in $Z_U$'s vertex list
    if $e$ spans $Z_U$
      // Create new trapezoid recursively
      – Treat $e$ as right edge of trapezoid $\bar{Z}$
      – Create $\bar{Z}$'s edgelist and vertex list
    endif
    // Test $e$ with respect to $Z_L$:
      // Similar to above
  endfor
```

Note:

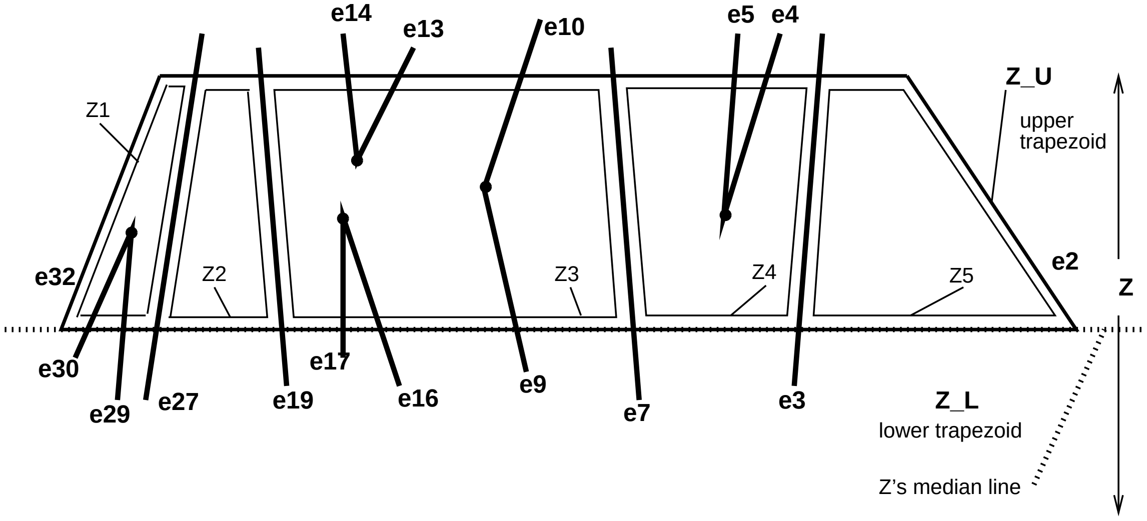
- Trapezoids in for-loop are created left to right.
- By scanning the edges left to right, we will have the edge list for each trapezoid by the time the trapezoid's right edge is scanned.
  ⇒ this is why we keep the edges in a topologically sorted list.

Consider the for-loop:

- When the tree node for $\bar{Z}$ is created, what is done with it?
- If several edges span Z_U, we will end up with a *collection* of (independent) subtrees in the for-loop.
- These subtrees have to be placed in a *balanced* tree
  ⇒ balancing mechanism needed.
- Balancing must be done *after* the for-loop.

• More about balancing:

- Consider an example:



∗ Here, trapezoid Z has been horizontally partitioned into Z_U and Z_L. (Only Z_U is shown).

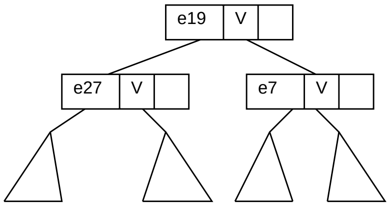
∗ Z_U has edges e_{32} and e_2 for left and right sides.

∗ Edges e_{27}, e_{19}, e_7 and e_3 are spanning edges.

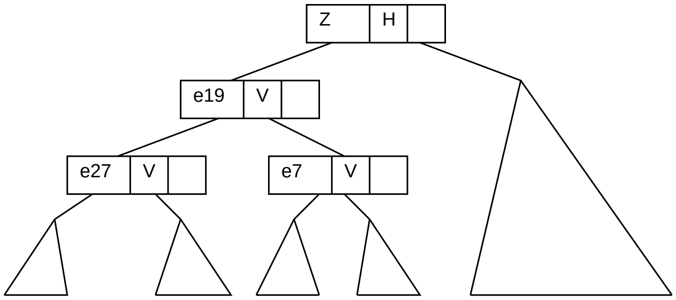
∗ The spanning edges create 5 subtrapezoids.

∗ These trapezoids are placed in a balanced tree.

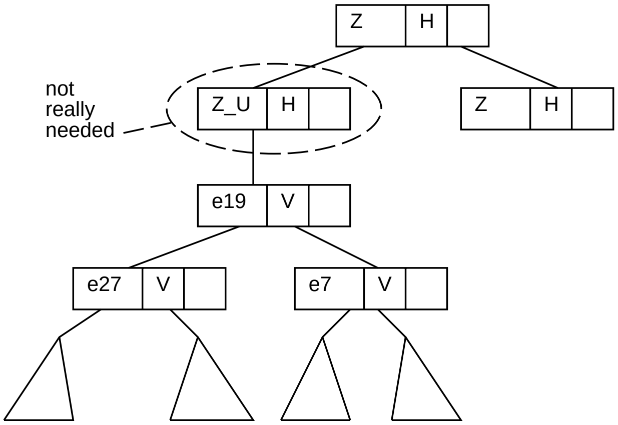
- For example, the balanced tree could turn out to have e_{19} as root:



- This would become a subtree of Z:



- Note that Z_U is only a transient trapezoid (not actually appearing in the tree).
- Another way to think about it:

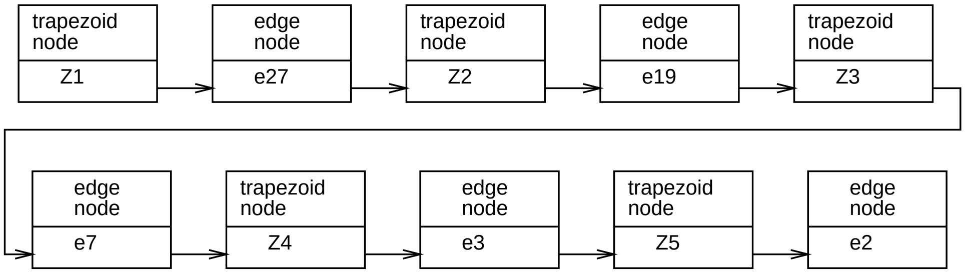


– Now, the trapezoids $Z_1, \ldots, Z_5$ are recursively created in the for-loop

$\Rightarrow Z_1$ (could be a huge tree) is created before Z_3.

– But, balancing needs to be done later – after the for-loop.

- What to do:
  - Create a list of subtrees (one for Z_1, one for Z_2 etc) while inside the for-loop.
  - Use list as input to a balancing algorithm outside the for-loop.
- Note that some tree nodes are “edge” (vertical) nodes.
- Thus, list should contain these edges.
- We will create a *balance-list* inside the for-loop:
  - A balance-list is a (linked) list of items.
  - Each item is either a spanning edge or a trapezoid tree node.
- Example (above):

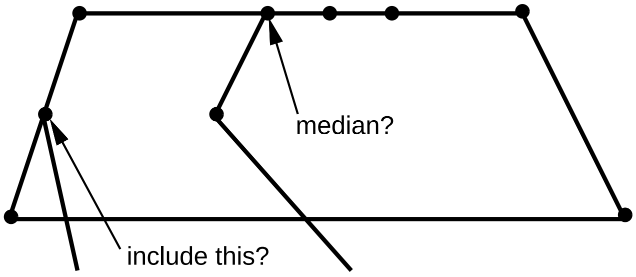


- Version 2.0 of Make-Trapezoid-Subtree:

```
Make-Trapezoid-Subtree ($Z$)
   if $Z$ is empty
      – create a leaf node
      – check containment in $P$
      – return leaf node pointer
   endif

   // Otherwise, proceed with partition
   Find median vertex in $Z$
   Create upper and lower trapezoids $Z_U$ and $Z_L$
   Create a left-balance-list and right-balance-list
   // Process edges in $Z$'s edge list
   for each edge $e = (a, b)$ in $Z$.edgelist
```

```
        // Test $e$ with respect to $Z_U$:
        if $a$ or $n$ is in $Z_U$
            insert in $Z_U$'s vertex list
        if $e$ spans $Z_U$
            // Create new trapezoid recursively
            – Treat $e$ as right edge of trapezoid $\bar{Z}$
            – Left edge of $\bar{Z}$ := current_left_edge
            – temp_node := MAKE-TRAPEZOID-SUBTREE ($\bar{Z}$)
            – add temp_node to left-balance-list
            – Make an edge node out of $e$ and add to left-balance-list
            – current_leftedge := $e$
        endif
        // Test $e$ with respect to $Z_L$:
            // Similar to above, except use right-balance-list
    endfor
    // Now create root of subtree and balance the two lists
    Create root of subtree $R$
    $R$.leftchild := BALANCE (left-balance-list)
    $R$.rightchild assn BALANCE (right-balance-list)
    return $R$
```

- Details: should points on the boundary of a trapezoid Z be included in Z's vertex list?

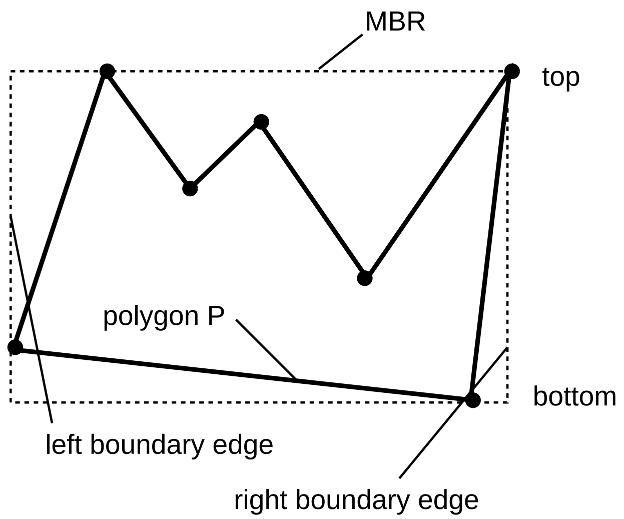


Convention:

- Do not consider points on top and bottom sides of trapezoid.

  - Do consider points on left and right sides.

- Details: Creating the initial trapezoid Z_0



  - Create MBR.
  - Some MBR edges are not in P.
  - Assume P's edges are $e_1, \ldots, e_n$.
  - Create special codes for
    - $*$ left boundary edge
    - $*$ right boundary edge

    These will be needed in trapezoids.
  - Only need y-values of top and bottom edges of MBR.
  - To create Z_0's edgelist
    $\Rightarrow$ use plane-sweep to topologically sort edges in P.

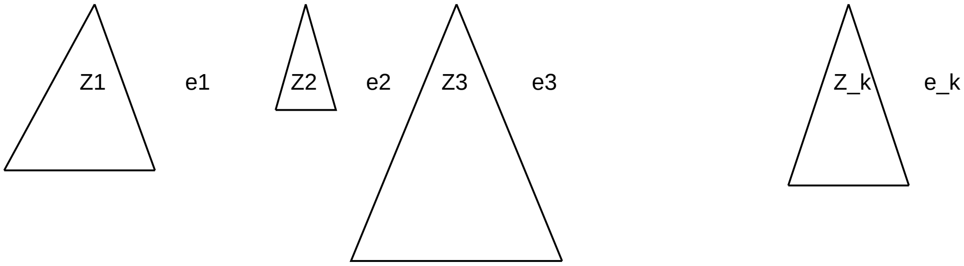
- Details: what does a tree node contain?

  Suppose Znode is a tree node.

  - Znode.leaf – A boolean indicating whether Znode is a leaf.
  - Znode.vertical – is this a vertical or hortizontal partition?
  - Znode.ymedian – if horizontal, we need the y-value of the hortizontal partition.

  - Znode.edge – if vertical, we need the spanning edge.
  - Znode.leftchild – pointer to left subtree.
  - Znode.rightchild – pointer to right subtree.

- Details: balancing a balance-list
  - Recall: a balance-list is a (linked) list of tree nodes.
  - Each node is either the root to a subtree or a spanning edge, alternating in the list.
  - Let's use the notation: $Z_1 e_1 Z_2 e_2 \ldots Z_k e_k$.
  - We need to create a balanced tree out of this list.
  - Note: each Z_i is a subtree (itself containing trapezoids etc). Thus, it is better to think of the list as:



  - Observe: subtrees can be of different sizes
    ⇒ must take sizes of subtrees into account while balancing.
  - Define:

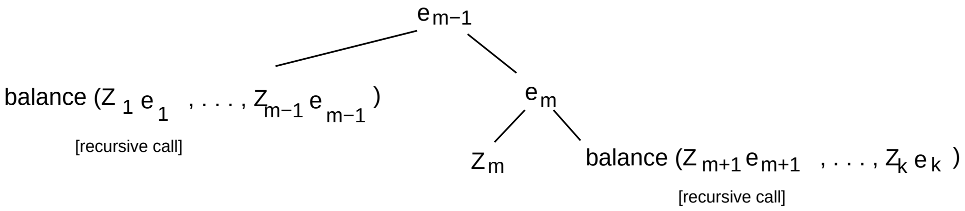
$$\begin{array}{lcl} Z.\text{node\_weight} & = & 1, \text{ if } Z \text{ was horizontally partitioned} \\ & & 1, \text{ if } Z \text{ is a leaf trapezoid} \\ & & 0, \text{ otherwise} \\ Z.\text{weight} & = & Z.\text{node\_weight} \\ & & +\ Z.\text{leftchild}\rightarrow\text{weight} \\ & & +\ Z.\text{rightchild}\rightarrow\text{weight}. \end{array}$$

    Informally: Z's weight is the number of leaf trapezoids plus the number of non-edge nodes in the subtree with Z as root.
  - In balancing, we create a weight-balanced tree.
  - To create the tree from the list $Z_1 e_1 \ldots Z_k e_k$:

- ∗ Find the smallest m such that

$$\sum_{i=1}^{m} \text{weight}(Z_i) \; > \; \frac{1}{2}\sum_{i=1}^{k} \text{weight}(Z_i)$$

- ∗ Create a tree rooted at e_{m-1} (a vertical node):


e_{m-1}
balance ($Z_1 e_1$, . . . , $Z_{m-1} e_{m-1}$)
[recursive call]
e_m
Z_m
balance ($Z_{m+1} e_{m+1}$, . . . , $Z_k e_k$)
[recursive call]

- ∗ Note: several special cases exist for bottoming out of the recursion.

- Pseudocode:

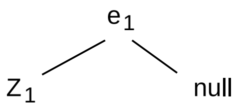
```
Algorithm:   Make-Trapezoid-Tree ($P$)

Input: polygon $P$.
Output: root of trapezoid tree
    $R$ := Compute-MBR ($P$);
    $Z$.top := $R$.topright.y;
    $Z$.bottom := $R$.bottomleft.y
    $Z$.leftedge := Leftmost-Edge-of-MBR;
    $Z$.rightedge := Rightmost-Edge-of-MBR;
    $Z$.edgelist := Topological-Sort ($P$);
    Add $Z$.rightedge to $Z$.edgelist;
    $Z$.v := Nonboundary-Vertices ($P$,$Z$);
    root := Make-Trapezoid-Subtree ($Z$);
    return root;
```

Algorithm: Make-Trapezoid-Subtree (Z)

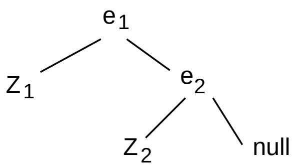
Input: Trapezoid Z.
Output: Root of subtree containing Z and its subtrapezoids.

