Relational Model Definitions

- A **relation** is a table with columns and rows.
- An **attribute** is a named column of a relation.
- A **tuple** is a row of a relation.
- A **domain** is a set of allowable values for one or more attributes.
- The **degree** of a relation is the number of attributes it contains.
- The **cardinality** of a relation is the number of tuples it contains.
- A **relational database** is a collection of normalized relations with distinct relation names.

Relational Query Languages

- **Query languages:**
  - Allow specification of schemas and constraints
  - Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages ≠ programming languages!
  - QLs not expected to be “Turing complete”.
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

SQL: Structured Query Language

The standard language for relational data

- Invented by folks at IBM, esp. Don Chamberlin
- Actually not a great language…
- Beat a more elegant competing standard, QUEL, from Berkeley

Separated into a DML & DDL

SQL DML component based on relational algebra & calculus

- Data definition (DDL) – to define schema/tables
  - Define Schema
  - Define Constraints
**SQL**
- Based primarily on Relational Algebra with some features from Relational Calculus
- Components: Data definition language, Manipulation language
- Other SQL features
  - Transaction definition: end of query by default
  - Security, Views, Index
- Embedded SQL – embed SQL commands in a general purpose language
- Database connectivity packages allow queries to be passed to DB from applications – JDBC

**SQL Basic Rules...read up on SQL syntax**
- Some basic rules for SQL statements:
  - 1) There is a set of reserved words that cannot be used as names for database objects. (e.g. SELECT, FROM, WHERE)
  - 2) SQL is case-insensitive.
    Only exception is string constants. 'FRED' not the same as 'fred'.
  - 3) SQL is free-format and white-space is ignored.
  - 4) The semi-colon is often used as a statement terminator, although that is not always required.
  - 5) Date and time constants have defined format:
    Dates: 'YYYY-MM-DD' e.g. '1975-05-17'
    Times: 'hh:mm:ss[.f]' e.g. '15:00:00'
    Timestamp: 'YYYY-MM-DD hh:mm:ss[.f]' e.g. '1975-05-17 15:00:00'
  - 6) Two single quotes '' are used to represent a single quote character in a character constant. e.g. 'Master''s'.

**Basic SQL Query**

| SELECT | [DISTINCT] attribute-list |
| FROM | relation-list |
| WHERE | qualification/predicate |

- **relation-list** A list of relation names (possibly with a range-variable, i.e., tuple variable, after each name).
- **attribute-list** A list of attributes of relations in relation-list
- **Qualification/predicate** Comparisons (Attr op const or Attr1 op Attr2, where op is one of <, >, =, ≤, ≥, ≠ ) combined using AND, OR and NOT.
- **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

**SQL Queries**
- Querying with SQL is performed using a **SELECT** statement.
  The general form of the statement is:

```
SELECT A₁, A₂, …, Aₙ attributes in result
FROM R₁, R₂, …, Rₘ tables in query
WHERE (condition)
```

- **Notes:**
  - 1) The *** is used to select all attributes.
  - 2) Combines the relational algebra operators of selection, projection, and join into a single statement.
  - 3) Comparison operators: =, !=, >, >=, <, <=.
SQL and Relational Algebra

- The **SELECT** statement can be mapped directly to relational algebra.

- **SELECT** \( A_1, A_2, \ldots, A_n \) /* this is projection
- **FROM** \( R_1, R_2, \ldots, R_m \) /* this is the selection op
- **WHERE** \( P \) /* this is cartesian product

is equivalent to:

\[
\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(\prod_{i=1}^m R_i))
\]

Conceptual Evaluation Strategy

- **Semantics** of an SQL query defined in terms of the following conceptual evaluation strategy:
- Compute the cross-product of **relation-list**.
- Discard resulting tuples if they fail **predicate qualifications**.
- Delete attributes that are not in **target attribute-list**.
  - If **DISTINCT** is specified, eliminate duplicate rows.
  - SQL allows duplicates in relations (unlike Rel. Algebra)
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.

