B trees, Sorting, indices

CS 2541: Database Systems
DBMS File Organization

Algorithms & ‘Data’ Structures for DBMS file organization

- B-trees: multi-level index
  - Most commonly used database index structure today
- Hash index
  - ‘standard’ hash table concept
- External sorting algorithms
  - Sorting data residing on disk
  - Time complexity measured in terms of disk read/write

Multilevel Index

Multilevel Indices and B-trees
### B-Trees and Indexing

- Multi-level indexes can improve search performance.
- One of the challenges in creating multi-level indexes is maintaining the index in the presence of inserts and deletes.
- We look at B-trees which are the most common form of index used in database systems today.

### Review of B-tree concepts...

- Search in trees = $O(h)$, $h$ is height of tree
- Binary tree worst case height = $O(n)$
- Tree may get unbalanced
- B-trees – are a class of balanced trees
- Forces all leaves to have same height
- Original motivation was search trees (not databases)

### Balancing the tree...

- Need to place some constraint that will force the tree to be balanced
  - This is accomplished by specifying the minimum and maximum number of entries at each node – the order $d$ of tree
  - Alternately, can specify minimum and maximum number of children at each node – called the degree $m$ of tree

### B-trees

A **B-tree** is a search tree where each node has $\geq n$ data values and $\leq 2n$, where we chose $n$ for our particular tree.

- Each key in a node is stored in a sorted array.
  - key[0] is the first key, key[1] is second key,...,key[2n-1] is the 2n$^th$ key
  - key[0] < key[1] < key[2] < ... < key[2n-1]
- There is also an array of pointers to children nodes:
  - child[0], child[1], child[2], ..., child[2n]
  - Recursive definition: Each subtree pointed to by child[i] is also a B-tree.
- For any key[i]:
  1) key[i] > all entries in subtree pointed to by child[i]
  2) key[i] <= all entries in subtree pointed to by child[i+1]
- A node may not contain all key values.
  - # of children = # of keys + 1
- A B-tree is balanced as every leaf has the same depth.
B-tree definition

- Every tree node of B-tree of order \(d\) (\(d \geq 1\)) must have:
  - (except the root) must have at least \(d\) key values (entries) sorted inside the node
  - Cannot have more than \(2d\) key values
  - Has one more tree pointer than the number of key values; i.e., between \(d+1\) and \(2d+1\)
  - Is either a leaf or an internal node

B-tree Alternate definition

- Every tree node of B-tree of degree \(m\) (\(m > 1\)) must have:
  - (except the root) must have at least \(m-1\) key values (entries) sorted inside the node
  - Cannot have more than \(2m-1\) key values
  - Has one more tree pointer than the number of key values; i.e., between \(m\) and \(2m\)
  - Is either a leaf or an internal node

B-trees Example

Programming View

Check your definitions

- If B tree has maximum of 10 keys per node, then what is the maximum number of children that a node can have?
- If B tree has degree 4 then what is the maximum number of keys it can have at a node?
Searching a B Tree: Example #1
B tree order 2 (1 or 2 keys at node)

- 30 35
- 10 20
- 36
- 33 34

Find 34

Searching a BTree Example #2

- 30 35
- 39
- 37 50
- 36
- 33 34

Find 82

Height of B-tree of n nodes ?
- Compute worst case height of B-tree of degree m (order m-1), with total n nodes and height h
  - Root at level 0 has only 1 node
  - Level 1 has 2 nodes, each has at least m children
  - Level 2 has at least 2m nodes, each has at least m children

Height of B-tree of n nodes ?
- Compute worst case height of B-tree of degree m (order m-1), with total n nodes and height h
  - Root at level 0 has only 1 node
  - Level 1 has 2 nodes, each has at least m children
  - Level 2 has at least 2m nodes, each has at least m children
  - Level 3 has at least 2m^2 nodes, each has at least m children
  - Level h has 2m^(h-1) nodes

Therefore, n = 1 + 2 + 2m + 2m^2 + ... + 2m^(h-1)

Therefore, h = O (log_m n)
B-trees as External Data Structures

- Now that we have seen how a B-tree works as a data structure – how can it be used for an index.
- A regular B-tree can be used as an index by:
  - Each node in the B-tree stores not only keys, but also a record pointer for each key to the actual data being stored.
  - Could also potentially store the record in the B-tree node itself.
  - To find the data you want, search the B-tree using the key, and then use the pointer to retrieve the data.
  - No additional disk access is required if the record is stored in the node.