```
$R$ := $Z$'s parameters (top, bottom etc);
if $Z$.n = 0 // Empty trapezoid ⇒ recursion ends
    $R$.weight := 1;
    $R$.leaf := true;
    $R$.inside := Check-Inside ($Z$);
    return $R$;
endif
// Otherwise, create horizontal cut
ymedian := Median-Vertex-Yvalue ($Z$);
Initialize trapezoids and balance-lists;
for each edge $e = (a, b)$ in $Z$.edgelist
    // Check top trapezoid
    if $a$ above ymedian add $a$ to $Z_U$;
    if $b$ above ymedian add $b$ to $Z_U$;
    if $e$ spans $Z_U$
        $Z_U$.leftedge := current_leftedge;
        $Z_U$.rightedge := $e$;
        Add $e$ to $Z_U$.edgelist;
        $r$ := Make-Trapezoid-Subtree ($Z_U$);
        Add $r$ to left-balance-list;
        Add $e$ to left-balance-list;
        current_leftedge := $e$;
        Reset $Z_U$;
    endif
    // Check bottom trapezoid $Z_L$. Similar to above – omitted
endfor
$R$.leaf := false;
$R$.leftchild := Balance (left-balance-list);
$R$.rightchild := Balance (right-balance-list);
$R$.weight := 1 + $R$.leftchild→weight + $R$.rightchild→weight;
return $R$;
```

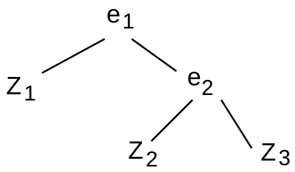
- Most other functions are straightforward. Only BALANCE() is a little complex:
  - BALANCE can be written recursively.
  - Several special cases for ending recursion have to be considered:
    1. Balance list is empty
       ⇒ return null pointer.
    2. Balance list is a single trapezoid Z
       ⇒ create a leaf node and return it.
    3. Balance list: $Z_1 e_1$
       ⇒ return



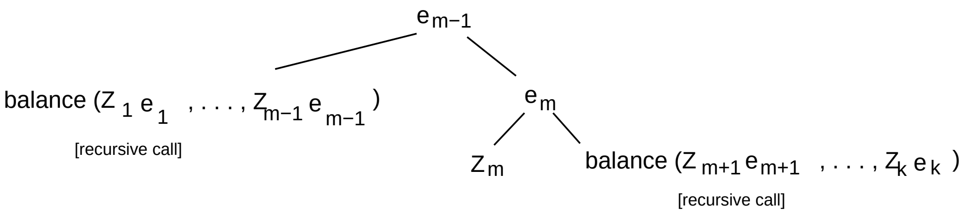
    4. Balance list: $Z_1 e_1 Z_2 e_2$
       ⇒ return



    5. Balance list: $Z_1 e_1 Z_2 e_2 Z_3$
       ⇒ return

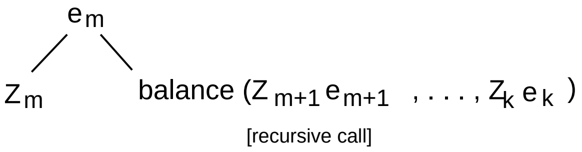


    6. Remaining cases use recursion:
       (a) Balance list: $Z_1 e_1 \ldots Z_{m-1} e_{m-1} Z_m e_m \ldots Z_k e_k$
           ⇒ return

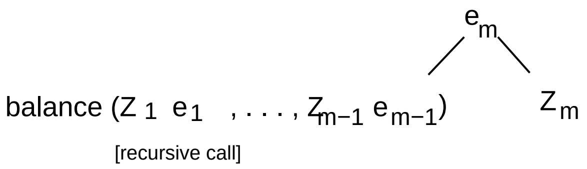


em−1
balance (Z 1 e 1 , . . . , Zm−1 e m−1)
[recursive call]
em
Zm
balance (Z m+1 em+1 , . . . , Zk ek)
[recursive call]

(b) Balance list: $Z_m e_m \ldots Z_k e_k$ (m turns out to be 1)
$\Rightarrow$ return



(c) Balance list: $Z_1 e_1 \ldots Z_m e_m$ (m turns out to be k)
$\Rightarrow$ return



- Why does it all work?

  To show that query time is $O(\log n)$, it is sufficient to show that any path from root to leaf in the tree is of length at most $O(\log n)$.

  Consider the path from the root to any leaf node.

  – How many horizontal-partition nodes can we encounter along the way?

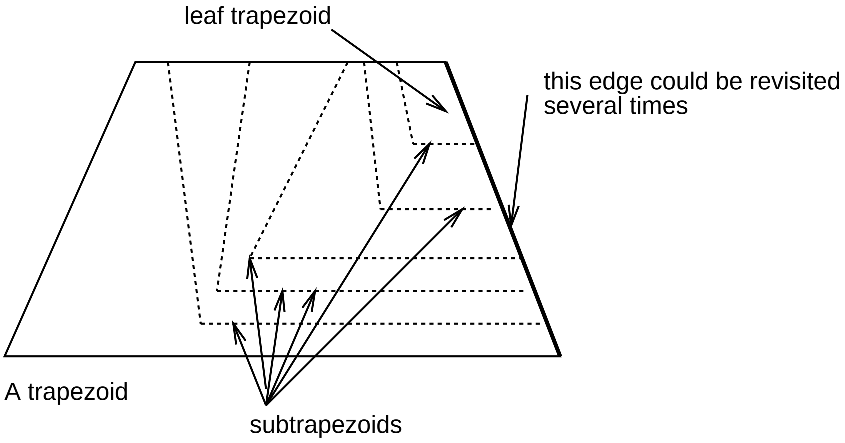
    ∗ The first H-node encountered cuts the number of points by half (approximately).

    ∗ This (half) set is further cut in half by the next H-node...etc.

    ∗ You can successively partition at most $\lceil \log n \rceil$ times.

  – What about vertical nodes?

- Note that a single edge can be revisited many times on the path:



- How many times can a single edge be revisted?
  ⇒ worst-case, as many times as it is cut horizontally
  ⇒ at most $\lceil \log n \rceil$ times.
- How many different edges can we visit?
  * Every time we visit an edge, we go into either the left or right space partitioned by the edge.
  * This partitioning (with n edges can occur at most $O(\log n)$ times by the balancing mechanism.
  * Since balancing can result in an error by ± 1, it's at most $2\lceil \log n \rceil$.
- Some of these edges could be revisted ($O(\log n)$ visits per edge)
  ⇒ $O(\log^2 n)$ overall?
- No! The total number of revisits cannot exceed the number of horizontal cuts
  ⇒ $\lceil \log n \rceil$ revisits overall.
- Thus, path length $\leq 4\lceil \log n \rceil$

Thus, we've shown that

- Query time is $O(\log n)$.
- Storage is $O(n \log n)$.

What about preprocessing time?

- $O(n \log n)$ for the topological sort.
- If we ignore the for-loop in MAKE-TRAPEZOID-SUBTREE, only $O(1)$ work is done per node creation
  $\Rightarrow O(n \log n)$ work overall.
- Consider the total work done over all executions of the for-loop:
  - It is bounded by the number of edge-fragments created.
  - $O(1)$ work per edge-fragment.
  - But an edge is fragmented at most $\lceil \log n \rceil$ times.
  - Therefore, n edges are fragmented at most $n \lceil \log n \rceil$ times
    $\Rightarrow O(n \log n)$ work overall.

Thus, preprocessing is $O(n \log n)$.

12.7 Point Location: Summary

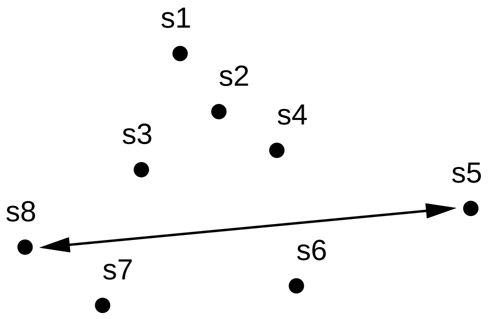
- Summary of methods covered:
  - *Ray-Crossings Method*:
    - $O(n)$ per query.
    - $O(n)$ storage (only for polygon).
    - No preprocessing.
    - Ideal for single-shot query.
  - *2D-Hashing*:
    - $O(1)$ per query (average, with uniform assumption), $O(n)$ worst-case.
    - $O(n)$ storage (average, with uniform assumption), $O(n^2)$ worst-case.
    - $O(n\sqrt{n})$ preprocessing (average, with uniform assumption), $O(n^2)$ worst-case.
    - Easy to implement, practical.
  - *Slab-Method*:
    - $O(\log n)$ per query.
    - $O(n^2)$ preprocessing.
    - $O(n^2)$ storage.
    - Easy to implement.
    - Too much storage required for large n.
  - *Trapezoid Method*:
    - $O(\log n)$ per query.
    - $O(n \log n)$ preprocessing.
    - $O(n \log n)$ storage.

    - Implementation more complex.
    - Good worst-case efficiency.

- Other methods:
  - *Triangulation-Tree Method*:
    - Triangulate polygon (can be done in time $O(n)$).
    - Use coarse triangulations higher up the tree.
    - $O(n \log n)$ preprocessing.
    - $O(\log n)$ per query.
    - $O(n)$ storage.
      $\Rightarrow$ optimal!
    - Much more complex to implement than Trapezoid Method. (Triangulation is difficult to implement).
  - *Persistent-Tree Method*:
    - Also optimal.
    - Complex implementation.
- Current research:
  - Data structures for point location that allow insertion and deletion of edges and points.
  - Practical, easy-to-implement and efficient algorithms.

12.8 The Set Diameter Problem

- Query: given a collection of points $S = \{s_1, \ldots, s_n\}$, find two that are farthest apart.
- Example:



Here, s_5 and s_8 are farthest apart.

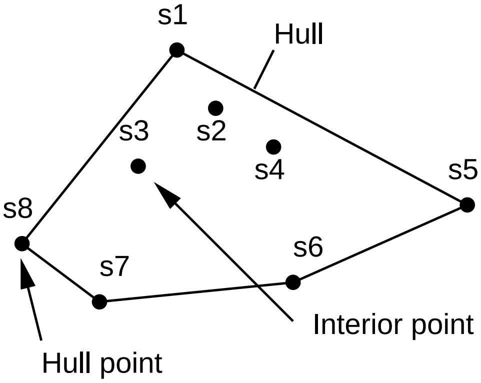
- Note: the problem has only a single-shot form.
- *All-pairs Method*:
  - Enumerate all possible pairs of points.
  - For each pair compute distance between points.
  - Pick pair with the largest distance
    ⇒ diameter.
  - Complexity: $\left(\begin{array}{c} n \\ 2 \end{array}\right) = O(n^2)$.
- *Convex-Hull Method*:
  - Compute convex hull of S in time $O(n \log n)$.
  - Compute diameter of Hull(S) in time $O(n)$.
    ⇒ $O(n \log n)$ overall.

Next, we will consider the details of each of the above computations.

12.9 Convex Hull: Introduction

- What is the *convex hull* of a set of points S?
  ⇒ the smallest (in perimeter) convex polygon that encloses S.

- Example:



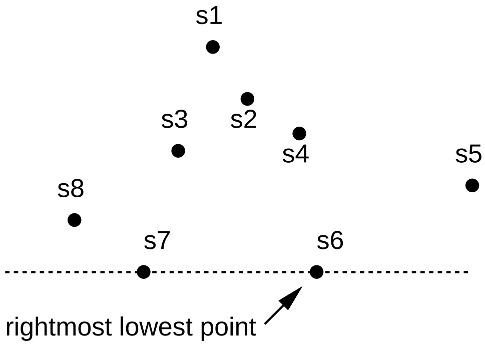
- Equivalent definitions:
  - The set of all convex combinations of points in S.
    ⇒ If $S = \{s_1, \ldots, s_n\}$, then for real numbers $\alpha_i \geq 0$ that satisfy $\sum_i \alpha_i = 1$, the convex combination $\sum_i \alpha_i s_i$ is a point.
  - The intersection of all half-spaces than contain S.

- Computing the convex hull: the *Gift-Wrapping Method*:
  - Suppose the given points are nails on a board.
  - Wrap a string around
    ⇒ you get the convex hull.
  - How to wrap?
    - Start with a point on the hull:
      ⇒ Pick the lowest point. If there are several, pick the rightmost lowest point.

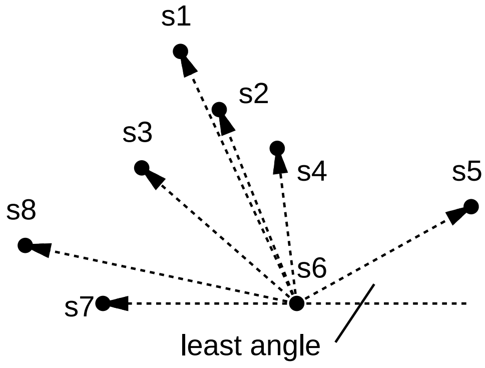
- Wrap the string around counter-clockwise
  ⇒ make left turns with each bend in the string.
- To decide next point where string turns:
  - For each point: compute angle required to turn to the point (left-turn).
  - Pick the point that requires the least turning angle.
- At each step we compute $O(n)$ angles. There are at most $O(n)$ steps
  ⇒ $O(n^2)$ computation.

– Example:

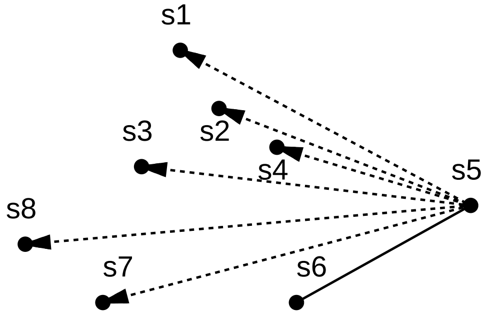
- Pick start: rightmost lowest point, s_6.



- Compute angles to all other points from lowest line:



- The least angle is the angle to s_5
  ⇒ s_5 is the next point
  ⇒ (s_6, s_5) is a hull edge.
- Compute angles to all other points from edge (s_6, s_5):

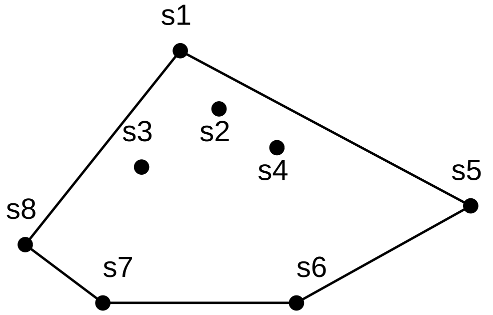


- ∗ The least angle (counter-clockwise) is the angle to s_1

  ⇒ s_1 is the next point

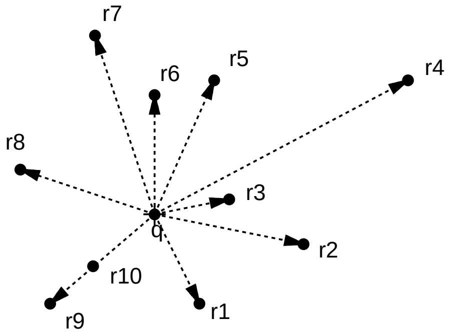
  ⇒ (s_5, s_1) is the next hull edge.

- ∗ ... continue until the start point is reached:

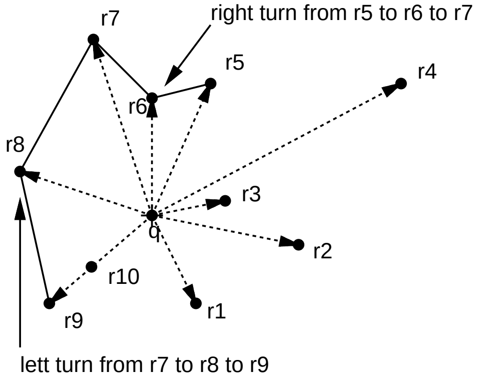


12.10 Convex Hull: Graham's Scan

- Key ideas:
  - Find any point q (not necessarily in S) that is internal to Hull(S).
  - Make q the origin and sort $s_1, \ldots, s_n$ by angle (using polar coordinates).
  - Find a hull point and label it r_1.
  - Label the points $r_1, \ldots r_n$ in counter-clockwise sort order.
  - Example:

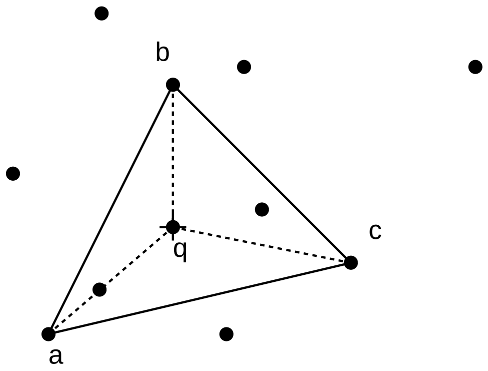


  - Next, consider groups of 3 consecutive points, r_{i-1}, r_i, r_{i+1}.
  - If angle $r_{i-1} r_i r_{i+1}$ is a right-turn then r_i is internal to the hull, e.g.,

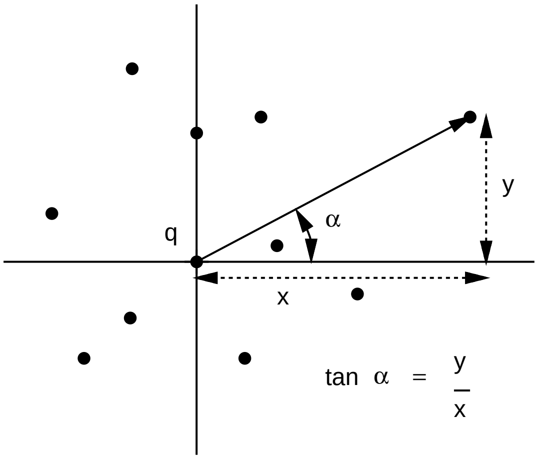


  - By successively scanning groups of points, get rid of points internal to hull.

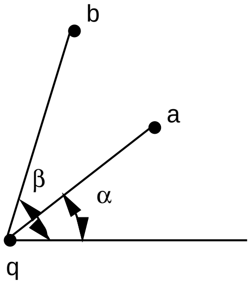
- Details: find the internal point (origin)
  - The centroid of any three non-collinear points must be internal to the hull (since it's internal to the hull of the 3 points).



  - To find 3 non-collinear points in S:
    - Pick any two points a and b.
    - Scan remaining points.
    - For each point scanned, check if it's collinear with (a, b).
    - If all points are collinear with (a, b), then S is a line (degenerate case).
    - Otherwise, we *will* find a 3rd point c that's not collinear with a and b.
    - Note: this takes $O(n)$ time.
  - Compute centroid of the three points: $a = \frac{1}{3}(a + b + c)$.
- Details: sorting the points in S by angle
  - One approach:
    - Translate coordinates so that q is the origin.
    - Compute angles for each point using tan^{-1}.

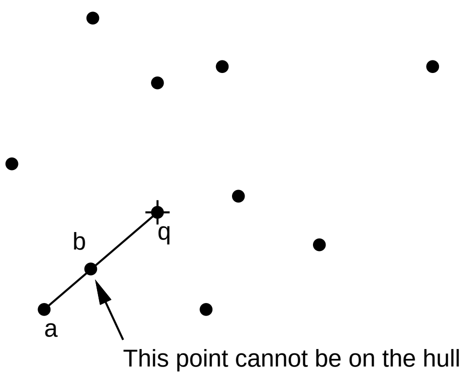


    - Sort points by angle.
  - A better approach:
    - Translate coordinates so that q is the origin.
    - Observation: for sorting, we don't really need to *compute* angles; we only need to *compare* them.
    - Comparison criterion:



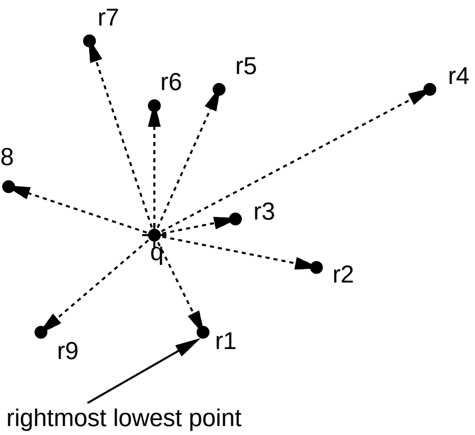
Angle β is greater than angle α if b is *left-of* line segment (q, a).

- Details: removal of some radially collinear points:
  - Among radially collinear points, retain only the outermost:



Here, we retain only a (for hull computation).

- Details: finding at least one hull point
  - Find lowest point in S (point with least y-value).
  - If there are several such points, pick rightmost
    ⇒ rightmost lowest point.
  - Label this point r_1.
  - Label remaining points in counter-clockwise sort order.



- Details: computing the hull
  - Create a doubly-linked list of points in sort-order $r_1, \ldots, r_n$ such that:
    - * $\text{Next}(r_i) = r_{i+1}$.
    - * $\text{Prev}(r_i) = r_{i-1}$.

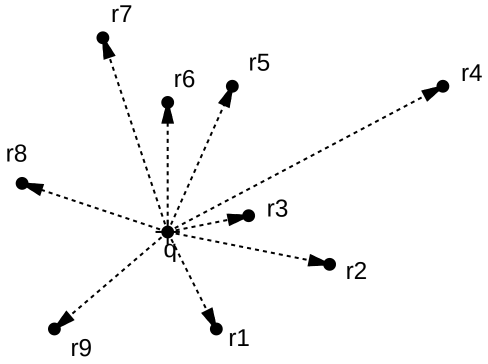
- Note: Next(r_n) = r_1 and Prev(r_1) = r_n.
- Note: if r_i gets deleted from the list, then
    * Next(r_{i-1}) = r_{i+1}.
    * Prev(r_{i+1}) = r_{i-1}.
- Scan groups of 3 points, deleting "right-turn points":