Simple SQL Query

```sql
SELECT *
FROM Product
WHERE category='Gadgets'
```

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

**Algorithm:** Scan each tuple in table and check if matches condition in **WHERE** clause.

Simple SQL Query

```sql
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
Duplicates in SQL
SQL returns ‘bag of words’
duplicates allowed in contrast to relational algebra
To remove duplicates use DISTINCT clause:
SELECT DISTINCT title
FROM emp;

Eliminating Duplicates
SELECT DISTINCT category
FROM Product

Compare to:
SELECT category
FROM Product

Category
Gadgets
Photography
Household

Expressions and Strings
SELECT S.sid
FROM Students S
WHERE S.name LIKE ‘%Sam%’
• Illustrates use of arithmetic expressions and string pattern matching: *Find students whose name includes ‘Sam’.*
• LIKE is used for string matching. ‘_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.

• Find students whose name begins with S and at least three characters:
  • Replace with ‘S__ %’

Ordering Result Data
• The query result returned is not ordered on any attribute by default. We can order the data using the ORDER BY clause:
  • STUDENTS [ sid, name]

SELECT name
FROM students
ORDER BY name ASC

• ‘ASC’ sorts the data in ascending order, and ‘DESC’ sorts it in descending order. The default is ‘ASC’.
• The order of sorted attributes specified by the ‘sort key’ (name in above example) NULL is normally treated as less than all non-null values.
Ordering Result Data using multiple attributes

- Can define sort on major and minor sort keys
  - Corresponds to “filing order” of k-tuples

emp: [ ssn, ename, salary]

- The order of sorted attributes is significant. The first attribute specified is sorted on first, then the second attribute is used to break any ties, etc.

```
SELECT ename, salary
FROM emp
WHERE salary > 30000
ORDER BY salary DESC, ename ASC
```

Questions: Write out SQL Queries for 1—4

- Find all rows in deposit where balance is greater than $1000
- Find names of customers whose name is the same as the street they live on.
- Find names and IDs of customers whose name begins with a C
- Find IDs of all customers who have Loans between 1200 and 2500

- Do NOT code…write your answers on the paper/text file

Join Query Example

- Multiple tables can be queried in a single SQL statement by listing them in the FROM clause.
  - Note that if you do not specify any join condition to relate them in the WHERE clause, you get a cross product of the tables.

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
    AND Price <= 200
```

Joins

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

Join between Product and Company
Joins
Find all products under $200 manufactured in Japan; return their names and prices.

Product
<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

Company
<table>
<thead>
<tr>
<th>CName</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName AND Country = 'Japan'
AND Price <= 200

Renaming and Aliasing
- Equivalent to rename operator in relational algebra
- Often it is useful to be able to rename an attribute in the final result (especially when using calculated fields). Renaming is accomplished using the keyword AS:

  ```sql
  SELECT ename, salary AS pay
  FROM emp
  WHERE ename = 'A. Lee'
  ```

  Result
  
<table>
<thead>
<tr>
<th>name</th>
<th>pay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Lee</td>
<td>192.31</td>
</tr>
<tr>
<td>A. Lee</td>
<td>923.08</td>
</tr>
</tbody>
</table>

  Note: AS keyword is optional.

Tuple/Range variables
- Concept of tuple/range variables borrowed from relational calculus
  - Tuple t of type R: t ∈ R
  - What about x ∈ R, y ∈ R
- It performs the job of the rename operator in relational algebra
  - One variable with name x and one with name y
- Need to worry about scope of tuple variables when we have nested queries

Renaming & Tuple Variables

Person(pname, address, worksfor)
Company(cname, address)

```sql
SELECT DISTINCT pname, address
FROM Person, Company
WHERE worksfor = cname
```
Tuple Variables

Person(pname, address, worksfor)
Company(cname, address)

SELECT DISTINCT x.pname, y.address
FROM Person, Company
WHERE worksfor = cname;

x is a copy of Person, y is a copy of Company
P is a variable of ‘type’ Person C is a variable of ‘type’ Company

SELECT DISTINCT P.pname, C.address
FROM Person P, Company C
WHERE P.worksfor = C.cname;

Renaming: Joining table with itself

- Aliases/Tuple variables must be used when relation has to be ‘joined’ with itself – i.e., two or more copies of the same table are needed. Using aliases allows you to uniquely identify what table you are talking about.

Example: Return names of employees and their managers.