B-tree nodes & Index records

- B-tree is collection of blocks/nodes that contain:
  - Search-key (index) values
  - Data pointers to data records
  - Tree pointers to next/children node
  - Some info local to block
- Each B-tree node resides on one disk block
  - When tree is traversed, relevant nodes/blocks are brought into main memory
- Given this description, how might we calculate the best B-tree order.
  - Depends on disk block and record size.
  - We want a node to occupy an entire block.
Size of B-tree node

- What is the maximum number of bytes needed by a B-tree node of order $d$?
  - Tree pointers, data pointers, key fields, byte specific info
- What is the maximum number of bytes you can use for the node?
  - Size of the disk block!
  - One node fits in one disk block
- What is maximum no of entries of B-tree if key requires 5 bytes, and tree and data pointer need 10 bytes and disk block size=100 bytes?

Picking the order of a B-tree

- Best "packing": when entire node fits into one disk block with minimum space wasted
- Assume $P_d$ bytes needed to specify address of data pointer
- Assume $P_t$ bytes needed for address of tree pointer
- Assume key field requires $K$ bytes
- Assume disk block size = $B$ bytes
- Assume $x$ bytes needed for block-specific information

Order of B-tree

- Order $d$ tree has at most $2d+1$ tree pointers, $2d$ data pointers and $2d$ key values
- $x + 2d(K + P_d) + (2d+1)P_t \leq B$

Degree of B-tree

- Degree $m$ tree has at most $2m$ tree pointers, $2m - 1$ data pointers and $2m-1$ key values
- $x + (2m -1)(K + P_d) + (2m)P_t \leq B$
Example: Calculating order of B-tree

- Disk block size = 512 bytes, Key is varchar(20), and 32 bit (4 byte) addresses
- B = 512 bytes, x = 2, K = 20, Pt = Pd = 4

\[ 2 + 2d(20+4) + (2d+1)4 \leq 512 \]
\[ 56d + 6 \leq 512 \]
\[ d \leq 9 \]

Search in B-tree

- In-order search
- Time complexity?
  - Height of tree = \( \log(N) \)

Inserting Data into a B Tree

- Find correct leaf \( L \) : search \( O(\log N) \)
- Put data entry into \( L \).
- If \( L \) has enough space, done!
- Else, must split \( L \) (into \( L \) and a new node \( L_2 \))
  - Redistribute entries evenly, push up middle key.
  - Insert index entry pointing to \( L_2 \) into parent of \( L \).
- This can happen recursively

- Splits “grow” tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.
Inserting 8 Example: d=2

Want to insert here; no room, so split & push up:

2* 3* 7* 8*

5

Entry to be inserted in parent node.

B-tree and Multilevel Index

- Same? Or Different?

Multilevel

Key differences

- In multi-level index:
  - Data records at level 0 and index records at all other levels (contain key value but no data pointer)
  - The lowest level index records are ordered
    - Leaves form a linked list

- In B tree
  - Nodes contain both data pointers and tree (index) pointers
  - To get a list of sequential records, we have to search multiple levels of the tree
### B+ Tree: The DB World’s Favorite Index

- Insert/delete at \( \log F \cdot N \) cost
  - \( F = \text{fanout}, N = \# \text{leaf pages} \)
  - Keep tree height-balanced
- Minimum 50% occupancy (except for root).
- Each node contains \( d \leq m \leq 2d \) entries. \( d \) is called the \textit{order} of the tree.
- Supports equality and range searches efficiently.

### B+ Trees – Definition

- Similar structure to B-tree
- Distinguish between internal nodes and leaf nodes
  - Leaf nodes contain all search keys inserted into tree and connected by linked list in sorted order
  - Each leaf node:
    - Has no tree pointers
    - Has only search keys and data pointers
    - Has linked list pointer to next leaf node
  - Each internal node:
    - Has search keys
    - Has tree pointers
    - Has no data pointers
  - Search keys in internal nodes used only for navigation

---

### B+ trees

- Search keys in internal nodes are repeated in leaves
  - Similar to multilevel index
- Every search key does not occur as internal value
- To get the data record, we have to get to the leaf level
  - All searches will take \( O(h) \) where \( h \) is height of tree

### B+ trees: Definition

- B+-tree of order \( d \)
  - Each internal node or leaf node must contain at least \( d \) keys
  - Each internal or leaf node can contain at most \( 2d \) keys
  - Each internal node can contain at most \( 2d+1 \) pointers to next node in tree
  - Each leaf node must contain as many data pointers as there are keys in the node
  - Each leaf node has pointer to next leaf node in linked list
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...

Based on the search for 15*, we know it is not in the tree!

Picking the order of a B+-tree

- Same logic as for B-tree:
- Best "packing": when entire node fits into one disk block with minimum space wasted
- Assume P_t bytes needed to specify address of data/tree pointer
- Assume key field requires K bytes
- Assume disk block size = B bytes
- Assume x bytes needed for block-specific information

Size of B+-tree node

- What is the maximum number of bytes needed by a B+-tree node of order d?
  - pointers, key fields, byte specific info
- What is the maximum number of bytes you can use for the node?