```
Current := $r_1$;
while Next(Current) $\neq r_1$
   $a$ := Next(Current);
   $b$ := Next($a$);
   if (Current, $a$, $b$) is a left turn
      Current := $a$;
   else
      Delete $a$;
      if Current $\neq r_1$ // We don't want to go past $r_1$
         Current := Prev(Current);
   endif
endwhile
```

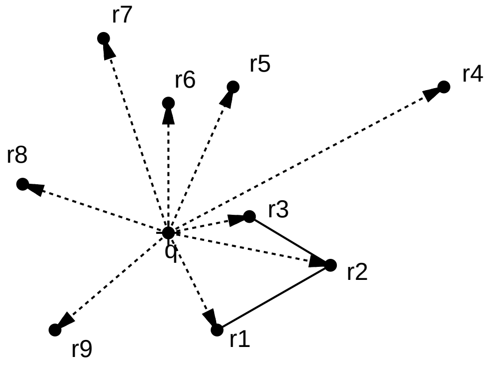
- After execution, the list has the hull vertices in counter-clockwise order.

Note: r_1 cannot be deleted because it is a hull point.

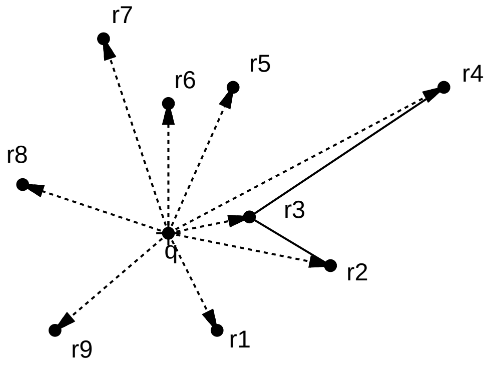
- Example:
    - Suppose we have found internal point q and labeled the the points in counter-clockwise sort-order:



- Initial list: $r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9$.
- Step 1:
  - Current = r_1.
  - r_1, r_2, r_3 is a left turn.
$\Rightarrow$ advance Current.
  - Current := r_2.
  - List: $r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9$.

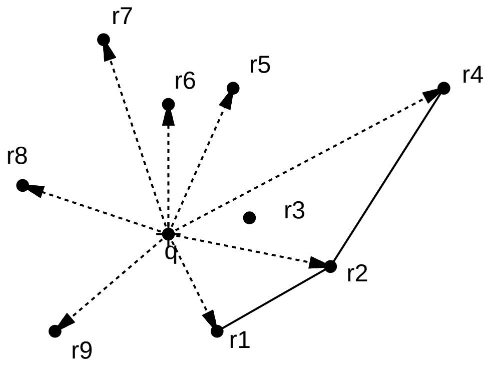


- Step 2:
  - Current = r_2.
  - r_2, r_3, r_4 is a right turn.
$\Rightarrow$ delete r_3.
  - Current := r_1.
  - List: $r_1, r_2, r_4, r_5, r_6, r_7, r_8, r_9$.



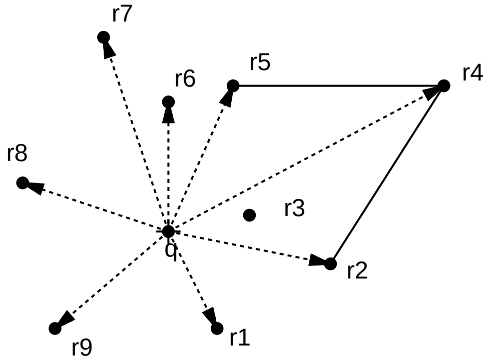
- Step 3:

- $*$ Current $= r_1$.
- $*$ r_1, r_2, r_4 is a left turn.
  $\Rightarrow$ advance Current.
- $*$ Current $:=$ r_2.
- $*$ List: $r_1, r_2, r_4, r_5, r_6, r_7, r_8, r_9$.



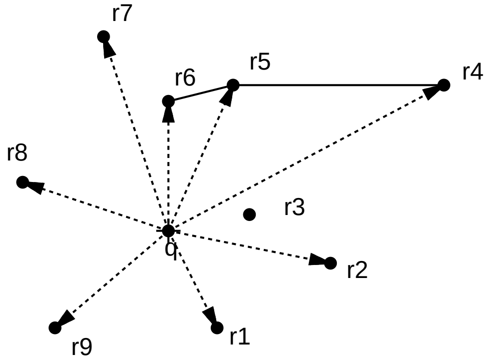
– Step 4:

- ∗ Current = r_2.
- ∗ r_2, r_4, r_5 is a left turn.
  ⇒ advance Current.
- ∗ Current := r_4.
- ∗ List: $r_1, r_2, r_4, r_5, r_6, r_7, r_8, r_9$.



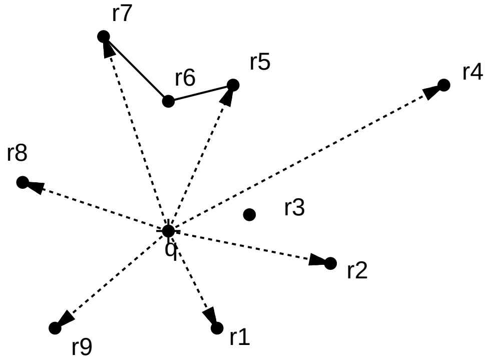
– Step 5:

- ∗ Current = r_4.
- ∗ r_4, r_5, r_6 is a left turn.
  ⇒ advance Current.
- ∗ Current := r_5.
- ∗ List: $r_1, r_2, r_4, r_5, r_6, r_7, r_8, r_9$.

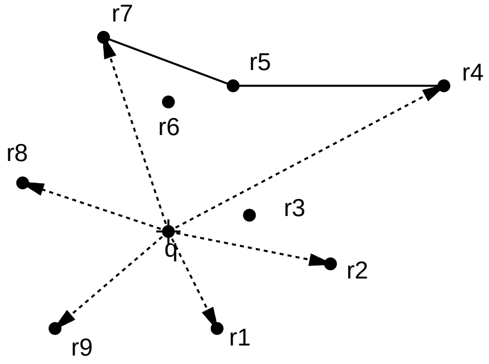


– Step 6:

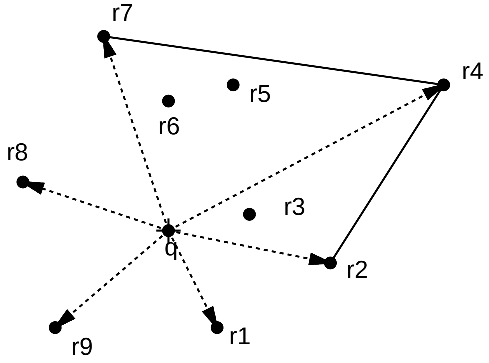
- ∗ Current = r_5.
- ∗ r_5, r_6, r_7 is a right turn.
  ⇒ delete r_6.
- ∗ Current := r_4.
- ∗ List: $r_1, r_2, r_4, r_5, r_7, r_8, r_9$.



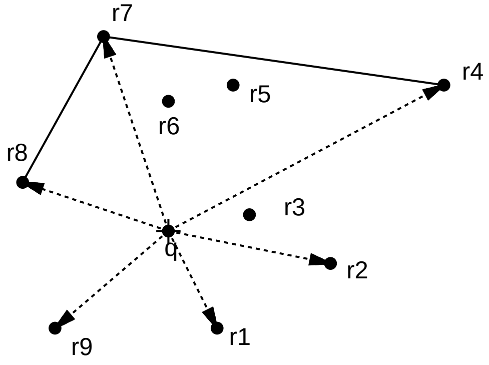
- Step 7:
  - ∗ Current = r_4.
  - ∗ r_4, r_5, r_7 is a right turn.
    ⇒ delete r_5.
  - ∗ Current := r_2.
  - ∗ List: $r_1, r_2, r_4, r_7, r_8, r_9$.



- Step 8:
  - ∗ Current = r_2.
  - ∗ r_2, r_4, r_7 is a left turn.
    ⇒ advance Current
  - ∗ Current := r_4.
  - ∗ List: $r_1, r_2, r_4, r_7, r_8, r_9$.

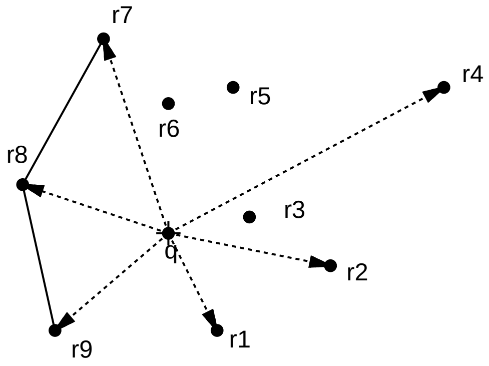


- Step 9:
  - ∗ Current = r_4.
  - ∗ r_4, r_7, r_8 is a left turn.
    ⇒ advance Current
  - ∗ Current := r_7.
  - ∗ List: $r_1, r_2, r_4, r_7, r_8, r_9$.



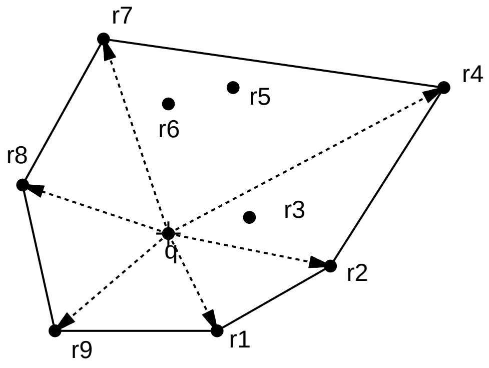
- Step 10:

  - ∗ Current = r_7.
  - ∗ r_7, r_8, r_9 is a left turn.
    $\Rightarrow$ advance Current
  - ∗ Current := r_8.
  - ∗ List: $r_1, r_2, r_4, r_7, r_8, r_9$.



- Step 11:

  - ∗ Current = r_8.
  - ∗ r_8, r_9, r_1 is a left turn.
    $\Rightarrow$ stop.
  - ∗ List: $r_1, r_2, r_4, r_7, r_8, r_9$.



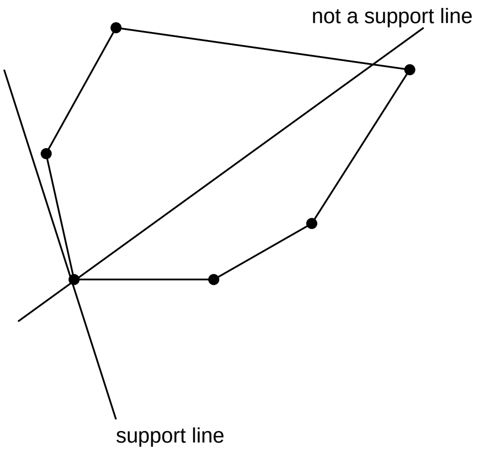
Hull: r1,r2,r4,r7,r8,r9

- Complexity of scan:
  - At most n deletes.
  - At most n advances of Current.
  - $O(1)$ work for checking left or right turn.
    $\Rightarrow O(n)$ for scan.
- Overall complexity of Graham's scan:
  - $O(n)$ for finding internal point q.
  - $O(n \log n)$ for sorting by angle.
  - $O(n)$ for scan.
    $\Rightarrow O(n \log n)$ overall.
- Note: it can be shown that $O(n \log n)$ is a lower bound
  $\Rightarrow$ Graham's scan is optimal.
- Note: Graham's scan does not generalize to 3D. Other fast algorithms, such as Quickhull (divide and conquer), do generalize.

12.11 Diameter of a Convex Polygon

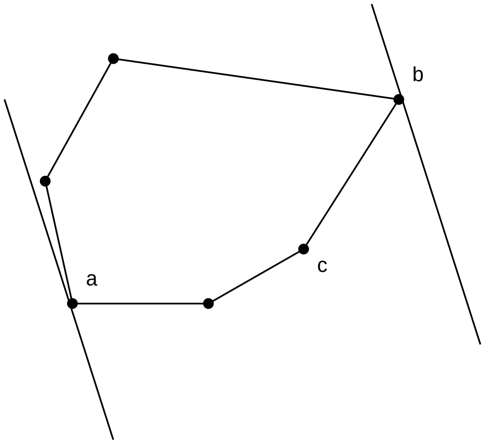
- Definition: L is a support line of polygon P if L passes through a vertex of P but not its interior.

  Example:



- Points a and b of a polygon P are called *antipodal* if there exist parallel support lines through a and b

  Example:

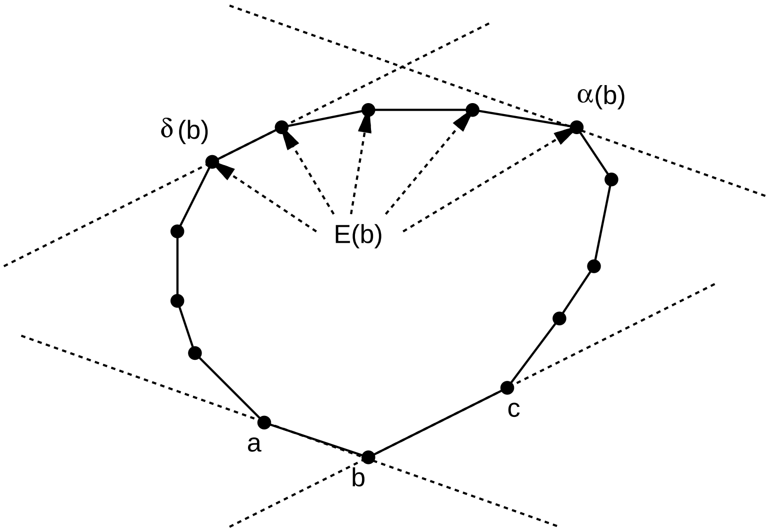


Here a and b are antipodal, but a and c are not.

- Observation: the diameter of P is the largest distance among antipodal pairs.

  Intuition: think of calipers.

- Some definitions:
  - Consider some vertex $b \in P$ with incident edges (a, b) and (b, c).



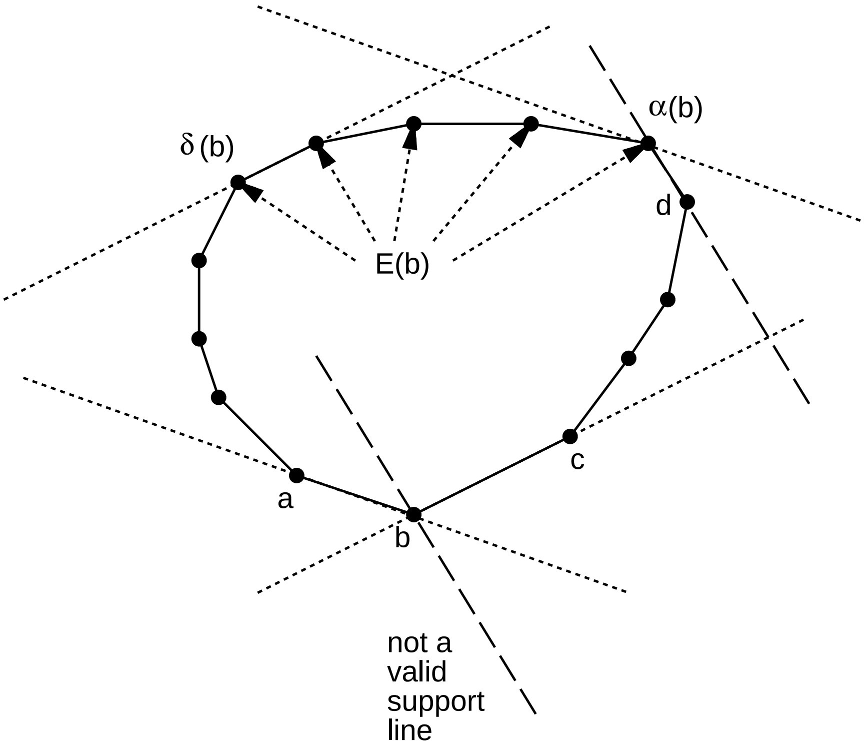
  - Draw infinite lines through edges (a, b) and (b, c).
  - Call these lines the ab and bc lines respectively.
  - Define $\alpha(b)$:
    * The vertex farthest away from the ab line (perpendicular distance).
    * If there are several such vertices, pick the one first encountered in a counter-clockwise walk from b.
  - Define $\delta(b)$:
    * The vertex farthest away from the bc line (perpendicular distance).
    * If there are several such vertices, pick the one first encountered in a clockwise walk from b.

- Define $E(b)$ as the set of all vertices encountered in a walk from $\alpha(b)$ to $\delta(b)$ (endpoints included).

- Observation: for each point $s \in E(b)$, b and s are antipodal.

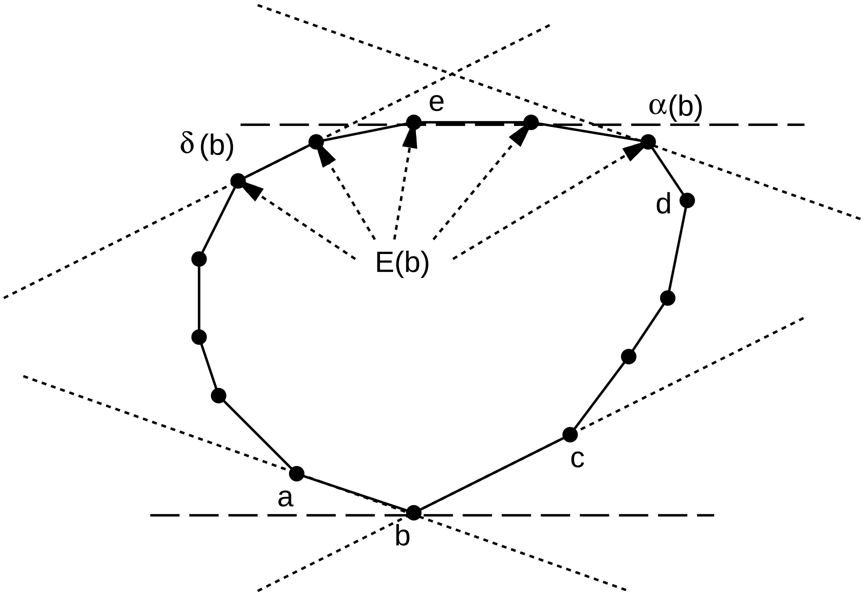
  Why?

  - Consider rotating the ab line *counter-clockwise* about the point b towards the bc line.

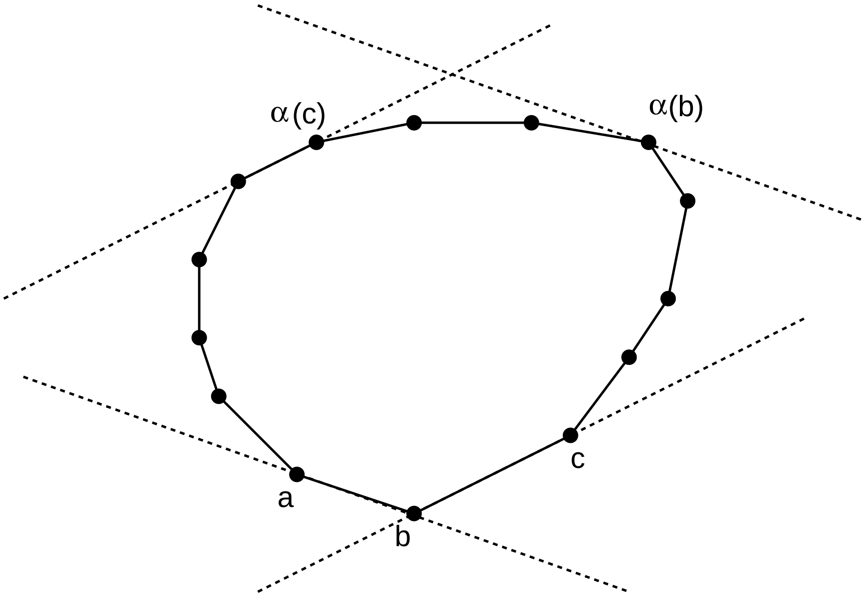


  - All intermediate positions in this rotation cannot be support lines through b.
  - Thus, any support line through a point d before $\alpha(b)$ cannot have a parallel support line through b.
  - A similar argument holds for $\delta(b)$.

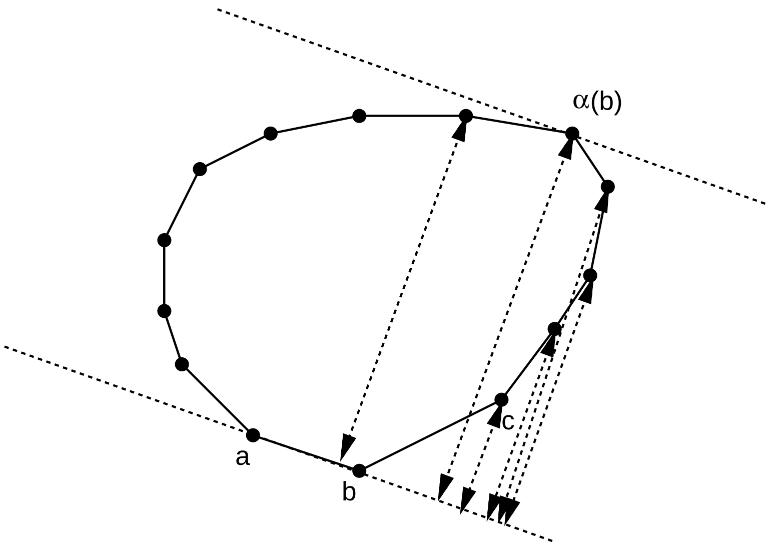
– Finally, any support line of a point $e \in E(b)$, has a parallel support line through b:



- Another observation: $\alpha(c) \geq \alpha(b)$, i.e., for the next point past b (i.e., c), the start of $E(c)$ occurs past the start of $E(b)$.



- Another observation: in finding $\alpha(b)$
  - we scan vertices in counter-clockwise order;
  - the distances increase monotonically, and then decrease monotonically
    $\Rightarrow$ we stop before the first decrease occurs.

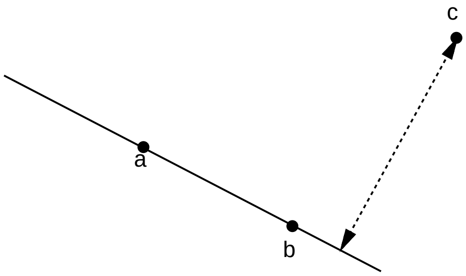


  - Note: this property is due to convexity.

- An $O(n)$ algorithm to generate all antipodal pairs:
  - Start with any vertex v_1 and find $\alpha(v_1), \delta(v_1)$.
    $\Rightarrow O(n)$ worst-case.
  - Scan $E(v_1)$ as it's being created (and check distances).
  - Find antipodal range $E(v_i)$ for successive points v_i.
    * We only need search past $\alpha(v_{i-1})$ to look for $\alpha(v_i)$.
    * We only need search past $\delta(v_{i-1})$ to look for $\delta(v_i)$.

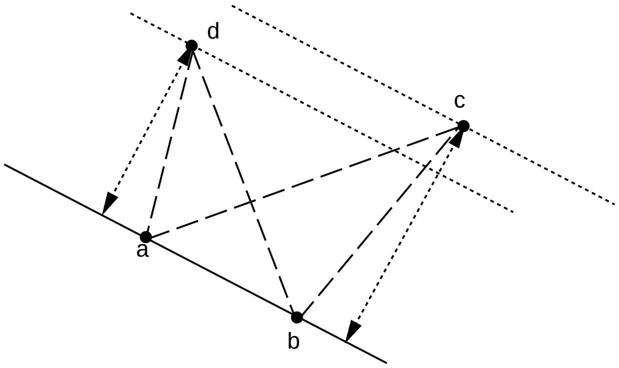
    $\Rightarrow$ only one scan needed
    $\Rightarrow O(n)$ time.
  - While creating antipodal pairs, record maximum distance.

- Details: Finding the farthest vertex from a line:
  - One approach: use standard coordinate geometry to compute distance of a point from a line



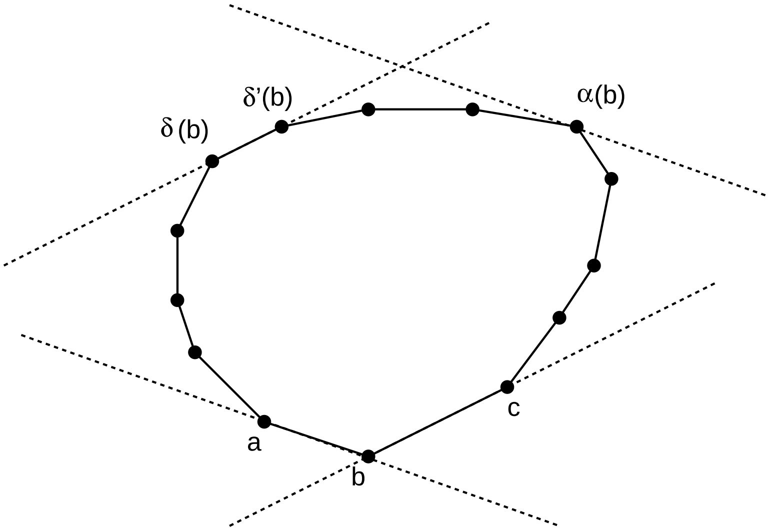
    Pick vertex with largest distance.

  - Better approach:
    * We only need to *compare* distances, rather than *compute* them.
    * Distances can be compared by comparing areas:



    * Since area = $\frac{1}{2}$ base × height and base = ab, an area comparison is really a height comparison.

- Define $\delta'(b)$:
  - The vertex farthest away from the bc line.
  - If there are several, pick the one first encountered in a *counter-clockwise* direction.



- Note:
  - the distance of $\delta(b)$ from the bc line is the same as the distance of $\delta'(b)$

    $\Rightarrow$ we don't have to search past $\delta'(b)$ in generating antipodal distances.
  - $\delta'(b) = \alpha(c)$

    $\Rightarrow$ start search for $E(c)$ where $E(b)$ ended.
- Summary of set diameter computation:
  - Find convex hull of given set S

    $\Rightarrow O(n \log n)$ time.
  - Generate antipodal pairs and check distances

    $\Rightarrow O(n)$ time.

  $\Rightarrow O(n \log n)$ time overall.

- Recall: the *All-Pairs Method* took $O(n^2)$.

12.12 Data Mining: An Introduction

- What is data mining?
  - **Def**: Data mining is the process of extracting hidden patterns or trends in large data sets for the purpose of prediction.
  - The basic idea: comb through available data, looking for unusual unobvious patterns and report them.
- Why is data mining important?
  - Businesses are interested in exploiting knowledge about patterns.
  - Standard statistical techniques (multivariate analysis) work only on numeric data and with few variables.
- Examples of applications:
  - *Banking.*
    - Suppose a bank sifted through its archives and discovered the following statistic:
      "77 percent of loan defaults involved (1) a customer in the age group 18-21, (2) a car loan for a red sportscar and (3) income group $15,000-$20,000".
    - The bank can use this pattern to avoid giving loans.
    - Similarly, some unobvious patterns can indicate likely attributes of a "good" customer.
    - Today, many banks (e.g., Citibank, Signet) use information extracted by data mining algorithms.

- *Sports.*
    - Suppose a basketball team (e.g., Chicago Bulls) sifted through their records and discovered the following: "On 60 percent of plays in which Scottie Pippen is defended by the opposing guard, the Bulls eventually win the possession."
    - The coach can use this pattern to improve chances of winning.
    - Today, several NBA teams use Advanced Scout, a data mining package to produce such statistics.
- *Retail industry.*
    - By sifting through customer purchase data, a grocery store discovers that "40 percent of customers that buy wine also buy a specialty cheese".
    - The store can use the information in marketing and display strategies.

- Several types of data mining:
    - *Association rule mining*: Find "rules" of the sort "77 percent of loan defaults with attributes A,B,C also have attributes X and Y".
    - *Clustering*: Find groupings of data based on available attributes based on the structure of the data, e.g., "90 percent of renters in the Williamsburg area fall into either the 18-25 or 65-75 age groups".
    - *Classification*: Find natural groupings of data based on available attributes that seek to predict an outcome. e.g. group bank customers into three groups: (1) "most-likely to repay"; (2) "most-likely to default" and (3) "don't know".
    - *Other*: finding patterns in sequences (Stock Market application), deviation detection (Fraud detection application).

- Types of data sets:
  - Most data sets are large relational tables, with many attributes, e.g., bank customers may collect 50-100 attributes on a loan application.
  - Some data sets are unnormalized "basket" data, such as the list of items checked out by each customer at a grocery store.
- Why data mining is an interesting problem:
  - Typical data sets are very large with many attributes, e.g.,
    - Census data: about 400 attributes per individual.
    - Retail store data: millions of transactions, thousands of attribute types.
  - A naive approach of trying all possible rules causes a combinatorial explosion, e.g,
    - Consider a relation $R(A_1, A_2, \ldots, A_{100})$ where each attribute value is boolean.
    - Suppose we are interested in generating rules of the sort $A_{i_1}A_{i_2}\ldots A_{i_k} \rightarrow A_{j_1}A_{j_2}\ldots A_{j_m}$ e.g., of the 100 records with $A_3A_4A_7$ true, 68 of them also have A_1A_8 true.
    - Consider all possible combinations of $A_{i_1}A_{i_2}\ldots A_{i_k} \rightarrow A_{j_1}A_{j_2}\ldots A_{j_m}$.
    - For each such combination, scan relation R to count percentages.

12.13 Association Rule Mining: Introduction

- Consider data collected at a supermarket checkout counter:
  - The system records customer purchases in a variable-size record (unnormalized), e.g.

    r_{257}=<Eggs, bread, pasta, milk, cheese, beer, soap>.

    (Customer 257 bought eggs, bread etc).
  - The system has thousands of such customer purchase-records each day.
  - An *association rule* seeks to answer questions like:
    when pasta and pasta sauce are bought, what is the probability that mushrooms are also purchased?
  - In terms of available data, this question can be rephrased as:
    among those records that contain both pasta and pasta sauce, how many also contain mushrooms?
  - Why is this question useful?
    The answer (if high) can drive pricing and display strategies
    ⇒ package discount for the combination of pasta and mushrooms.
  - Suppose our data has 100,000 records, of which
    - 30,000 records contain pasta and pasta sauce;
    - 22,500 of these 30,000 records contain mushrooms.

    Then, we have the *association rule*

    $$\{\text{pasta, pasta sauce}\} \rightarrow \{\text{mushrooms}\}$$

    with

    $$\begin{array}{lclcl} \text{support} & = & 22{,}500/100{,}000 & = & 0.225 \\ \text{confidence} & = & 22{,}500/30{,}000 & = & 0.75 \end{array}$$

  - Intuition: 75 percent of the time when pasta and pasta sauce are bought, mushrooms are also bought.

- Both *support* and *confidence* are important:
  - Consider the rule

    $$\{\text{Non-alcoholic beer}\} \rightarrow \{\text{chips}\}.$$

  - Suppose

    $$\begin{array}{lcl} \text{confidence} & = & 0.9 \\ \text{support} & = & 0.001 \end{array}$$

  - Thus, 90 percent of customers that buy non-alcoholic beer also buy chips.
  - But, this pattern occurs only in 0.1 percent of the data
    ⇒ not an important rule.

- **Def**: an association rule $X \rightarrow Y$, where X and Y are sets of attributes, satisfies confidence level c and support s if:
  1. the actual confidence is at least c and
  2. the actual support is at least s.

- The **association rule mining problem**: *given a confidence level c and a support level s, find* **all** *rules that satisfy c and s.*

12.14 Association Rule Mining: Problem Formulation

- Notation:
  - Let $I = \{I_1, I_2, \ldots, I_m\}$ be a set of items.

    Think of I as {eggs,cheese,pasta,...} in the supermarket example. (Set of all possible supermarket products).
  - A subset of items $X \subseteq I$ will sometimes be called an itemgroup.
  - We will use letters like X and Y to denote itemgroups.
  - Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of unnormalized records (basket data).

    Here, r_i is the set of items bought by customer i,
    e.g. r_{257} ={pasta, pasta sauce, tomatoes, beef, soap}
    Thus, $\forall i : r_i \subseteq I$.
  - **Def**: a record $r \in R$ *contains* itemgroup X if $X \subseteq r$.
  - Let $R(X) = \{r \in R : r \text{ contains } X\}$.
  - For any itemgroup X, let $\alpha(X) = |R(X)|$, the number of records that contain X.
  - For any itemgroup X, define the support of X to be

    $$\beta(X) = \frac{\alpha(X)}{|R|}.$$
  - Define

    $$F_s(R) = \{X \subseteq I : \beta(X) = \frac{\alpha(X)}{|R|} \geq s\}.$$

    (All the itemgroups satisfying support s).

  - **Def**: A rule $X \rightarrow Y$ is an *association rule* satisfying support s and confidence c if

    1. X and Y are itemgroups (i.e., $X, Y \subseteq I$).
    2. X and Y are disjoint (i.e., $X \cap Y = \emptyset$).
    3. At least s fraction of records contain both X and Y, i.e.,

    $$\beta(X \cup Y) = \frac{\alpha(X \cup Y)}{|R|} \geq s.$$

    4. Of those records containing X, at least c fraction contain Y, i.e.,

    $$\frac{\alpha(X \cup Y)}{\alpha(X)} \geq c.$$

- Example: $I = \{$milk, eggs, pasta, pasta sauce, cheese$\}$
  R is given by:

$$\begin{aligned}
r_1 &= \text{<milk, eggs>} \\
r_2 &= \text{<milk, eggs>} \\
r_3 &= \text{<milk, eggs, cheese>} \\
r_4 &= \text{<milk, pasta, cheese>} \\
r_5 &= \text{<eggs, pasta sauce, cheese>} \\
r_6 &= \text{<pasta, pasta sauce>} \\
r_7 &= \text{<pasta, pasta sauce, cheese>} \\
r_8 &= \text{<pasta, pasta sauce, cheese>} \\
r_9 &= \text{<milk, eggs, pasta, pasta sauce, cheese>} \\
r_{10} &= \text{<milk, pasta, pasta sauce, cheese>}
\end{aligned}$$

  - Consider X=$\{$eggs, pasta sauce$\}$. Then,

$$\begin{aligned}
R(X) &= \{r_5, r_9\} \\
\alpha(X) &= 2 \\
\beta(X) &= \frac{2}{10} = 0.2
\end{aligned}$$

– Consider X={milk, eggs}.

$$\begin{aligned} R(X) &= \{r_1, r_2, r_3, r_9\} \\ \alpha(X) &= 4 \\ \beta(X) &= \frac{4}{10} = 0.4 \end{aligned}$$

– Consider X={milk,eggs} and Y={eggs}
  ⇒ not a valid rule since $X \cap Y \neq \emptyset$.

– $X \rightarrow Y$ is a potential association rule where X={milk} and Y={eggs,pasta,cheese}.

– Consider X={milk,eggs} and Y={cheese}. Then

$$\begin{aligned} |R| &= 10 \\ \alpha(X) &= 4 \\ \alpha(X \cup Y) &= 2 \\ \beta(X \cup Y) &= \frac{2}{10} = 0.2 \end{aligned}$$

Hence

$$\begin{aligned} \text{support} &= \beta(X \cup Y) = \frac{\alpha(X \cup Y)}{|R|} = \frac{2}{10} = 0.2 \\ \text{confidence} &= \frac{\alpha(X \cup Y)}{\alpha(X)} = \frac{2}{4} = 0.5 \end{aligned}$$

Thus, $X \rightarrow Y$ with support 0.2 and confidence 0.5.

- Suppose $s = 0.3$. Since $|R| = 10$, we want all itemgroups that appear in at least 3 records, i.e.,

$$F_{0.3}(R) = \{X \subseteq I : \beta(X) = \frac{\alpha(X)}{|R|} \geq 0.3\}.$$

  Here, $F_{0.3}(R)$ ={ {milk}, {eggs}, {pasta}, {pasta sauce}, {cheese}, {milk,eggs}, {pasta, pasta sauce}, {milk,pasta}, {milk,cheese}, {pasta,cheese}, {pasta,pasta sauce,cheese}, {milk,eggs,cheese}, {milk,pasta,cheese} }.

- **Def**: A rule $X \to Y$ is a 1-RHS rule if $|Y| = 1$.
  (Right-hand side has only one item).

- Typical restriction on problem: find all 1-RHS association rules (satisfying given s and c).

  Examples of 1-RHS rules from above:

$$\begin{array}{lcl}
\{\text{milk}\} & \to & \{\text{eggs}\} \\
\{\text{eggs}\} & \to & \{\text{milk}\} \\
\{\text{pasta}\} & \to & \{\text{cheese}\} \\
\{\text{pasta,cheese}\} & \to & \{\text{milk}\} \\
\{\text{milk,eggs}\} & \to & \{\text{cheese}\}
\end{array}$$

  Note that

$$\{\text{milk}\} \to \{\text{eggs,cheese}\}$$

  is a rule but not a 1-RHS rule.

- An observation:
  - Suppose we have computed $\beta(X)$ (support) for each possible itemgroup X.
  - Consider a rule $X \rightarrow Y$. Then,

$$\begin{aligned}\frac{\beta(X \cup Y)}{\beta(X)} &= \frac{\alpha(X \cup Y)/|R|}{\alpha(X)/|R|} \\ &= \frac{\alpha(X \cup Y)}{\alpha(X)} \\ &= \text{confidence of rule } X \rightarrow Y\end{aligned}$$

  - Thus, given only support numbers for itemgroups, we can compute rule confidences.
  - Also, for a rule $X \rightarrow Y$ to meet the required support s, we must have $\beta(X \cup Y) \geq s$
    $\Rightarrow X \cup Y \in F_s(R)$.
  - Note that $\beta(X \cup Y) \geq s \Rightarrow \beta(X) \geq s$.
    $\Rightarrow X, X \cup Y \in F_s(R)$.
    $\Rightarrow$ The association rule mining problem reduces to finding itemgroups with large enough support, i.e., computing $F_s(R)$.
  - Thus, for the remainder we will focus on simply identifying $F_s(R)$, the set of itemgroups with large enough support.

    Typically, we will want to output each itemgroup and its actual support.

12.15 Two Naive Algorithms

- **Algorithm**: Naive-1 (R, I, s)
  - Generate all possible itemgroups and initialize a counter for each.

    Note: All possible itemgroups = 2^I = all possible subsets of I.
  - Scan R once and count support for each itemgroup.
  - Output those itemgroups satisfying s.
- Analysis of Naive-1:
  - How many possible itemgroups with $I = \{I_1, \ldots, I_m\}$?
    $\Rightarrow \left|2^I\right| = 2^{|I|} = 2^m$
  - If m is large (say, $m > 100$), 2^m is too big for main memory.
  - Also, if $|F_s(R)|$ is small, we waste time updating counts for itemgroups not in $F_s(R)$.
- **Algorithm**: Naive-2 (R, I, s)