SELECT E.ename, M.ename
FROM emp as E, emp as M
WHERE E.supereno = M.eno;

- Don’t need the AS keyword above

Meaning (Semantics) of SQL Queries with tuple variables

SELECT a1, a2, ..., ak
FROM R1 x1, R2 x2, ..., Rn xn
WHERE Conditions
Answer = {}
for x1 in R1 do
  for x2 in R2 do
    ....
    for xn in Rn do
      if Conditions
        then Answer = Answer \cup \{(a1,...,ak)\}
  return Answer

Tuple variables

- Find students who are taking the same course as Sam with sid=1234.
- Need to access Enrolled table twice
  - Once to extract courses (X) taken by Sam with ID=1234
  - Second time to find students who are taking these X courses
  - Define two “variables” A,B of ‘type’ Enrolled
  - B is variable that corresponds ID 1234 and its cid field is equal to “X”
  - A is a variable whose CID is equal to “X”
Exercises 2: Write SQL Queries for Ques. 5—7

- Find names of all customers, sorted (ascending) by name, who have Loans between 1200 and 2500.
- Find the names and IDs of customers who live on the same street as customer(s) named Lennon
- Find names of customers who have an account at the same branch as a customer named Lennon.

Exercise 3: Code your solutions on the MySQL server

- Code your solutions, run them.
- Submit your code on github
- Deadline: 2:30pm

**BETWEEN**

- Sometimes the condition in the WHERE clause will request tuples where one attribute value must be in a range of values.
- Example: Return the employees who make at least $20,000 and less than or equal to $45,000.

  ```sql
  SELECT ename
  FROM emp
  WHERE salary BETWEEN 20000 AND 45000
  ```

- We can use the keyword **BETWEEN** instead:

  ```sql
  SELECT ename
  FROM emp
  WHERE salary BETWEEN 20000 AND 45000
  ```

**IN Operator**

- To specify that an attribute value should be in a given set of values, the IN keyword is used.
- Example: Return all employees who are in any one of the departments {'D1', 'D2', 'D3'}.

  ```sql
  SELECT ename
  FROM emp
  WHERE dno IN ('D1', 'D2', 'D3')
  ```

- Note that this is equivalent to using OR:

  ```sql
  SELECT ename
  FROM emp
  WHERE dno = 'D1' OR dno = 'D2' OR dno = 'D3'
  ```

- more practical uses of **IN** and **NOT IN** when we study nested subqueries.
Set Operations

- The set operations of union, intersection, and difference are used to combine the results of two SQL queries.
  - UNION, INTERSECT, EXCEPT
  - Note: UNION ALL returns all rows

- Example: Return the sid of students who are either taking course with cid=123 or course with cid=345.

```sql
(SELECT sid
 FROM students
 WHERE cid = '123')
UNION
(SELECT sid
 FROM students
 WHERE cid = '345');
```

NULLS in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
  - Value does not exist
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if it can be null (nullable attribute) or not
  - NOT NULL after declaring attribute domain
- How does SQL cope with tables that have NULLs?

Set Operations

- MySQL does NOT support MINUS (EXCEPT) and INTERSECT
  - Can implement using other operators

Null Values

- If x = NULL then 4*(3-x)/7 is still NULL
- If x = NULL then x="Joe" is UNKNOWN
- In SQL there are three boolean values:
  ```sql
  FALSE = 0
  UNKNOWN = 0.5
  TRUE = 1
  ```
Null Values

- $C_1 \text{ AND } C_2 = \min(C_1, C_2)$
- $C_1 \text{ OR } C_2 = \max(C_1, C_2)$
- NOT $C_1 = 1 - C_1$

**Rule in SQL:** include only tuples that yield TRUE

```sql
SELECT *
FROM Person
WHERE (age < 25) AND (height > 6 OR weight > 190)
```

E.g.
- age=20
- height=NULL
- weight=200

Null Values

Unexpected behavior:

```sql
SELECT *
FROM Person
WHERE age < 25 OR age >= 25
```

Some persons are not included!

Null Values

Can test for NULL explicitly:
- $x \text{ IS NULL}$
- $x \text{ IS NOT NULL}$

```sql
SELECT *
FROM Person
WHERE age < 25 OR age IS NULL
```

Now it includes all persons

More SQL stuff to come...

- Nested Queries
- Set membership operations
- Calculate columns
- Aggregate operators
- Views
- authorization