Order of B-tree

- Order d tree has at most 2d key values and 2d+1 pointers (data pointers and linked list pointer for leaf node, and tree pointers for internal node)
- x + 2d*(K) + (2d+1)*P_t <= B
Example

- $B=512$ bytes, $x=2$, $K=20$, $P_t = 4$
- $2 + 2d(20) + (2d+1)4 \leq 512$
  - $48d + 6 \leq 512$
  - $d \leq 10.54$
- Higher than degree of B-tree!

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = 2,352,637$ records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

B+ trees..next

- Insertion
- Deletion
- Bulk loading
- Other indices

Inserting Data into a B+ Tree

- Find correct leaf $L$.
- Put data entry onto $L$.
  - If $L$ has enough space, done!
  - Else, must split $L$ (into $L$ and a new node $L_2$)
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to $L_2$ into parent of $L$.
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.
B+ tree insertion Algo: Outline

- Input (Key, Pointer to Data) \((K, P)\), B+ tree of order \(N\) (max \(2N\) values per node)
- Insert \((T, K, P)\)
- Search Tree to find Leaf \(L_i\) to insert \((K, P)\)
  - Insert into Leaf \(L_i\)
  - If number of entries in \(L_i\) \(<= 2N\) then done
  - Else Split\((L_i)\)
    - Find key value of \((N+1)\) entry \(Z_i\) in leaf
    - Split \(L_i\) into two leaves \(L_{i1}\) and \(L_{i2}\)
      - \(L_{i1}\) contains all entries less than or equal to \(Z_i\)
      - \(L_{i2}\) contains all entries greater than \(Z_i\)
      - Create pointer from \(L_{i1}\) to \(L_{i2}\)
      - Create pointer \(P_i\) pointing to \(L_{i1}\) and \(P_{i2}\) pointing to \(L_{i2}\)
      - If \(L_i\) was root node, then create new root and insert \(Z_i\) into new root and stop.
      - Else Insert \((T, Z_i)\)

Inserting 8\# into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Recall that all data items are in leaves, and partition values for keys are in intermediate nodes
  - Note difference between copy-up and push-up.

Inserting 8\# Example: Order 2 B+ tree

Root

\(13\) \(17\) \(24\) \(30\)

\(2\) \(3\) \(5\) \(7\) \(14\) \(16\) \(19\) \(20\) \(22\) \(24\) \(27\) \(29\) \(33\) \(34\) \(38\) \(39\)

Want to insert here; no room, so split & copy up:

8\#

Inserting 8\# Example: Copy up

Root

\(13\) \(17\) \(24\) \(30\)

\(2\) \(3\) \(5\) \(7\) \(14\) \(16\) \(19\) \(20\) \(22\) \(24\) \(27\) \(29\) \(33\) \(34\) \(38\) \(39\)

Want to insert here; no room, so split & copy up:

8\#

Entry to be inserted in parent node.
(Note that it is copied up and continues to appear in the leaf.)
Deleting Data from a B+ Tree

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has only d-1 entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.
- …details in notes

B+ Tree Summary

- B+ tree and other indices ideal for range searches, good for equality searches.
- High fanout (F) means depth rarely more than 3 or 4.
- Almost always better than maintaining a sorted file.
- Typically, 67% occupancy on average.
- Note: Order (d) concept replaced by physical space criterion in practice (“at least half-full”).
  - Records may be variable sized
  - Index pages typically hold more entries than leaves
Other Kinds of Indices

- Multidimensional indices
  - R-trees, kD-trees, …
- Text indices
  - Inverted indices…have you seen this before?
- Structural indices
  - Object indices: access support relations, path indices
  - XML and graph indices: dataguides, 1-indices, d(k) indices
    - Describe parent-child, path relationships

Speeding Operations over Data

- Three general data organization techniques:
  - Indexing
  - Sorting
  - Hashing

Technique 2: Hashing

- A familiar idea:
  - Requires “good” hash function (may depend on data)
  - Distribute data across buckets
  - Often multiple items in same bucket (buckets might overflow)
- Types of hash tables:
  - Static
  - Extendible (requires directory to buckets; can split)
  - Linear (two levels, rotate through + split; bad with skew)

Hash Files

- Basic idea: while domain is large, actual range of values is much smaller
  - If last name is 12 characters long, then domain is $12^{26}$ but actual number of names is much smaller
- In hashed file, records are distributed among a number of B buckets
- Hash function $h$, takes value as input and maps to one of B buckets
  - $h(k) = x, \ x \in \{0, \ldots B-1\}$
Hashing as indexing technique