```
– while not over do
    ∗ Generate a new itemgroup.
    ∗ Scan $R$ to obtain support.
    ∗ if support $\geq s$, retain itemgroup.
– endwhile
– Output all itemgroups retained.
```

- Analysis of Naive-2:
  - Too many scans of the data.

- **Key observation**: $X \notin F_s(R) \Rightarrow X \cup Y \notin F_s(R)$ for any Y.
  (If itemgroup X does not satisfy s, neither will any extension of X such as $X \cup Y$)

  Example: if {milk} occurs in only 0.1 fraction of records, then {milk,eggs} occurs in no more than 0.1 fraction of records.

  This observation is used in better algorithms.

- We will use the following example for illustration:

$$
\begin{aligned}
I &= \{A, B, C, D, E\} \\
r_1 &= < A, B > \\
r_2 &= < A, B > \\
r_3 &= < A, B, E > \\
r_4 &= < A, C, E > \\
r_5 &= < B, D, E > \\
r_6 &= < C, D > \\
r_7 &= < C, D, E > \\
r_8 &= < C, D, E > \\
r_9 &= < A, B, C, D, E > \\
r_{10} &= < A, C, D, E >
\end{aligned}
$$

12.16 Algorithm Record-Derived-Itemgroups

- Key ideas:
  - Consider the itemgroup ABD and the item E. If ABD has poor support then so does $ABDE$, the extension of ABD to E.

    Thus, $\beta(ABD) < s \Rightarrow \beta(ABDE) < s$.
  - We will assume the items are lexicographically ordered.

    Thus, we will extend ACF to $ACFG$ but not $ACDF$.

    (Because $ACDF$ will be considered when ACD is extended).
  - The support for a tentative extension can be estimated using independence:

    $$\hat{\beta}(ABDE) = \beta(ABD)\beta(E).$$

    For example, if $\beta(ABD) = 0.4$ (40 percent of R) and $\beta(E) = 0.6$ is known from previous iterations, then

    $$\hat{\beta}(ABDE) = 0.4 \times 0.6 = 0.24.$$

    Of course, independence may turn out to be a poor approximation.
  - The algorithm makes multiple scans of data. In each scan:
    - Counts are maintained for various itemgroups.
    - At the end of each scan, itemgroups with low support are discarded.
    - As each record is encountered, the items within it are used to create new potential itemgroups.
    - If the estimated support is high, additional extensions are considered.
    - If the estimated support is low, an itemgroup is placed in a Next_Frontier set (to be re-examined at the end of the pass).

- Generally, if an itemgroup was mistakenly placed in the Next_Frontier set (support underestimated), it's count will actually be high, and therefore is considered for extension later.
- If an itemgroup was mistakenly extended too much (support overestimated), it will be discarded at the end of the scan.
- Example: consider the itemgroup A, B and the record $\{A, B, D, E, F\}$.
  - ∗ Itemgroup AB can be extended to create the following potential itemgroups: ABD, ABE, ABF, $ABDE$, $ABDF$, $ABEF$ and $ABDEF$.
  - ∗ Suppose it turns out

$$\begin{array}{rcl} \hat{\beta}(ABD) & \geq & s \\ \hat{\beta}(ABE) & < & s \\ \hat{\beta}(ABF) & \geq & s \end{array}$$

    Then,

    - · ABD can be extended to the next size ($ABDE$ or $ABDF$) lexicographically.
    - · ABE is not extended (and placed in Next_Frontier).
    - · ABF cannot be extended because the record has nothing beyond F.
    - · Since ABD got extended to $ABDE$, we consider expanding $ABDE$ to $ABDEF$ (if the estimated support is good).
- Why are low-estimate itemgroups kept around in Next_Frontier?
  ⇒ need to compute counts in case estimate was bad
  ⇒ they may still satisfy s.

- Pseudocode:
  - Note: A first pass is done separately to initialize counts for the 1-item itemgroups.
  - Assume that the set of items is $I = \{I_1, \ldots, I_m\}$.

Algorithm: Record-Derived-Itemsets (R, I, s)

Input: Set of records R, set of items I, support s.
Output: Collection of itemgroups with large enough support.

```
Large := $\emptyset$;
// First pass
$\forall k :\ \alpha[I_k] := 0$ // Initialize counts
**for** $j := 1$ **to** $|R|$ **do**
   **for** $k := 1$ **to** $m$ **do**
      **if** $I_k \in r_j$
         $\alpha[I_k] := \alpha[I_k] + 1$;
**for** $k := 1$ **to** $m$
   $\beta[I_k] := \alpha[I_k]/|R|$;
   **if** $\beta[I_k] \geq s$
      Large := Large $\cup \{I_k\}$;
**endfor**;
// All other passes.
Frontier := $I$; // Keep Frontier sorted by size.
**while** Frontier $\neq \emptyset$
   $H := \emptyset$;
   **for** $j := 1$ **to** $|R|$ // Scan data.
      **for each** itemgroup $X \in$ Frontier
         **if** $X \in$ record $r_j$
            $G :=$ Compute-Extensions $(X, I, r)$;
         **for each** $Y \in G$
            **if** $Y \in H$
               $Y$.count := $Y$.count + 1;
            **else**
               $H := H \cup \{Y\}$;
               $Y$.count := 1;
            **endif**;
         **endfor**;
      **endfor**;
   **endfor**;
... continued
```

Algorithm: Record-Derived-Itemsets ... continued

```
        // Identify itemgroups that satisfy $s$.
    for each $Y \in H$
        if $Y$.count/$|R| \geq s$
            Large := Large $\cup \{Y\}$;
// Set next frontier to be considered for extension
    Frontier := Next_Frontier $\cap$ Large;
endwhile;
return Large;
```

```
Algorithm:    COMPUTE-EXTENSIONS ($X$, $I$, $r$)

Input: an itemgroup $X$, the set of items $I$, record $r$.
Output: Extensions of $X$.
  $k$ := $|X|$;
  $G'$ := $\{X\}$;
  repeat
     No_change := true;
     // Compute extensions for size $k$.
     for each $Y \in G'$ such that $|Y| = k$
        // Suppose $Y = I_{j_1} I_{j_2} \ldots I_{j_k}$.
        for $l$ := $j_k + 1$ to $m$
           if $I_l \in$ record $r$
              $Z$ := $Y \cup \{I_l\}$;
              // See if $Z$ is worth the trouble.
              $\hat{\beta}[Z]$ := $\hat{\beta}[Y] * \beta[I_l]$;
              if $\hat{\beta}[Z] \geq s$
                 $G'$ := $G' \cup \{Y\}$;
                 No_change := false;
              else
                 Next_Frontier := Next_Frontier $\cup$ $Y$;
              endif;
           endif;
        endfor;
     endfor;
     $k$ := $k + 1$;
  until No_change or $k = m$;
  return $G$ := $G' \cup$Next_Frontier;
```

- Example: $s = 0.3$
  - First pass: $\alpha(A) = 6$, $\alpha(B) = 5$, $\alpha(C) = 6$, $\alpha(D) = 6$, $\alpha(E) = 7$.

$\beta(A) = 0.6$, $\beta(B) = 0.5$, $\beta(C) = 0.6$, $\beta(D) = 0.6$, $\beta(E) = 0.7$.
Frontier = $\{A, B, C, D, E\}$

– Second pass:

1. When $r_1 =< A, B >$ is scanned:
   - ∗ The only possible extension is AB (BA is not considered because it is not a lexicographic extension).
   - ∗ $\hat{\beta}(AB) = \hat{\beta}(A)\hat{\beta}(B) = 0.3$.
     $\Rightarrow AB.\text{count} \ := \ 1$.
   - ∗ $H = \{AB\}$.
2. When $r_2 =< A, B >$ is scanned:
   - ∗ Only extension possible is AB.
   - ∗ AB.count = 2, $H = \{AB\}$.
3. When $r_3 =< A, B, E >$ is scanned:
   - ∗ Extensions (with support):
     $G = \{AB(0.3), AE(0.42), BE(0.35), ABE(0.21)\}$.
   - ∗ AB.count=3, BE.count=1, ABE.count=1.
   - ∗ $H = \{AB, AE, BE, ABE\}$.
   - ∗ Next_Frontier=$\{ABE\}$ (it's estimate was not high enough).
4. When $r_4 =< A, C, E >$ is scanned:
   - ∗ $G = \{AC(0.36), AE(0.42), ACE(0.252), CE(0.42)\}$.
   - ∗ Counts: $\boldsymbol{AB(3)}, \boldsymbol{AC(1)}, \boldsymbol{AE(2)}, ABE(1), \boldsymbol{ACE(1)}, BE(1), \boldsymbol{CE(1)}$.
   - ∗ Next_Frontier=$\{ABE, ACE\}$.
5. When $r_5 =< B, D, E >$ is scanned:
   - ∗ $G = \{BD(0.3), BDE(0.21),$
     $BE(0.35), DE(0.42)\}$.
   - ∗ Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), \boldsymbol{BD(1)}, \boldsymbol{BE(2)}$,
     $\boldsymbol{BDE(1)}, CE(1), \boldsymbol{DE(1)}$.
   - ∗ Next_Frontier=$\{ABE, ACE, ADE, BDE\}$.
6. When $r_6 =< C, D >$ is scanned:

- $*$ $G = \{CD(0.36)\}$.
- $*$ Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), BD(1), BE(2), BDE(1), \boldsymbol{CD(1)}, \boldsymbol{CE(1)}, DE(1)$.
- $*$ Next_Frontier=$\{ABE, ACE, ADE, BDE\}$. (Unchanged).

7. When $r_7 =< C, D, E >$ is scanned:
   - $*$ $G = \{CD(0.36), CE(0.42), CDE(0.252), DE(0.42)\}$.
   - $*$ Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), BD(1), BE(2), BDE(1), \boldsymbol{CD(2)}, \boldsymbol{CE(2)}, \boldsymbol{CDE(1)}, \boldsymbol{DE(2)}\}$.
   - $*$ Next_Frontier=$\{ABE, ACE, ADE, BDE, CDE\}$.
8. When $r_8 =< C, D, E >$ is scanned:
   - $*$ $G = \{CD(0.36), CE(0.42), CDE(0.252), DE(0.42)\}$.
   - $*$ Counts: $AB(3), AC(1), AE(2), ABE(1), ACE(1), BD(1), BE(2), BDE(1), \boldsymbol{CD(3)}, \boldsymbol{CE(3)}, \boldsymbol{CDE(2)}, \boldsymbol{DE(3)}\}$.

Continuing, the large itemgroups turn out to be:

Large = $\{A, B, C, D, E, AB, AC, AE, ACE, BE, CD, CE, CDE, DE\}$.

– Third pass:

- $*$ Here, the size 3 itemgroups are $\{ACE, CDE\}$.
  They were expected-small but turned out to have enough support.
- $*$ These can't be expanded (they end in E)
  $\Rightarrow$ we're done.

– Final result:

$$A, B, C, D, E, AB, AC, AE, ACE, BE, CD, CE, CDE, DE.$$

12.17 Algorithm Pass-Derived-Itemgroups

- In the previous algorithm, the following problem arises:
  - Suppose a record has items $< A, B, C, D, E, F >$ and suppose that $\{AC, AD, CD, CE\}$ are in the current Frontier.
  - Potential extensions include $\{ACF, ADF, CDF, CEF\}$.
  - If all of them have small expectations, we're still going to maintain counts for them
    $\Rightarrow$ the algorithm wastes time counting useless itemgroups.
  - We will try to minimize this problem in the next algorithm.
- Key ideas in Algorithm Pass-Derived-Itemgroups:
  - In pass k only itemgroups of size k are considered.
  - At the end of pass $k-1$, we will have counts for the large itemgroups of size $k-1$
    $\Rightarrow$ we know L_{k-1}=\{large itemgroups of size $k-1$\}.
  - Before starting pass k, we compute all possible itemgroups of size k.
  - Example:
    * Suppose in pass $k = 5$ we generate $ACDEF$ has a potential itemgroup.
    * We consider all possible $(k-1)$-size subgroups, such as $ACEF$ and $ADEF$.
    * If any of these subgroups is not in L_{k-1}, we can reject $ACDEF$ immediately.
  - Another idea used is to generate potential groups in an intelligent way (exploiting lexicographic order):
    * Suppose $k = 5$ and $L_4 = \{ABCD, ABCE, BCDE, BCEF\}$.

        - The naive way to generate size-5 itemgroups would be to consider all possible extensions of the above four itemgroups.
        - Note instead, that the itemgroup $ABCDE$ will be large only if $ABCD$ and $ABCE$ are already large,
          i.e., only if both $ABCD$ and $ABCE$ are in L_4.
          $\Rightarrow$ we will allow $ABCDE$ to be generated only from $ABCD$ and $ABCE$.
        - In general, we will generate $X_1X_2\ldots X_{k-1}X_k$ from $X_1X_2\ldots X_{k-2}X_{k-1}$ and $X_1X_2\ldots X_{k-2}X_k$.
        - Interestingly, this "combining" can be stated as a join:
            - Note that L_{k-1} is a collection of size-$(k-1)$ itemgroups.
            - Suppose that each itemgroup is considered a tuple in a relation called L_{k-1}, where the i-th attribute in a tuple is the i-th item in the itemgroup.
            - Example ($k = 5$): the tuple for $ABCE$ will be the tuple $< A, B, C, E >$.
            - Suppose the attribute names are item$_1$,..., item$_{k-1}$.
            - The join statement is:

```
select  $p$.item$_1$, ..., $p$.item$_{k-1}$, $q$.item$_{k-1}$
from    $L_{k-1}$ as $p$, $L_{k-1}$ as $q$
where   $p$.item$_1$ = $q$.item$_1$
        ...
        and $p$.item$_{k-2}$ = $q$.item$_{k-2}$
        and $p$.item$_{k-1}$ < $q$.item$_{k-1}$
```

    - One additional observation:
        - Consider $k = 6$ and suppose the join resulted in $ABCDEF$.
        - We now need to look at all possible subgroups (of size $k-1$) of $ABCDEF$.
        - How many possible subgroups are there?
          $\Rightarrow$ at most 6 (drop one letter at a time for each size 5 string).

- Pseudocode:

Algorithm: PASS-DERIVED-ITEMGROUPS (R, I, s)

Input: Set of records R, set of items I, support s.
Output: Collection of itemgroups with large enough support.

```
// First pass
$\forall k: \ \alpha[I_k] := 0$ // Initialize counts
for $j := 1$ to $|R|$ do
    for $k := 1$ to $m$ do
        if $I_k \in r_j$
            $\alpha[I_k] := \alpha[I_k] + 1$;
// $L[1] :=$ BUILD-SET ($\emptyset$);
for $k := 1$ to $m$
    if $\alpha[I_k]/|R| \geq s$
        $L_1 := L_1 \cup \{I_k\}$; // ADD-SET ($L[1], I_k$);
// All other passes.
$k := 2$;
while $L_{k-1} \neq \emptyset$ // SET-NOT-EMPTY ($L[1]$);
    $C :=$ COMPUTE-JOIN ($L_{k-1}, L_{k-1}$);
    // $C :=$ BUILD-SET (COMPUTE-JOIN ($L_{k-1}, L_{k-1}$));
    for each itemgroup $X = I_{j_1} \ldots I_{j_k} \in C$ // Winnowing
        $l := 1$;     over := false;
        while $l \leq k - 2$ and not over
            if $Y = I_{j_1} \ldots I_{j_{j-1}} I_{j_{l+1}} \ldots I_{j_k} \notin L_{k-1}$ // NOT-IN-SET ($L[k-1], Y$)
                $C := C - \{Y\}$; // REMOVE-ELEMENT ($C, Y$)
                over := true;
            else
                $l := l + 1$;
            endif;
        endwhile;
    endfor;
... continued
```