- Key idea: partition records among B buckets

Hashing

- Hashing function $h$ applied to each record
  - Select "key" field (name, SSN, etc.) to hash
  - Each integer associated with a bucket number
    - $h(x) = 20$ means $x$ is in bucket 20
- Initially a disk block to each bucket
  - As bucket gets large, add more blocks to bucket
  - Chain the blocks in each bucket
- Hash table itself is stored on disk block(s)

Hashing: Buckets

Hash Functions

- Ideally, distribute the records evenly across buckets
  - As buckets get long, there are long overflow chains and it resembles heap
  - Dynamic hashing methods handle non-uniformity among buckets
- Preserving order in hash function extremely difficult
  - If $x < y$, then $h(x) < h(y)$ --- not possible
Operations on Hash files

- **Insert**: record with key field value \( x \)
  - Compute \( h(x) \) to find bucket, and insert into bucket
    - If no space, then get new block

- **Delete**:
  - Compute \( h(x) \) to find bucket, search for record, and delete

- **Search**
  - Compute \( h(x) \) to find bucket, and search in the bucket (all blocks)

Time analysis of Hash files

- **Assume** \( B \) buckets, \( n \) records
  - Assume uniform distribution across buckets
  - \( n/B \) records in each bucket
  - Number of blocks in each bucket = \( (n/Bp) \)
    - \( p \) is blocking factor – no. of records per block

- **Search/Lookup**: compute \( h(x) \) and search in bucket
  - Bucket is a Heap file of size \( (n/Bp) \)
  - Average time \( \frac{1}{2} (n/Bp) \); worst case \( (n/Bp) \)

- **Insert**: compute \( h(x) \) and insert into bucket in one disk access
  - 1 disk access

- **Delete**: search and delete

---

**Example:**

- 1 million records
- 1000 buckets, \( B = 1000 \)
- \( n/B = 1000 \) records per bucket
- Blocking factor \( p = 20 \) (record size 200, block size 4096)
- Bucket size = \( (1000/20) = 50 \) blocks
- Lookup time:
  - Average time = 25
  - Worst case = 50
- Insert time: 1
- Delete time: same as lookup time

---

Speeding Operations over Data

- **Three general data organization techniques**:
  - **Indexing**
  - **Sorting**
  - **Hashing**
**Why Sorting?**

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing GPA order
- **Sort-merge** join algorithm involves sorting.
- **Internal sorting**
  - Quicksort, heapsort, etc.
- Sorting Problem considered here: sort data much larger than main memory
  - Example: sort 1Gb of data with 1Mb of RAM.
    - Cannot hold all data in main memory – cannot use internal sort

---

**Sorting Algorithms**

- Quicksort, Heapsort

- Complexity?

- Use the same algorithm to sort our data files?

---

**External Sorting**

- Data to be sorted = data file stored on disk
- Sorted file must be stored back on disk

- **External sorting problem:**
  - Data file too large to fit in memory
  - Data must be sorted in pieces
  - Data usually heap size
  - Desired result: sorted file (sorted on some key field)

---

**Mergesort: Recall**

- Merge two sorted lists
  - (2,5,10,19)
  - (4,6,7,20)

- If we have two sorted files of length n/2 then binary merge gives sorted file of length n
**External Mergesort**

- File is set of pages/blocks
- Read one block at a time from disk into main memory
  - Assume 2 pages to hold a page from each input file and 1 page for output sorted block
Finally…

Observations

- How many disk I/Os? (how many read and write operations)
- Length of sorted file vs length of input file(s)?

External Sorting: Key ideas

- Individual blocks/pages can be easily sorted
  - Read them into memory and use internal sort algorithm
- Create runs
  - A run is a group/set of sorted blocks (i.e., sorted piece of a file)
  - Runs can be created by merging data from several blocks
- Merge shorter runs into longer runs
  - Merge runs to get sorted file
**External Mergesort**

- What is unsorted input?
  - Unsorted file of $n$ blocks is $n/2$ groups of 2 blocks each
  - Each group has two blocks to be sorted
- What if we merge $n/2$ groups, each group of 2 sorted blocks?
  - $n/4$ groups, each of run 4 sorted blocks
- What if we merge $n/4$ groups, each group of size 4 (i.e., 4 sorted blocks)
  - $n/8$ groups, each of size 8 blocks
  - How many disk I/Os?
- What if we merge 2 groups, each group is run of size $n/2$
  - We get sorted file of $n$ blocks
  - How many disk I/Os?
  - How many phases to get to this last step?

---

**External Sorting**

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, ..., etc.:
  - three buffer pages used.

---

**Two-Way External Merge Sort**

- Each pass we read, write each page in file.
- $N$ pages in the file => the number of passes = $\lceil \log_2 N \rceil + 1$
- Total cost is:
  $$2N\left(\lceil \log_2 N \rceil + 1\right)$$
- Idea: Divide and conquer: sort subfiles and merge
**General External Merge Sort**

- **How can we utilize more than 3 buffer pages?**
- To sort a file with $N$ pages using $B$ buffer pages:
  - **Pass 0:** use $B$ buffer pages. Produce $\lceil N/B \rceil$ sorted runs of $B$ pages each.
  - **Pass 2, ..., etc.:** merge $B-1$ runs.

**M-way External Mergesort**

- **Divide file into groups of $M$ blocks each**
- Merge each group into a sorted run: $N/M$ runs
- Create groups of $N/M$ runs, merge each group into single run

**Example:** $N=400$ block file, $M=4$ input buffers
- **Phase 1:** create groups of 4 blocks
  - 100 groups of 4 blocks each $= 100$ runs of 4 blocks each
- **Phase 2:** create groups of 4 runs; $100/4=25$ groups
  - 25 groups of 16 blocks each $= 25$ runs of 16 blocks each
- **Phase 3:** create groups of 32 runs; $25/4=7$ groups
  - 7 groups of up to 32 blocks each $= 7$ runs of 32 blocks
- **Phase 4:** groups of 64 runs; $7/4=2$ groups
  - 2 groups, up to 64 blocks $= 2$ groups of 64 blocks
- **Phase 5:** group of 128 runs; $2/4=1$ group
  - 1 group, up to 128 blocks; 4-way merge gives 400 blocks

**Cost of External Merge Sort**

- **Number of passes:** $1 + \lceil \log_M \lceil N \rceil \rceil$
- **Cost:** $2N \ast (\# \text{ of passes})$
Hashing and Sorting

- Sorted files:
  - Search on sorted field is fast
  - Insertion slow
  - Periodic reorganization needed

- Hashing:
  - Fast equality search
  - Bad for range search
  - Quick insert time
  - Challenge of finding a good hash function
    - Performance depends on how well records are divided amongst buckets

SQL CREATE INDEX

```
CREATE [UNIQUE] INDEX < index name >
[ index type ]
ON < table name> (<column name> [order]…)
```

- Creates index to support queries using that attribute
  - (column name)
  - order: ascending or descending
- Index type: USING { BTREE | HASH }
- can also create multi-attribute indices
  - Example: last name, first name

How to determine which indices to create?

- Query mix
- Tuning
  - Observe performance and if slow then create additional indices
- Create indices for every single field!

Next......Relational DBMS Physical schema design

- Tie it all together....
Query Processing

- Logical Design
  - Table schemas, SQL queries
- Physical Schema Design
  - Disk organization, (maybe) OS routines rewritten
  - Indices for data files
    - Use index to speedup search for data
- Putting them together: how does it work?
  - Input SQL query
  - System processes query and outputs result data

DBMS: Architecture – Physical Layer Structure

- Query parser
  - Parse SQL query, translate into intermediate form
    - Intermediate form – relational algebra
- Query optimizer
  - Rewrite query into an equivalent but more efficient form
- Query evaluation: Execution engine
  - How to implement the relational operators
- File manager (study in Operating systems)
  - Manages allocation of space on disk & data structures
  - Store data on disk using efficient scheme
  - Specify indices for the relations, Index structures, Buffer/Page management
- Recovery manager and Concurrency control (study in Operating systems)
  - Ensures database remains consistent despite failures
  - Ensures correct concurrent access to data

Processing the Query

SELECT * FROM STUDENT, Takes, COURSE WHERE STUDENT.sid = Takes.sID AND Takes.cid = cid

Architecture of Query Proc. Engine

You will study parsers in Foundations course

SQL query

Parse Query

Select Logical Plan

Select Physical Plan

Query Execution
**Query Processing Overview**

- The goal of the query processor is very simple:
  - Return the answer to a SQL query in the most efficient way possible given the organization of the database.

- Achieving this goal is anything but simple:
  - Different file organizations and indexing affect performance.
  - Different algorithms can be used to perform the relational algebra operations with varying performance based on the DB.
  - Estimating the cost of the query itself is hard.
  - Determining the best way to answer one query in isolation is challenging. How about many concurrent queries?

**Query Processing Steps**

- First, Query Optimization (Logical Plan)
  - Transform Relational Algebra expression

- Next, Query execution (Physical Plan)
  - Find efficient algorithms to implement the relational operators

---

**Logical Query Plan**

```
SELECT S.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
  Q.city='seattle' AND
  Q.phone > '5430000'
```

Note: Relational Algebra is internal representation... like "assembly"

---

**Physical Query Plan**

```
SELECT S.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
  Q.city='seattle' AND
  Q.phone > '5430000'
```

Query Plan:
- logical tree
- implementation choice at every node
- scheduling of operations.