```
Algorithm:    PASS-DERIVED-ITEMGROUPS ... continued

    for $i$ := 1 to $R$ do // Check counts
      for each $k$-sized itemgroup $X \in r_i$ do
        if $X \in C$ // SET-MEMBER-OF $(X, C)$
          $X$.count := $X$.count + 1;
      endfor;
    endfor;
    $L_k$ := $\{X \in C : X.\text{count} \geq s\}$;
    // Use BUILD-SET and ADD-SET here.
    $k$ := $k + 1$;
  endwhile;
  return $\cup_{k \geq 1} L_k$;
```

NOTE:

- The join computation is not shown.
- Some set operations are shown mathematically, with comments indicating the kinds of set-manipulation functions needed.

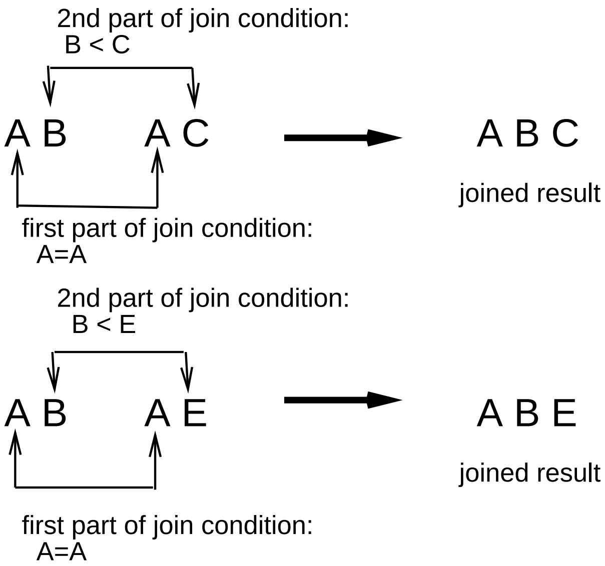
- Example: (same example as before with $s = 0.3$)
  - First pass: $\alpha(A) = 6$, $\alpha(B) = 5$, $\alpha(C) = 6$, $\alpha(D) = 6$, $\alpha(E) = 7$.
    $L_1 := \{A, B, C, D, E\}$.
  - Second pass ($k = 2$):
    - The (L_1, L_1)-join gives $C = \{AB, AC, AD, BC, BD, BE, CD, CE, DE\}$.
    - Since each 1-size subset of each of these is in L_1, the winnowing does not remove anything from C.
    - After a scan, the counts obtained are:

      $$C = \{AB(4), AC(3), AD(2), BC(1), BD(2), BE(3), CD(5), CE(5), DE(5)\}.$$

    - Those with high enough count (3 or more) are retained:

$$L_2 \leftarrow \{AB, AC, AE, BE, CD, CE, DE\}$$

- Third pass ($k = 3$):
    - The (L_2, L_2)-join gives $C = \{ABC, ABE, ACE, CDE\}$, e.g.,


2nd part of join condition:
B < C
A B
A C
A B C
joined result
first part of join condition:
A=A
2nd part of join condition:
B < E
A B
A E
A B E
joined result
first part of join condition:
A=A

    - *Winnowing*:
        - For ABC, we need to check whether $BC \in L_2$
$\Rightarrow$ not in L_2
$\Rightarrow$ discard ABC.
        - For ABE, we need to check whether $BE \in L_2$
$\Rightarrow BE \in L_2$
$\Rightarrow$ retain ABE.
        - Continuing, we find that ABE, ACE, CDE are retained.
    - After a scan, the counts are:

$$ABE(2), ACE(3), CDE(4).$$

    - Those with high enough count (3 or more) are retained:

$$L_3 = \{ACE, CDE\}.$$

- Fourth pass ($k = 4$):
  - The (L_3, L_3)-join is empty.
- Final result:

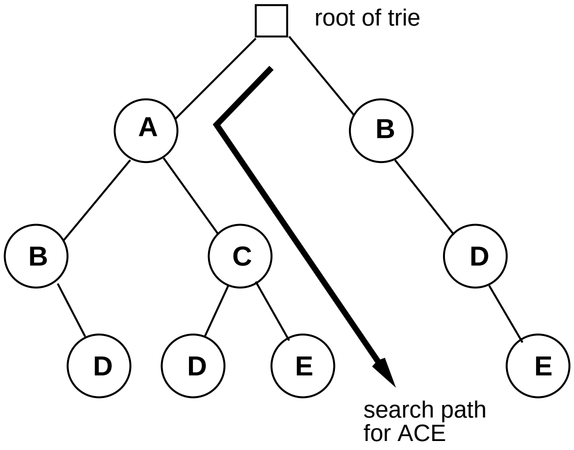
$$A, B, C, D, E, AB, AC, AE, BE, CD, CE, DE, ACE, CDE.$$

12.18 Using Hashtrees: The Apriori Algorithm

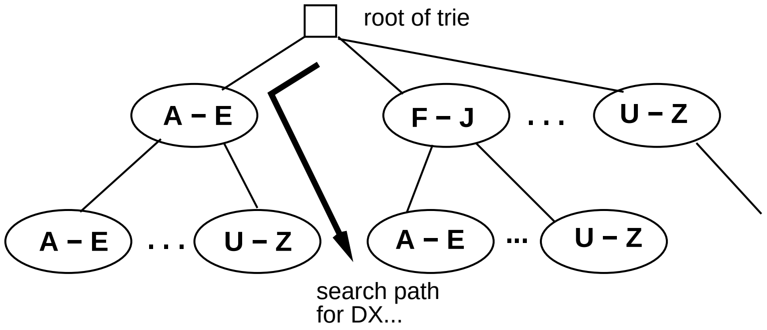
- Recall that in the k-th pass of Pass-Derived-Itemgroups:
  - A set of size-k itemgroups is computed via a join.
  - The set is pruned using the size-$(k-1)$ itemgroups.
  - A scan is made to generate the count for each itemgroup.
- A run-time profile of the previous algorithm shows that a lot of time is spent in generating counts:
  - For each record, we need to figure out which counts should be updated.
  - Example:
    - Suppose $C = \{ABC, ABD, ABE, ACE, BCE, BDE, CDE\}$.
    - Consider a record $< A, B, D, E, F >$.
      Which of the above itemgroups occur in the record?
    - One way of checking: **Record Subset method**
      1. generate all possible size-3 itemgroups in the record:
         $ABD, ABE, ABF, ADE, ADF, AEF, BDE, BDF, BEF, DEF$
         (10 itemgroups).
      2. For each such itemgroup, check whether it is in C.
    - For a record with n items and size-k itemgroups: $\begin{pmatrix} n \\ k \end{pmatrix}$.

      $\Rightarrow$ very large for even moderate sizes (e.g.,$n = 20, k = 10$).
    - Another approach: **Itemgroup Scan method**
      1. Scan each itemgroup in C.
      2. Test whether each itemgroup is in the given record.
         $\Rightarrow$ will be slow if number of itemgroups is large.

- Using a trie in the Record Subset method:
  - To test whether whether ABC is in C, one approach is to test the string ABC against each itemgroup in C.
  - A faster approach is to use a suitable data structure, such as a trie.
  - Example: suppose $C = \{ABD, ACD, ACE, BDE\}$



  - However, if the number of items is large (.e.g, 10^5), the trie could be very wide
    $\Rightarrow$ may not fit in memory.
  - To reduce branching factor, some paths can be coalesced:



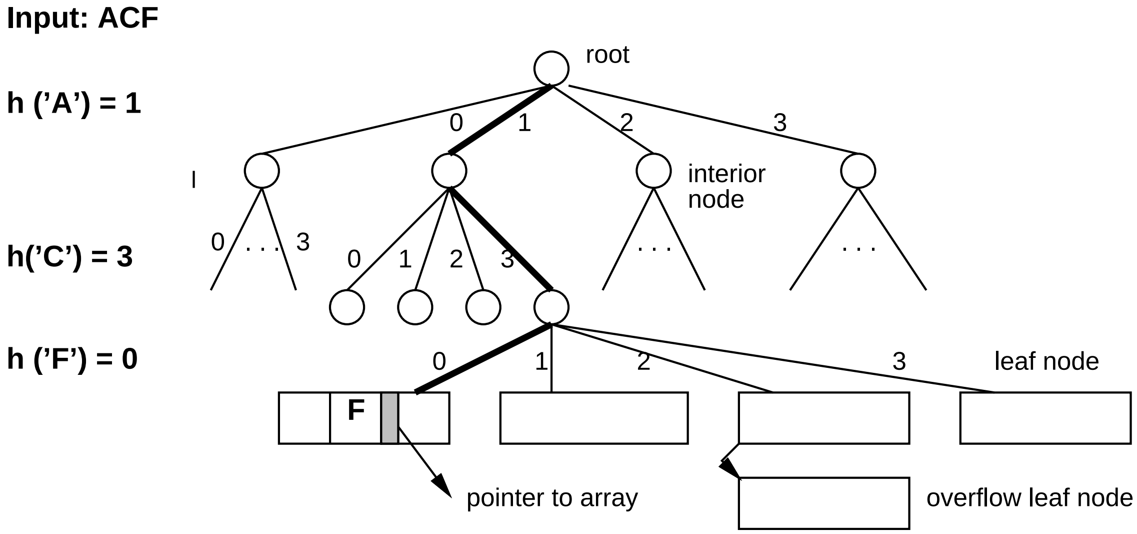
  This is the basic idea used in the hashtree.

- Hashtrees: key ideas
    - Recall: in pass k, C is a list of size-k itemgroups.
    - In pass k, a fresh hashtree is constructed for size-k itemgroups.
    - The list of itemgroups is stored in an array, e.g.

      ```
      typedef struct itemgroup_type {
        char *itemgroup;   // The actual itemgroup
        int count;         // The count, initially set to zero
      } itemgroup_type;
      itemgroup_type *itemgroup_array;
        ...
      // Later
         itemgroup_array[i].count ++; // Incrementing the count
      ```

    - The hashtree is like a B+-tree in some ways:
        - The tree stores pointers to the actual data.
        - In this case, the correct index into the `itemgroup_array` is stored.
        - The hashtree has *internal* and *leaf* nodes.
        - Internal nodes are used for navigation.
        - Leaf nodes contain pointers (offsets) to the itemgroup array.
    - The hashtree is also different in many ways:
        - The leaf nodes are not linked.
        - The search is not in-order: which branch to take depends on a hashing function.

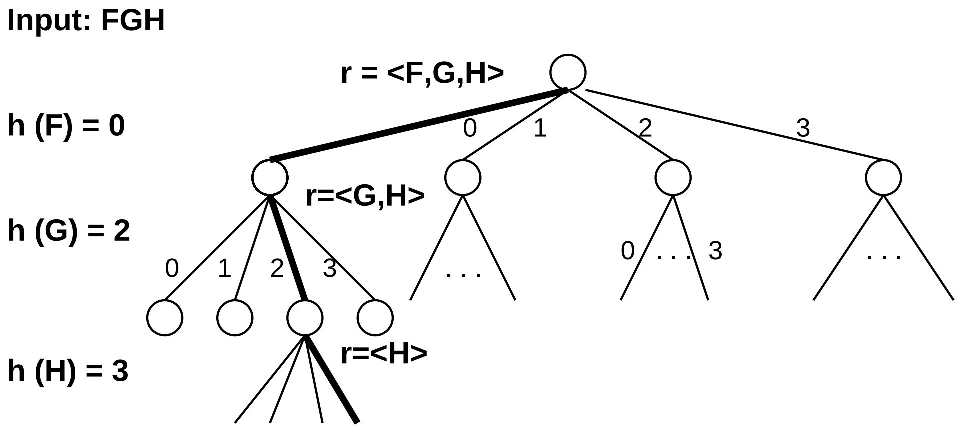
- Example:
  - Consider a hashtree with *branching factor* = 4. (Typically, branching factor is higher).
  - Suppose we want to check whether the itemgroup ACF is in the itemgroup array:



    - $*$ Apply the hashing function to A at the first level, to C at the next level and F at the third level.
    - $*$ Once at the leaf, search for F in the leaf and follow the pointer to the array.
    - $*$ Note: the depth of the hashtree is always the itemgroup size
      $\Rightarrow$ we will always stop at the leaf level with the last item.

- Insertion:
  - First insert the itemgroup in the itemgroup array, and note the array offset (pointer).
  - Then find the appropriate leaf by doing a regular search.
  - Insert in sorted order in the leaf, along with pointer to the itemgroup array.
  - If leaf is full, extend by adding an overflow leaf node.

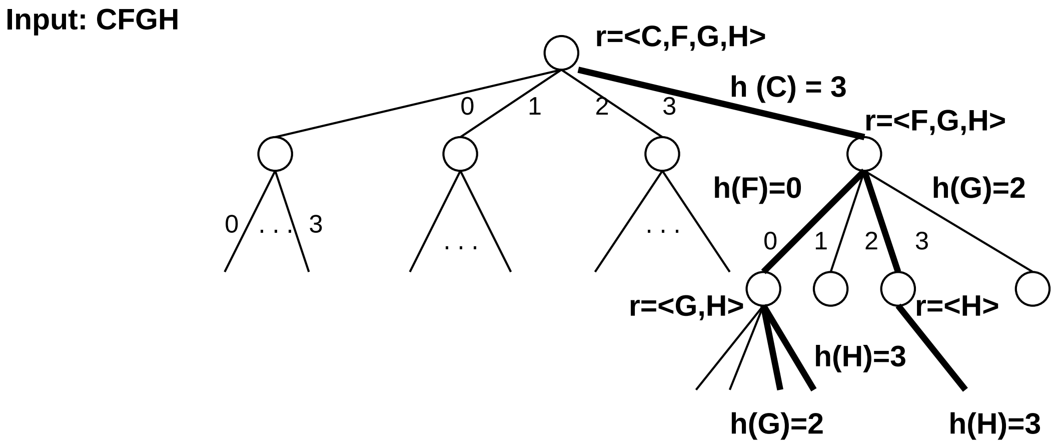
- Checking which subsets are in a record:
  - In a search, we are given a record (e.g., $< A, C, F, G, H >$) and we want to know which itemgroups are in the record
$\Rightarrow$ need to increment their counters.
  - One approach:
    - Generate all possible size-k itemgroups of the record.
    - For each such itemgroup, traverse the hashtree and see if it exists in the itemgroup array.
    - For every itemgroup that is found, increment the counter.
  - Recursive approach:
    - Rather than generate all possible size-k itemgroups, a simple recursive method can be used.
    - Observation: each subtree of the root is a size-$(k-1)$ hashtree.
    - Thus, to check all itemgroups beginning with A:
      - Hash $A \Rightarrow$ say, we get the '1' branch.
      - The remainder of the record is $< C, F, G, H >$.
      - Now apply the function recursively to the '1' subtree with record $< C, F, G, H >$.
    - Now, size-3 itemgroups in the record $< A, C, F, G, H >$ can start with any one of the items A, C or F.
    - Thus, we do hash-search at the root for each of these (and the reminder of the record).
    - In the case of starting with F, the only possibility is to hash G at the next level, then H at the third level.


Input: FGH
r = <F,G,H>
h (F) = 0
0
1
2
3
r=<G,H>
h (G) = 2
0
1
2
3
. . .
0
. . .
3
. . .
r=<H>
h (H) = 3

∗ But if we start with C at the root:

- At the next level we have $< F, G, H >$.
- Both F and G are hashed separately because we have three possible size-2 itemgroups: FG, FH and GH.

  ⇒ the first items are F and G.



- Pseudocode:

  The algorithm has been called the Apriori Algorithm in the literature (because "checking for low-support subgroups" is done *prior* to a scan).

Algorithm: Apriori (R, I, s)

Input: Set of records R, set of items I, support s.
Output: Collection of itemgroups with large enough support.

```
// First pass
$\forall k: \ \alpha[I_k] := 0$ // Initialize counts
**for** $j := 1$ **to** $|R|$ **do**
   **for** $k := 1$ **to** $m$ **do**
      **if** $I_k \in r_j$
         $\alpha[I_k] := \alpha[I_k] + 1$;
$h :=$ Hashtree-Create (1) // Size = 1.
**for** $k := 1$ **to** $m$
   **if** $\alpha[I_k]/|R| \geq s$
      Hashtree-Insert $(I_k)$;
// All other passes.
$k := 2$;
**while** Hashtree-Not-Empty $(h)$
   $C :=$ Compute-Join $(L_{k-1}, L_{k-1})$;
   $h' :=$ Hashtree-Create $(k)$;
   **for each** itemgroup $X = I_{j_1} \ldots I_{j_k} \in C$ // Winnowing
      $l := 1$;      valid := true;
      **while** $l \leq k - 2$ **and** valid
         // $Y = I_{j_1} \ldots I_{j_{j-1}} I_{j_{l+1}} \ldots I_{j_k}$.
         **if not** Hashtree-Recursive-Search $(h, Y)$
            valid := false;
         **else**
            $l := l + 1$;
         **endif**;
      **endwhile**;
      **if** valid // All subgroups checked out
         Hashtree-Insert $(h', X)$;
   **endfor**;
... continued
```

Algorithm: APRIORI ... continued

```
    HASHTREE-DESTROY ($h$);
    $h$ := $h'$;
    for $i$ := 1 to $|R|$ do
        HASHTREE-UPDATE-COUNTS ($h, r_i, k$);
    // Remove all itemgroups with low counts.
    for each itemgroup $X \in h$ do // $X$ is in array
        if $X$.count $< s$
            HASHTREE-REMOVE-ITEMGROUP ($h, X$);
    $k$ := $k + 1$;
endwhile;
return $\cup_{k \geq 1} L_k$;
```

Algorithm: HASHTREE-UPDATE-COUNTS (h, r, k)

Input: hashtree id h, record r, size k.
Output: Counts are updated.

```
    // Suppose $r =< I_{j_1}, I_{j_2}, \ldots, I_{j_l} >$.
for $p$ := 1 to $l - k$
    HASHTREE-RECURSIVE-UPDATE ($r, p, k, h$.root);
return;
```

```
Algorithm:  HASHTREE-RECURSIVE-UPDATE ($r, p, k$, node)

Input: record $r$, offset $p$, size $k$, hashtree node.
Output: Counts are updated.
      // Suppose $r =< I_{j_1}, I_{j_2}, \ldots, I_{j_l} >$.
      if node.leaf = true // Bottom out of recursion.
         if $I_{j_p} \in$ node
            follow pointer to itemgroup array and increment count;
            // Note: count should be incremented only once
            // for each record.
            return;
         endif;
      endif;
      $c$ := hashfunction ($I_{j_p}$);
      node2 := node.child[$c$];
      for $q$ := $p + 1$ to $l - k$
         HASHTREE-RECURSIVE-UPDATE ($r, q, k - 1$, node2);
      return;
```

Algorithm: HASHTREE-RECURSIVE-SEARCH (node, Y, i)

Input: hashtree node, itemgroup Y, offset i.
Output: true if itemgroup is in tree, false otherwise.

```
// Assume $Y = I_{j_1} \ldots I_{j_k}$.
if node.leaf = true
   if $I_{j_k} \in$ node
      return pointer to location in itemgroup array;
   else
      return NULL; // false
else
   $c$ := hashfunction $(I_{j_1})$;
   node2 := node.child[$c$];
   return HASHTREE-RECURSIVE-SEARCH $(\text{node2}, Y, i + 1)$;
endif;
```

12.19 On-Line Analytical Processing (OLAP): Introduction

- On-Line Analytical Processing (OLAP) is the term used for a class of *aggregate* queries.
- Consider an airline (McValue Airlines) with the following data

  SALES (YEAR, CONTINENT, FLT_TYPE, REVENUE).

  where the FLT_TYPE is given by

FLT_TYPE	DESCRIPTION
1	Short-domestic
2	Long-domestic
3	International

and where the data in SALES is: (revenue in millions)

SALES	YEAR	CONTINENT	FLT_TYPE	REVENUE
	1997	Europe	1	125
	1997	Europe	2	50
	1997	Europe	3	225
	1997	Asia	1	25
	1997	Asia	2	75
	1997	Asia	3	100
	1997	N.America	1	325
	1997	N.America	2	450
	1997	N.America	3	75
	1998	Europe	1	110
	1998	Europe	2	40
	1998	Europe	3	200
	1998	Asia	1	20
	1998	Asia	2	130
	1998	Asia	3	50
	1998	N.America	1	460
	1998	N.America	2	170
	1998	N.America	3	30

Note: number of tuples = 2 YEARs × 3 CONTINENTS × 3 FLT_TYPEs = 18.

- Typical queries:
  - What is the total 1997 revenue?
  - Output the total revenue in each continent year-by-year.
  - What is the total 1997 revenue for International flights?
  - What is the total revenue on European domestic (long and short) flights across all years?
  - What is the maximum revenue in any European flight category in any year?

  NOTE:

  - All the queries involve aggregate functions (**sum** and **max** above).
  - The queries involve aggregates across various subsets of the attributes.

  The answers:

  - What is the total 1997 revenue?
    ⇒ $1,450 million.
  - Output the total revenue in each continent year-by-year.

1997	Europe	400
1998	Europe	350
1997	Asia	200
1998	Asia	200
1997	N.America	850
1998	N.America	660

  - What is the total 1997 revenue for International flights?
    ⇒ $400 million.
  - What is the total revenue on European domestic (long and short) flights across all years?
    ⇒ $325 million.

  - What is the maximum revenue in any European flight category in any year?
    ⇒ $225 million (International).

- Computing the queries in SQL:
  - What is the total 1997 revenue?

```
select      S.YEAR, sum(S.REVENUE)
from        SALES S
where       S.YEAR = 1997
group by    S.YEAR
```

  - Output the total revenue in each continent year-by-year.