Some operators are from relational algebra, and others (e.g., scan) are not.
Query Processing

- Goal?
  - Correct implementation
  - Performance

- Who takes care of optimizing the final executable code?

- Should user be aware of physical design parameters?

Time to process query depends on:
- Note: efficiency/time defined in terms of disk I/O ops

Query Optimization: Find efficient relational algebra expression
- Take input RA expression, transform into more efficient, equivalent expression.

Query Evaluation Plan (Implementation): Find efficient algorithm to implement the operators
- What algorithm to use to process the operators?...depends on index
- What index structure to use?
- How to store file?

Overview of Query Optimization

- A query plan: algebraic tree of operators, with choice of algorithm for each op

Two main issues in optimization:
- For a given query, which possible plans are considered?
  - Algorithm to search plan space for cheapest (estimated) plan
- How is the cost of a plan estimated?

- Ideally: Want to find best plan
- Practically: Avoid worst plans!

Query Optimization: First step

- Given relational algebra expression, find a more efficient, equivalent expression
- Develop a logical query plan
Query Optimization: First step

- Given relational algebra expression, find a more efficient, equivalent expression
- Develop a logical query plan
- Analogous to Code rewriting
  - (1) \( x = (a + 2b) + a \)
  - (2) \( x = 2(a + b) \)
  - Which one is faster/more efficient?

- Key concept: equivalence of relational algebra expressions

Logical Algebra Operators

- Union, intersection, difference
- Selection \( \sigma \)
- Projection \( \Pi \)
- Join \( \bowtie \)
- Duplicate elimination \( \delta \)
- Grouping \( \gamma \)
- Sorting \( \tau \)

Relational Algebra Equivalences

- Allow us to choose different join orders and to ‘push’ selections and projections ahead of joins.

- **Selections:**
  - \( \sigma_{c_1 \ldots c_n}(R) = \sigma_{c_n}(\ldots(\sigma_{c_1}(R))) \) (Cascade)
  - \( \sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_1}(\sigma_{c_2}(R)) \) (Commute)

- **Projections:**
  - \( \pi_{a_1}(R) = \pi_{a_1}(\ldots(\pi_{a_n}(R))) \)

- **Joins:**
  - \( R \bowtie T = (R \bowtie S) \bowtie T \) (Associative)
  - \( (R \bowtie S) = (S \bowtie R) \) (Commute)

Show that: \( R \bowtie (S \bowtie T) = (T \bowtie R) \bowtie S \)

More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection
- Selection between attributes of two arguments of a cross-product converts cross-product to a join
- A selection on ONLY attributes of \( R \) commutes with \( R \bowtie S \):
  - \( \sigma(R \bowtie S) = \sigma(R) \bowtie S \)
- If a projection follows a join \( R \bowtie S \), we can “push” it by retaining only attributes of \( R \) (and \( S \)) that are needed for the join or are kept by the projection
Optimization Rules

- Selections $\sigma$
- Project $\pi$
- Joins
- ...

Bank Example

- Customer (CustID, name, street, city)
- Loan (CustID, Loan-num, Amount, Branch-name)
- Deposit (CustID, Acct-num, Balance, Branch-name)
- Branch (Branch-name, Assets, City)

Selection Operations

- “Find assets and name of all banks who have depositors living in NYC”

- $\Pi_{\text{branch-name, assets}} (\sigma_{\text{customer.city='NYC'}} \text{customer} \bowtie \text{deposit} \bowtie \text{branch})$

Optimization goals

- Size of intermediate result customer $\bowtie$ deposit $\bowtie$ branch too large to be kept in main memory
  - System must access disk to read/write intermediate result
  - Size of int. result proportional to number disk accesses
  - Treat this as an overhead
- Optimization goal: Can process query more efficiently if you reduce size of intermediate result
Query interested only in tuples of customer that have city = ‘NYC’
- Perform selection before the Join
- By reducing number of tuples in customer, can reduce size of intermediate result

Optimized Query:
\[ \Pi_{\text{branch-name,assets}} (\sigma_{\text{customer.city} = \text{NYC}} (\text{Customer}) \bowtie \text{deposit} \bowtie \text{branch}) \]

Optimization Rule
- Rule: Apply selections as early as possible
- Find all assets and name of all branches that have depositors living in NYC and with a balance greater than $1000.
  - Two selections: on balance, and on city

Projections
- Rule: Perform projections early
  - Need to be careful not to drop attributes required for join conditions
  - Should you access a relation only for the projection?

\[ \Pi_{\text{branch-name,assets}} ( (\sigma_{\text{customer.city} = \text{NYC}} (\text{Customer}) \bowtie \text{deposit} \bowtie \text{branch}) ) \]

- What attributes do you need from Customer relation to process rest of query?
Some more useful equivalences...

- $\sigma_P (R_1 \cup R_2) = \sigma_P (R_1) \cup \sigma_P (R_2)$
- $\sigma_P (R_1 - R_2) = \sigma_P (R_1) - \sigma_P (R_2)$
- $(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3)$
- $(R_1 \cup R_2) = (R_2 \cup R_1)$
- ...all set equivalence properties hold.

Next...Implementing Relational Operators

- How do we compute selection, projection, joins?
- What algorithms to use?
  - Depends on indices chosen?
  - Should we create indices just to process the query?
- What about multiple queries?
  - Application consists of a number of queries
    - Frequency of use for each query

Size and Cost estimation...why?

- Can guide our choice of query plans.
- Cost Estimation: for each plan considered, estimate cost
  - estimate cost of each operation in plan tree.
    - Depends on input cardinalities.
    - also estimate size of result for each operation in tree
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
    - Assume can calculate a "reduction factor" (RF) for each selection predicate

Cost Estimation and System Statistics...more data to be stored!

- The system catalog:
  - Stores schema, constraints, ...
  - Statistics about the data in the DB
    - Used during query evaluation
System Statistics

- System stores statistics about relation/table sizes
  - Stored as part of the system catalog
- Typical Statistics stored:
  - \( n_r \): number of tuples/records of relation \( r \)
  - \( s_r \): size (bytes) of tuple in relation \( r \)
  - \( V(A,r) \): the number of distinct values that appear in relation \( r \) for attribute \( A \)
    - If \( A \) is key, then what is \( V(A,r) \)?
  - Typical assumption: Uniform distribution of values
    - All values have equal probability of occurrence…
    - Not true in many cases: Example of Names

Example:

```
select acct-num
from deposit
where branch-name = 'Downtown' and
ID='1234' and
balance > 1000 ;
```

Some statistics for example..

- 20 tuples of deposit fit on a disk block
- \( n_r \), deposit = 10,000
  - 10,000 tuples/rows in deposit, fits on 500 blocks
- \( V(\text{branch-name}, \text{Deposit}) \) = 50
  - 50 distinct values of branch name
- \( V(\text{custID}, \text{Deposit}) \) = 200
  - 200 distinct customers
- \( V(\text{balance}, \text{Deposit}) \) = 500
  - 500 different balances
- Clustered B+ tree index for branch name
  - Records clustered by branch name on disk
- Non-clustered B+ tree index for ID

Index structure properties..

- \( V(\text{branch name}, \text{Deposit}) \) = 50, therefore
  - we have 10,000/50 = 200 tuples in Downtown branch
- Using clustered index on branch-name, we need to read from 200/20 = 10 disk blocks
- Assume B+ tree index has max 20 pointers
  - degree \( m \)=10, and min entries \( =9 \) and max \( =19 \)
  - Between 3 and 6 leaf nodes for the 50 values
  - Tree of depth 2
- Therefore we need to read 2 index blocks + 10 data blocks
- Total access time = 12
**B+ tree Index on ID**

- Non-clustering index on ID
- \(10,000 \div (\text{V(ID, deposit)} = 10,000 \div 200 = 50\)
- 50 tuples have ID = 1234
- Across 50 disk blocks (unclustered)
- Assuming B+ tree degree \(m = 10\)
- 200 names, therefore between 11 and 22 leaf nodes
- Depth = 2, therefore 2 index blocks read
- Lowest level may point to a bucket of record pointers for the 50 blocks – recall: handling duplicates
- Total access time = 53 to read all tuples with ID = 1234
  - Adding 1 to read bucket of pointers

**So what is your execution plan?**

- Search using ID
  - 53 disk reads
- Search using branch name
  - 12 disk reads
- Any other scheme
- Do we need to access all the 60 disk blocks to determine if record satisfies both conditions?
  - Bucket of pointers method??

**Manipulation of bucket pointers**

- All conditions have to be satisfied in query
  - Tuple must have branchname = Downtown and ID = 1234
- Use branch-name index to retrieve pointers to all records with branch-name = Downtown
  - Time = 2 index blocks read
  - Output = \(P_1\) (contains 200 pointers)
- Use ID index to retrieve pointers to all records with ID = 1234
  - Time = 3 index blocks read
  - Output = \(P_2\) (contains 50 pointers)
  - Next ??