```
select      S.YEAR, S.CONTINENT, sum(S.REVENUE)
from        SALES S
group by    S.YEAR, S.CONTINENT
order by    S.CONTINENT
```

  - What is the total 1997 revenue for International flights?

```
select      S.FLT_TYPE, sum(S.REVENUE)
from        SALES S
where       S.YEAR=1997 and S.FLT_TYPE=3
group by    S.FLT_TYPE
```

  What is the cost of computation?

  - Consider the query "Output the total revenue in each continent year-by-year."
    * Need to sort data by continent and year.
    * After sort, aggregates can be computed in a single scan.
  - If data was sorted by (CONTINENT, YEAR), then it must be resorted for aggregates on (FLT_TYPE).
  - Generally, if the data is already sorted according to the desired output, one scan is required.
  - Otherwise, a sort is also needed.

12.20 OLAP: The CUBE View

- Most OLAP applications consider data with $m + 1$ attributes in which m attributes are "parameter" attributes and the $(m + 1)$-st attribute is the "aggregate" attribute.

  E.g., in

  SALES (YEAR, CONTINENT, FLT_TYPE, REVENUE)

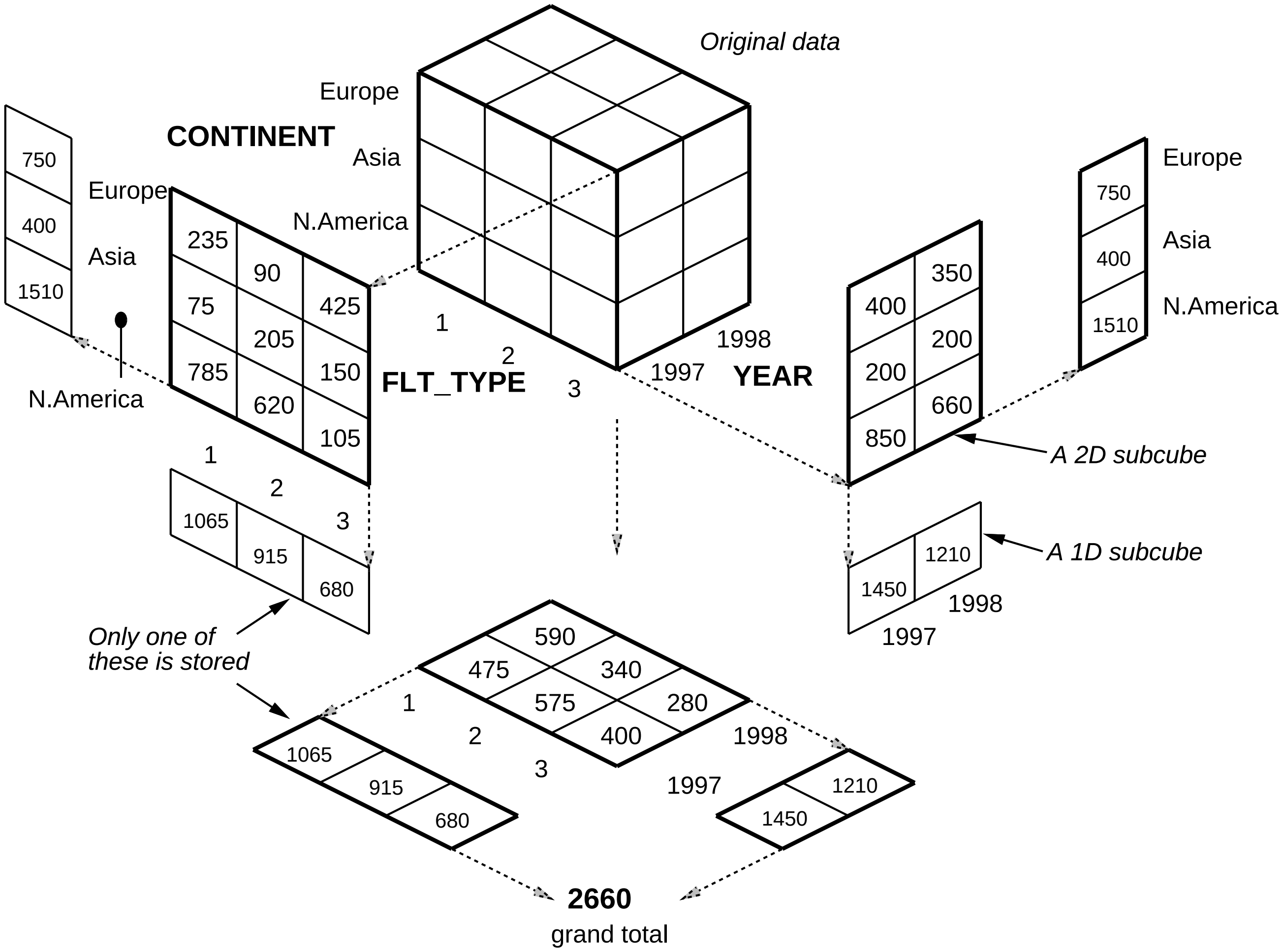
  - REVENUE is the aggregate attribute.
    (Sums are computed over REVENUE values.)
  - YEAR, CONTINENT and FLT_TYPE are parameter attributes.
  - Thus, there are 3 parameter attributes
    ⇒ we call this a 3D aggregate problem.
  - For m parameter attributes, it's an m-dimensional aggregate problem.

  In general, the data will be a relation $R(A_1, \ldots, A_m, F)$ where

  - $A_1, \ldots, A_m$ are the parameter attributes.
  - F is the aggregate attribute.

  The subcube with attributes $A_{i_1} \ldots A_{i_k}$ will be denoted by $S(A_{i_1} \ldots A_{i_k})$.

- It is often convenient to view a 3D problem using a cube: (Although strictly a cuboid, the term *cube* is used).



For m-dimensional data, there are several subcubes for each dimension $k < m$.

12.21 OLAP: Repeated Queries

- Typically, a manager or accountant sits at a terminal and queries on several attribute subsets repeatedly
  ⇒ multiple views required quickly.

  If queries are generated via SQL statements
  ⇒ could take a long time.

- Prior computation and storage of subcubes:
  - It is better to materialize each subcube and store it.
  - For example, the (YEAR,CONTINENT) subcube is computed as:

1997	Europe	400
1998	Europe	350
1997	Asia	200
1998	Asia	200
1997	N.America	850
1998	N.America	660

  - Storage options:
    1. Store each possible subcube separately:
       * Store the subcube S(YEAR, CONTINENT) in a relation S1 (YEAR, CONTINENT).
       * Store the subcube S(YEAR, FLT_TYPE) in relation S2 (YEAR, FLT_TYPE).
       * Store the subcube S(CONTINENT, FLT_TYPE) in relation S3 (CONTINENT, FLT_TYPE).
       * Store the subcude S(YEAR) in relation S4 (YEAR).
       * ...etc.
    2. Store each subcube within the original relation using **null** values. For example, the subcube S(YEAR, CONTINENT) is

1997	Europe	400
1998	Europe	350
1997	Asia	200
1998	Asia	200
1997	N.America	850
1998	N.America	660

These tuples would be extended with **null**'s and added to the original data:

SALES	YEAR	CONTINENT	FLT_TYPE	REVENUE
	1997	Europe	1	125
	1997	Europe	2	50
	1997	Europe	3	225
	1997	Asia	1	25
	1997	Asia	2	75
	1997	Asia	3	100
	1997	N.America	1	325
	1997	N.America	2	450
	1997	N.America	3	75
	1998	Europe	1	110
	1998	Europe	2	40
	1998	Europe	3	200
	1998	Asia	1	20
	1998	Asia	2	130
	1998	Asia	3	50
	1998	N.America	1	460
	1998	N.America	2	170
	1998	N.America	3	30
	1997	Europe	**null**	400
	1998	Europe	**null**	350
	1997	Asia	**null**	200
	1998	Asia	**null**	200
	1997	N.America	**null**	850
	1998	N.America	**null**	660

NOTE: since **null** already has a use, the use of the keyword **all** has been proposed.

- An observation:
  - The tuples for the subcube S(YEAR, CONTINENT) repeat the strings "1997" and "1998"

    ⇒ a waste of space.

  - The only real information is the collection of aggregates:

1997	Europe	**400**
1998	Europe	**350**
1997	Asia	**200**
1998	Asia	**200**
1997	N.America	**850**
1998	N.America	**660**

    Therefore, it is more efficient to directly store the subcubes using an internal representation of a matrix:
    ⇒ called the *multidimensional approach*.

- Problems with large dimensions:
  - A 20-dimensional data set has 2^{20} subsets of attributes
    ⇒ 2^{20} possible subcubes.
  - Many subcubes are of high dimension
    ⇒ need to store high-dimensional matrices.

12.22 The Multidimensional Approach: Hash-Based Computation

- Some useful observations:

  Computing sizes of subcubes:

  - Consider the subcube S(YEAR, CONTINENT).

    How many entries in the subcube?
    2 YEAR's, 3 CONTINENT's
    $\Rightarrow$ 6 entries

  - In general, for relation $R(A_1, \ldots, A_m)$ let

    $$n(A_i) = \# \text{ values of attribute } A_i \text{ present.}$$

    Then, the size of subcube $A_{i_1} A_{i_2} \ldots A_{i_k}$ is

    $$\text{size}(A_{i_1} \ldots A_{i_k}) = n(A_{i_1}) n(A_{i_2}) \ldots n(A_{i_k}) = \prod_{j=1}^{k} n(A_{i_j}).$$

  - We have made an implicit assumption: all possible combinations of values exist as tuples in the data
    $\Rightarrow$ the *full-cube* assumption. E.g., if 1997 exists as a YEAR and Europe exists as a CONTINENT then $< 1997, \text{Europe} >$ will exist in some tuple.

  Example:

  - Consider the relation $R(A_1, A_2, A_3, F)$ with

    $$\begin{aligned} n(A_1) &= 4 \\ n(A_2) &= 50 \\ n(A_3) &= 1000 \end{aligned}$$

Thus, there are $4 \times 50 \times 1000 = 200,000$ tuples.

- Which subcubes need to be computed?
  - ∗ The subcube $S(A_1A_2A_3)$ is the given data.
  - ∗ The three 2-dimensional subcubes $S(A_1A_2)$, $S(A_1A_3)$ and $S(A_2A_3)$.
  - ∗ The four 1-dimensional subcubes $S(A_1)$, $S(A_2)$, $S(A_3)$ and $S(A_4)$.
- Suppose 1000 integers fit into a block.
- Sizes:

$$\begin{aligned}
\text{size}(A_1A_2A_3) &= 200,000 \text{ values} = 200 \text{ blocks}\\
\text{size}(A_1A_2) &= 200 \text{ values} = 1 \text{ block}\\
\text{size}(A_1A_3) &= 4000 \text{ values} = 4 \text{ blocks}\\
\text{size}(A_2A_3) &= 50,000 \text{ values} = 50 \text{ blocks}\\
\text{size}(A_1) &= 4 \text{ values} = 1 \text{ block}\\
\text{size}(A_2) &= 500 \text{ values} = 1 \text{ block}\\
\text{size}(A_3) &= 1000 \text{ values} = 1 \text{ block}
\end{aligned}$$

- Computing each subcube via hashing:
  - Scan original file.
  - Hash tuples into hash table containing sums;
  - Update appropriate sum for each tuple scanned.

  For the above data:
  ⇒ 6 scans of data
  ⇒ 6 scans of 200,000 tuples.

  Observe:

  - Once the subcube $S(A_1A_2)$ is computed, $S(A_1)$ can be computed from a scan of $S(A_1A_2)$
    ⇒ only one block needs to be scanned.

- The hash table:
  - Consider computing the subcube A_1A_3.
  - There are 4000 values of A_1A_3
    ⇒ need a sum for each of these 4000 values.
  - Create a hashtable with 4000 such sums.
  - Scan data and hash each tuple to discover which sum to update.

  Example:

  - Suppose we are computing the subcube S(YEAR, CONTINENT).
  - We will need a sum for each possible combination of YEAR and CONTINENT:

1997	Europe
1998	Europe
1997	Asia
1998	Asia
1997	N.America
1998	N.America

    In this case, we need 6 sums (the size of the subcube).
  - In practice, the size of the subcube can be large
    ⇒ many counters will be needed.
  - As we scan the actual data, we need to add the revenue in a tuple to the appropriate counter.
  - A simple scan or binary search can be used, but hashing is very efficient.
  - If each sum is in a different bucket, a single access is needed for an update.

- Example:

  Consider the relation $R(A_1, A_2, A_3, A_4, F)$ with

$$\begin{aligned} n(A_1) &= 4 \\ n(A_2) &= 50 \end{aligned}$$

$$\begin{aligned} n(A_3) &= 1000 \\ n(A_4) &= 200 \end{aligned}$$

The subcubes are:

- 4-dim: $A_1A_2A_3A_4$.
- 3-dim: $A_1A_2A_3$, $A_1A_2A_4$, $A_1A_3A_4$, $A_2A_3A_4$.
- 2-dim: A_1A_2, A_1A_3, A_1A_4, A_2A_3, A_2A_4, A_3A_4.
- 1-dim: A_1, A_2, A_3, A_4.

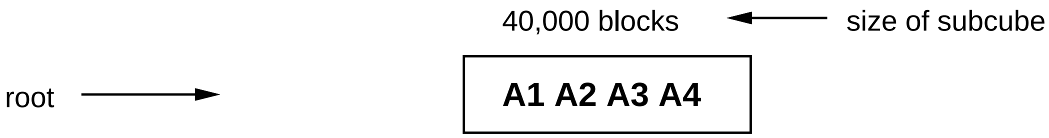
Sizes:

$$\begin{aligned} \text{size}(A_1A_2A_3A_4) &= 4 \times 10^7 \text{ values} = 40,000 \text{ blocks} \\ \text{size}(A_1A_2A_3) &= 200,000 \text{ values} = 200 \text{ blocks} \\ \text{size}(A_1A_2A_4) &= 40,000 \text{ values} = 40 \text{ blocks} \\ \text{size}(A_1A_3A_4) &= 800,000 \text{ values} = 800 \text{ blocks} \\ \text{size}(A_2A_3A_4) &= 10^7 \text{ values} = 10,000 \text{ blocks} \\ \text{size}(A_1A_2) &= 200 \text{ values} = 1 \text{ block} \\ \text{size}(A_1A_3) &= 4000 \text{ values} = 4 \text{ blocks} \\ \text{size}(A_1A_4) &= 800 \text{ values} = 1 \text{ block} \\ \text{size}(A_2A_3) &= 50,000 \text{ values} = 50 \text{ blocks} \\ \text{size}(A_2A_4) &= 10,000 \text{ values} = 10 \text{ blocks} \\ \text{size}(A_3A_4) &= 200,000 \text{ values} = 200 \text{ blocks} \end{aligned}$$

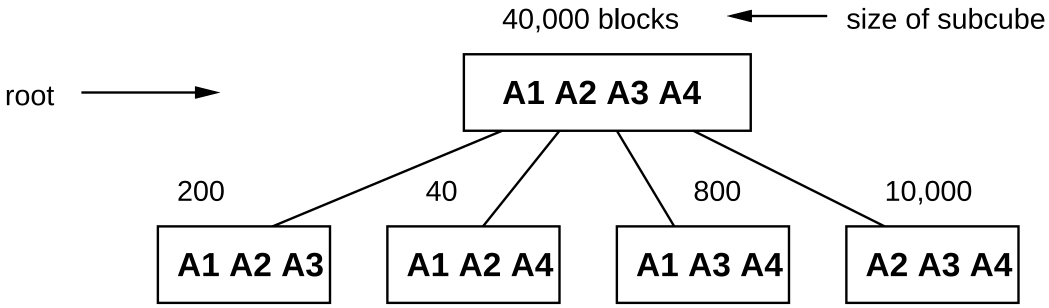
(Sizes of the 1-dim cubes not shown).

Construct a tree:

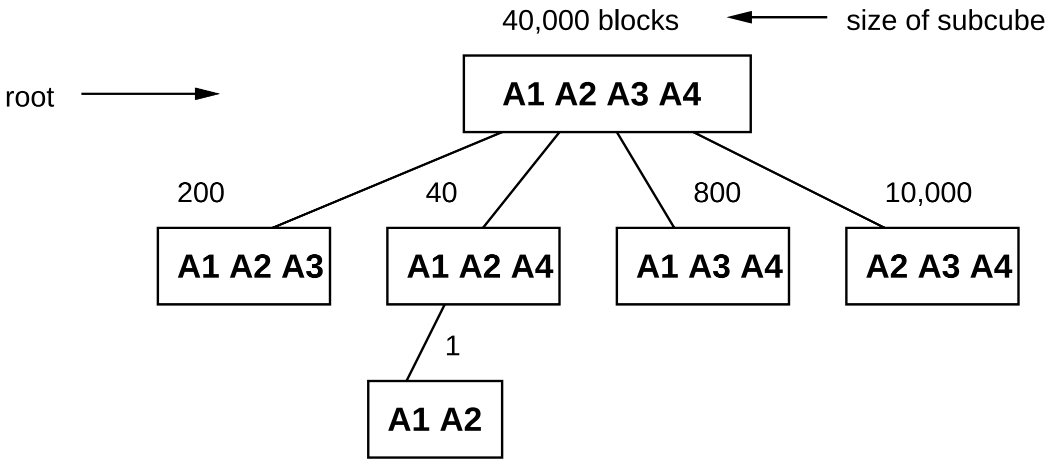
- Step 1: Place the subcube $A_1A_2A_3A_4$ at the root:



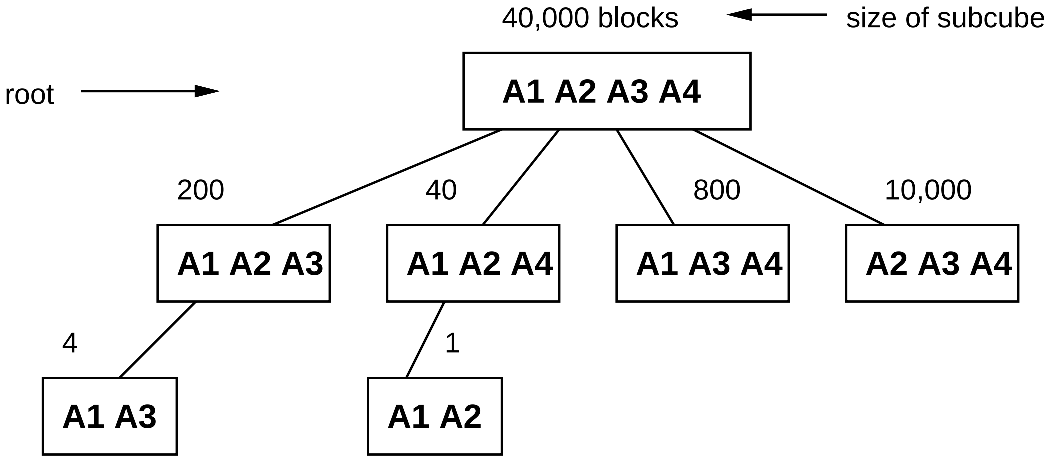
– Step 2: No choice of parent for 3-dim subcubes $A_1A_2A_3$, $A_1A_2A_4$, $A_1A_3A_4$, $A_2A_3A_4$.



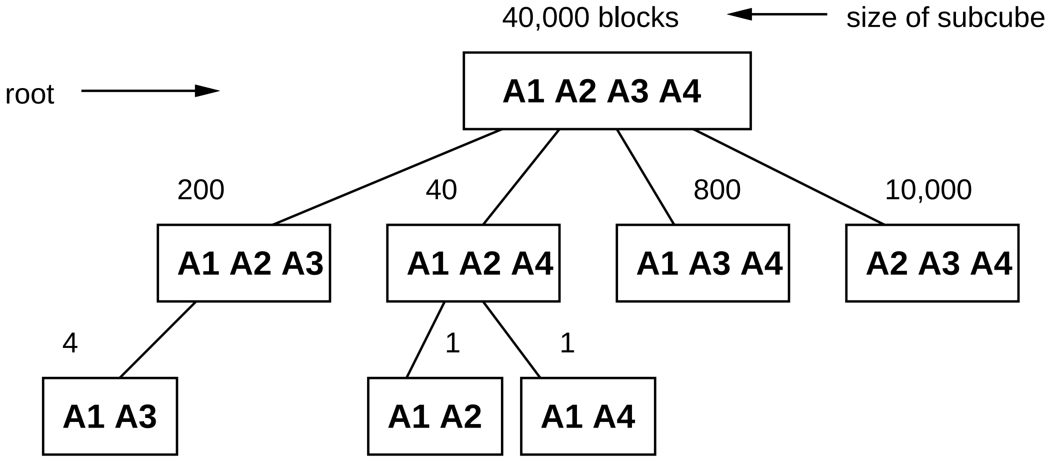
– Step 3: For subcube A_1A_2 pick smallest parent
$\Rightarrow$ 40 blocks of $A_1A_2A_4$.



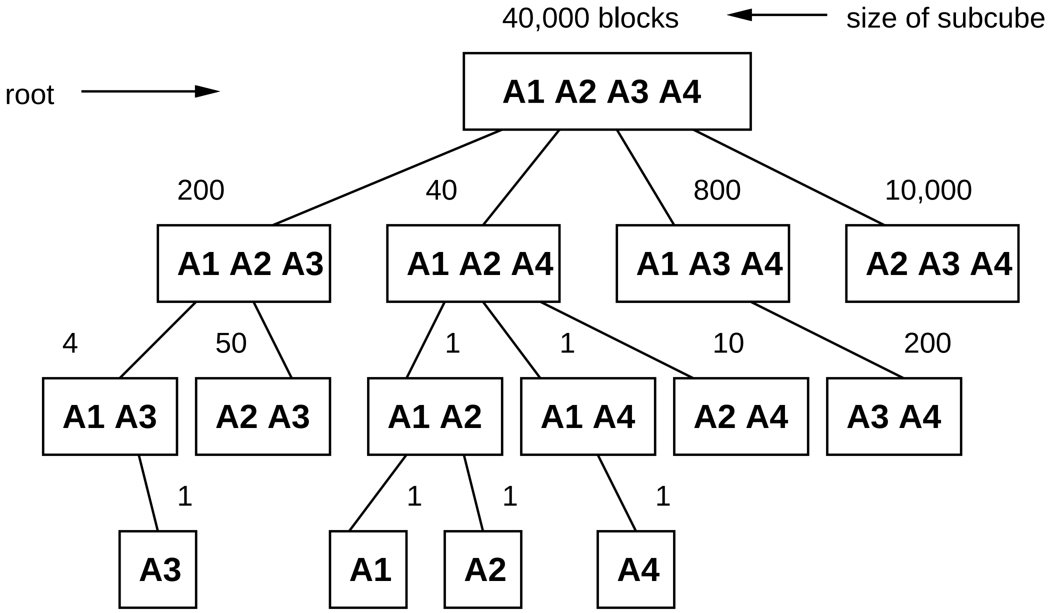
– Step 4: For subcube A_1A_3 pick smallest parent
$\Rightarrow$ 200 blocks of $A_1A_2A_3$.



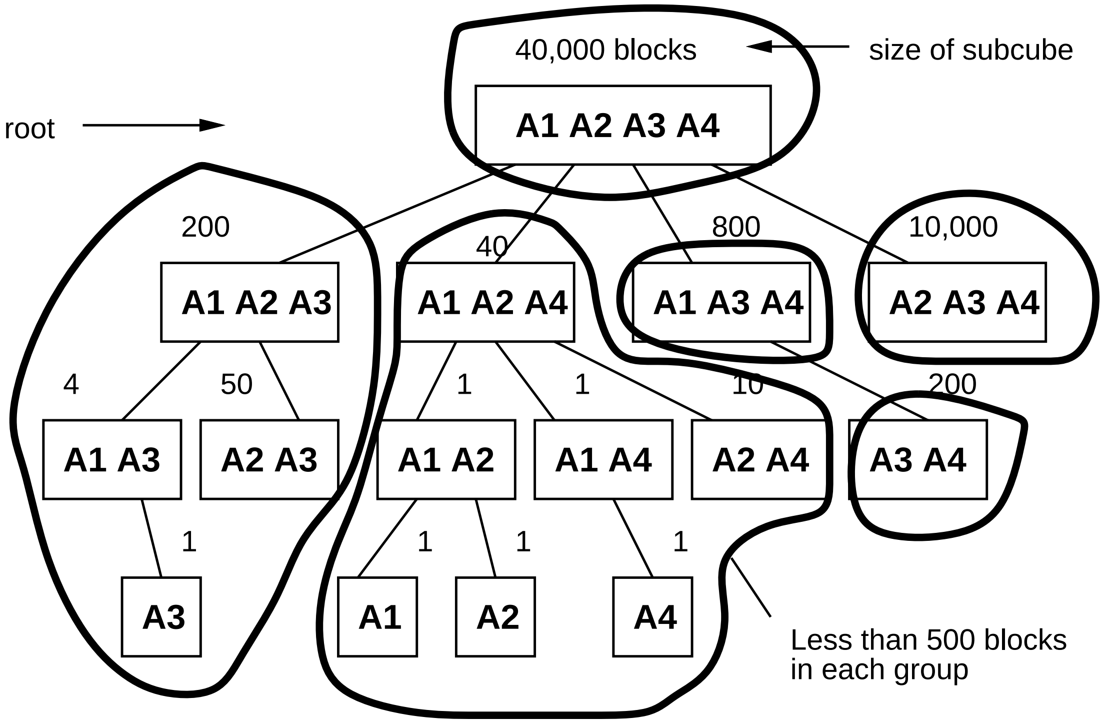
– Step 5: For subcube A_1A_4 pick smallest parent $\Rightarrow$ 40 blocks of $A_1A_2A_4$.



– ... continuing, we get the final tree:

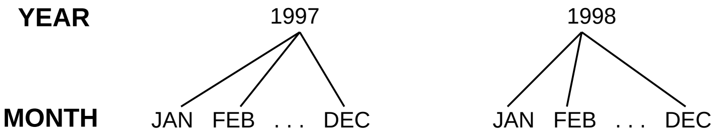


Suppose memory size is 500 blocks. Several subtrees can be computed in parallel:



12.23 Hierarchies on Attributes: Roll-up and Drill-down

- What is a hierarchy on an attribute?
  - Consider the attribute DATE in
    SALES (DATE, CONTINENT, FLT_TYPE, REVENUE).
  - There is a natural division of dates by YEAR and MONTH.
  - YEAR and MONTH form a hierarchical division:



- Why is this important?
  - Queries often use hierarchies.
  - Example:
    - ∗ A user requests aggregate revenue by the subcube (YEAR, CONTINENT).
    - ∗ Then, if that's interesting, the user wants to look at a breakdown month-by-month
      ⇒ a subcube addressed by (MONTH, CONTINENT).
    - ∗ This is an example of *drilling down* a hierarchy.
  - Example:
    - ∗ A user requests aggregate revenue by (DATE, FLT_TYPE).
    - ∗ Then, a broader picture can be obtained by requesting the summary (MONTH, FLT_TYPE)
      ⇒ a subcube addressed by (MONTH, CONTINENT).

  - ∗ This is an example of *rolling up* a hierarchy.

- Both *drill-down* and *roll-up* are useful OLAP operations.

Chapter 13

Summary

Course Notes on Database Systems

13.1 CS 780: A Summary

- What did we learn in this course?

  Concepts:

  - What is a database? How is it different from a simple file system?
    ⇒ File system does not provide recovery, concurrency, fast querying etc.
  - Data Models.
    ⇒ The Relational Model (tables), constraints.
  - Relational databases: relational algebra.
    ⇒ σ, Π, $\bowtie$ etc.
  - Relational databases: SQL.
    ⇒ **select** statement, **create table**, **update** etc.
  - Example: Oracle.
    ⇒ Structure of Oracle, using SQLPlus for SQL queries.
  - Database programming in Oracle.
    ⇒ Using the C-library (OCI) and embedded SQL.
  - Physical implementation: file structures.
    ⇒ Heapfiles, hashfiles, sorted files, disk I/O.
  - B-trees and B+-trees
    ⇒ Insertion, search and deletion.
  - Hashing
    ⇒ Extendible hashing, linear hashing.
  - Sorting
    ⇒ Binary merge-sort, polyphase merge-sort, snowplow method.
  - Query processing
    ⇒ Developing a query plan, evaluating different plans.

  - Database design: normalization
    ⇒ Theory of Functional Dependencies (FD's), normal forms: 2NF, 3NF, BCNF
  - Recovery
    ⇒ Shadow paging, Log-based recovery, **redo**'s and **undo**'s.
  - Concurrency
    ⇒ Problems with concurrent execution, serializability, locking, deadlock.
  - Spatial databases
    ⇒ Introduction, issues, types of queries.
  - Spatial indices
    ⇒ Rtrees, Peano curves, grid files.
  - Spatial query processing
    ⇒ point location, convex hulls, intersections.

- Advanced topics we have not covered:
  - Database systems (of interest to dbase system developers):
    * Details of implementing a transaction manager
      ⇒ Including code for locking, recovery, buffer management.
    * Advanced query processing
      ⇒ Query plan enumeration, transformation heuristics.
    * Disk subsystem implementation
      ⇒ Disk I/O, partitions and extents, directory management.
    * Parallel databases
      ⇒ parallel algorithms for implementing relational operators.
    * Distributed databases
      ⇒ Concurrency control and recovery in a distributed system.
  - Database theory:
    * Proofs for results on functional dependencies and normal forms.
    * Higher normal forms.
    * Detailed algorithms for decomposition and minimal covers.

  - Logic databases, deductive databases.
- Applications development:
  - Generating reports.
  - Oracle Report, combining forms and report.
  - Client-server application development, web-based interfaces.
  - ODBC and standard interfaces to other database systems.
  - JDBC and Java-related interfaces.
- Text processing and document retrieval:
  - String and pattern search algorithms
    ⇒ string search, approximate search, regular expression search.
  - Indices for text data, keyword-searching.
  - Organizing document databases
    ⇒ indices, clustering, thesaurus construction.
  - Data compression
    ⇒ Huffman coding, Liv-Zempel algorithm.
- Other data models:
  - Network and hierarchical models.
  - Object-oriented databases, persistent objects.
  - Object-relational databases.
- Non-traditional databases:
  - Image and multimedia databases.
    ⇒ Storage and retrieval of image, audio and video data.
  - Scientific databases
    ⇒ Databases for CAD, astronomy, DNA and medical applications.
  - Temporal databases
    ⇒ history-related queries.
- New business applications:
  - Data mining.
  - Online Analytical Processing (OLAP) and multidimensional databases.

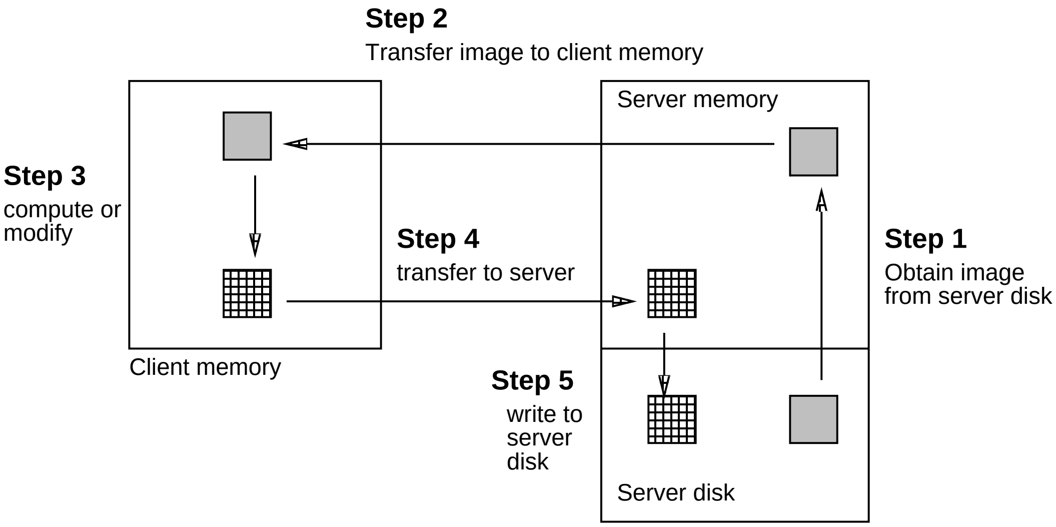
13.2 Object-Oriented and Object-Relational Databases

- The development of OO (Object-Oriented) and OR (Object-Relational) databases is motivated by:
  - Supporting new data types: image, spatial and multimedia data.
  - Providing applications developers with mechanisms to define their own data types, their own operators.
  - Allowing for efficient implementation of the above.
  - Integrating a dbase-like file system (with recovery and concurrency) into a high-level programming language (like C++).
  - Providing applications programmers with powerful OO features (inheritance, encapsulation).
- Consider supporting image data:
  - We want to allow users to store and retrieve images in a variety of formats (GIF, Postscript, Xbitmap, JPEG, etc).
  - We want to allow some kinds of querying, e.g., to determine if two images are of approximately equal intensity.
  - Currently, most commercial systems have a data type for binary (raw bits) data.
  - Thus, an image can be stored as follows:

```
create table IMAGE_TABLE ( bit      IMAGE,
                           varchar  NAME,
                           char(3)  TYPE,
                           number   CODE);
```

  - Now, a C program can be written (as a client) to retrieve images and work with them (e.g., to compute mean intensity etc).

- Unfortunately, this “compute-at-the-client” design causes lots of data to move back and forth between client and server memory:



- Thus, a query that searches through the whole dbase of images will cause a lot of data transfer
  ⇒ better to have all the work done at the server.
- Some database systems provide SQL-like functions to support non-traditional data.
  - These are usually very limited (e.g., only very few data types).
  - Also, there is no programmer flexibility in defining new types and functions.

- Object-Oriented Databases.

  OODB vendors take the view that:

  - Object-oriented programming is here to stay (e.g., C++, Smalltalk, Java).
  - Objects are the natural way to treat all types of data.
  - If database models can't handle objects, then it's time to change database software.

- There are three schools of thought on this matter:

- *Pure OO*:
    - ∗ Forget the concerns of dbase folks and instead simply design persistence into a programming language, e.g., persistent C++.
    - ∗ What does *persistence* mean? All objects that are written into are stored in an object repository and can be retrieved later.
    - ∗ Dbase vendors then can use a persistent language to develop dbase-specific libraries for applications programmers.
    - ∗ Do not provide a database server.
- *OODB*:
    - ∗ Since pure-OO folks don't really care about dbase concerns (such as recovery and concurrency), design a dbase-specific object repository.
    - ∗ Provide an interface via an object-oriented language but provide a full-fledged database server.
    - ∗ Allow applications programmers to program assuming persistent objects.
    - ∗ Handle recovery and concurrency (and efficient memory management).
    - ∗ Provide a library for SQL programmers (parser, interpreter, query optimizer etc).
- *Object-Relational*:
    - ∗ Relational systems aren't broken – let's instead amend them to handle newer data types.
    - ∗ Allow programmers to define new data types and functions that manipulate the data types.
    - ∗ The system then loads programmer-defined functions *into* the server (incorporating it as server code) to handle the new types.

- Object-Oriented Databases (OODB):
    - In a typical implemention, an application programmer will define an object using an Object Definition Language, e.g.,

```
object definition my_object {
    private data:
        int size; // For a size×size image
        real image[100][100];
    methods:
        int get_size ();
        set_size (int val);
        real** get_image ();
        set_image (real** I);
}
```

- The programmer then compiles this using a special compiler for the language.
- The compiler produces C++ output (in a `.h` file) that looks like the above:

```
class my_object {
  private:
    int size;
    float image[100][100]
  public:
    int get_size ();
    void set_size (int);
    float ** get_image ();
    void set_image (float** I);
};
```

- Then, the programmer is expected to write code for the functions in the class.
- Finally, the system compiles the code for use when the object is manipulated.
- An applications programmer then defines the object using an encapsulation (usually via templates in C++):

```
encap<my_object> A;
A.set_size (5);  // Not allowed (won't work)
A.update->set_size(5) // Works. Goes via the encapsulation.
```

The trick is to define the encapsulation so that it returns a pointer (usually a `const` pointer in C++). Then, an applications programmer cannot modify the data
⇒ must go via a member function of the encapsulation instead
⇒ server has control over modification.

- Since updates go via the OO-system, the data remains at the server, allowing for recovery and concurrency control.
- Example: The SHORE (Scalable Heterogeneous Object REpository) System, developed at the University of Wisconsin, Madison.

- The *Object-Relational* approach:
  - The programmer is allowed to create types and new functions.
  - Creating a type:
    - Suppose we want to create an image type called `image`.
    - First, create the image type in C:

```
typedef struct image {
  int size;
  float image[100][100];
} image;
```

    - Then, create two C functions that handle I/O for this new data type:

```
image* image_in (char *data)
{
  // Code to take in raw data and put it in the struct
}

char* image_out (image* I)
```

```
{
  // Code to take in an image and put out char data, e.g
}
```

and compile them.

∗ Now, tell the database what you've done:

```
CREATE FUNCTION image_in (char*)
  RETURNS image
  AS '/usr/mom3/joe/780/image.o'
  LANGUAGE 'C';

CREATE FUNCTION image_out (image)
  RETURNS char*
  AS '/usr/mom3/joe/780/image.o'
  LANGUAGE 'C';
```

∗ Next, create the type in the database:

```
 CREATE TYPE image ( size = 10002,
                     input = image_in,
                     output = image_out);
```

∗ Finally, create manipulation functions and declare them.

∗ After all of this is done, an applications programmer can use the new data type in a query:

```
SELECT name
FROM image_file
WHERE approx_equals (image_file.image, :test_image);
```

– Example: Postgres95 (University of California, Berkeley), Illustra (part of Informix), Oracle 8.