**Intersecting pointers**

- What is the set \(P_1 \cap P_2\)
- Addresses of all tuples that have ID = 1234 and branch-name = Downtown
- Now fetch these and select only those with balance > 1000
- How many to fetch = size of \(P_1 \cap P_2\)
- \(V(\text{ID, deposit}) = 200, V(\text{branch-name, dep}) = 50\)
  - Assume uniform distribution
  - 1 record out of every \(V(\text{ID, deposit}) \times V(\text{branch-name, dep})\) will satisfy both,
  - for a total of \((\text{size of } P_1 \cap P_2) \times V(\text{branch-name, dep})\)
  - Therefore \(10,000 \div (50 \times 200) = 1\) will satisfy both
  - size of \(P_1 \cap P_2\) = 1
  - Total time = 2 + 3 + 1 = 6 disk accesses!
JOIN algorithms

- JOINS are very expensive
  - Scanning entire relation is expensive

Simple Join Algorithms

- Simple nested loops
  - Scan tuples in one file, and for each one, scan the other file to find matching tuples

1. for each tuple $x \in$ PASSENGER
2. for each tuple $y \in$ FLIGHT
   - if $x$.FLT_ID = $y$.FLT_ID
     - then tuple in result

Recall sizes

- PASSENGER relation
  - Number of tuples $n_p = 100,000$ tuples
  - 20 tuples per block
  - Number of blocks $n_p = 5000$ blocks

- FLIGHT relation
  - Number of tuples $n_f = 50,000$ tuples
  - 50 tuples per block
  - Number of block $n_f = 1000$ blocks

Time taken

- Number of iterations inner loop?
- $(r_p \times n_f) + n_p$
  - $10^5 \times 5000 = 555.58$ hours!!!
- What if FLIGHT was outer relation
- $(r_f \times n_p) + n_f$
  - $25 \times 10^5 + 1000 = 1388.89$ hours

- Using simple nested looks is a really stupid way of implementing joins!
- Need better ways!...NEXT...
Block-nested Join

- Do we have to iterate over tuples?
  - Can we change the iteration parameter?
- How about a disk block at a time?
  - Iterate block by block
  - Fetch block, do all computations for a block
  - Fetch next block, ...
  - Read first block of passenger, fetch all blocks of Flight and check for join

Block-nested Join

1. for each block $x \in$ PASSENGER
2. for each block $y \in$ FLIGHT
3. if a pair in each block matches
   then put tuple in result of Join

How many iterations of inner loop?
How many iterations of outer loop?

Block-nested Join: Memory usage

- What if we had more memory: $M$ buffers/pages in memory
  - We used 2 input buffers (one for each relation)
- $M$ buffers in memory
  - Use $K$ for PASSENGER, $(M-K)$ for FLIGHT
  - Read first $K$ of PASSENGER
  - Process all blocks of FLIGHT
  - Read next $K$ blocks of PASSENGER
  - Process all blocks of FLIGHT
- …
- How many iterations?
  - $(n_r/n_k)$ groups of PASSENGER, each time $n_k$ blocks of FLIGHT
  - Total: $n_r/n_k \times n_f$
- What should $K$ be? ...
  - Large as possible
  - In example: let $M=200$, so choose $K=199$
- Time $= (5000/199)^2 \times 1000 + 5000 \times 28 \times 1000 + 5000 \times 31,000 = 10.3$ minutes
Block-nested Joins...

- Suppose FLIGHT is 100 blocks
- Then time = \( \frac{n_I}{K} \cdot n_F + n_F = \frac{5000}{199} \cdot 100 + 5000 \)
  \( = 7600 \) blocks
- What happens if we switch order
  - FLIGHT is inner relation
  - Time = \( \frac{n_F}{K} \cdot n_I + n_F \)
  \( = \frac{100}{199} \cdot 5000 + 1000 = 5000 + 1000 = 6000 \)
- Keep smaller relation in outer loop!

Index Nested Loop Joins

for each block in FLIGHT
for each tuple in current block
extract search key
search for matching tuple in PASSENGER
place in result if match

Conclusion on Join Algorithms

- Simple iterative methods are not fast enough
- Using index (hash or B+tree) makes them faster, but still slow
- Look at other ways.....

- Create an file organization just to process the join?

Conclusion

- Every relational algebra operator may be implemented using many different algorithms. The performance of the algorithms depend on the data, the database structure, and indexes.

- The actual algorithm is chosen by the query optimizer based on its query plan and database statistics.
Tuning....Manual Intervention

- DBA can take steps to improve performance of “tough” queries
  - Decide to create/remove indices
  - Decide types of indices
  - Use tools or system options to re-organize data on disk
- DBA needs to understand system workload
  - Frequently used queries and frequency
  - Updates and update frequencies
  - User requirements/complaints
- Identify Important Queries/Users and focus optimizations on them