**Midterm Administrivia**

- Midterm is on October 26, 2006 during class
- One (1) sheet of notes, both sides
- Covers everything up to and including physical database tuning from last Thursday’s lecture
- No electronic devices -- calculators, phones, iPads, palms, blackberries, PDAs, external brain packs, etc.
- Please use the restroom before the exam --- leaving the exam during the middle will only be allowed under extreme circumstances
- Study hard, have fun, and good luck!

---

**Motivation**

- Why use a DBMS? OS provides RAM and disk
  - Concurrency
  - Recovery
  - Abstraction, Data Independence
  - Query Languages
  - Efficiency (for most tasks)
  - Security
  - Data Integrity

---

**Relational Database: Definitions**

- Relational database: a set of relations.
- Relation: made up of 2 parts:
  - Schema: specifies name of relation, plus name and type of each column.
    - E.g. Students(id, string, name, string, login: string, age: integer, gpa: real)
  - Instance: a table, with rows and columns.
    - Rows = cardinality
    - Fields = degree / arity
- Can think of a relation as a set of rows or tuples.
  - I.e., all rows are distinct

---

**SQL - A language for Relational DBs**

- SQL (a.k.a. “Sequel”), standard language
- Data Definition Language (DDL)
  - create, modify, delete relations
  - specify constraints
  - administer users, security, etc.
- Data Manipulation Language (DML)
  - Specify queries to find tuples that satisfy criteria
  - add, modify, remove tuples

---

**Keys**

- Keys are a way to associate tuples in different relations
- Keys are one form of integrity constraint (IC)

**Enrolled**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>5360</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>5360</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>5360</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>5360</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

**Students**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>5360</td>
<td>Jones</td>
<td>genetics</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>5360</td>
<td>Smith</td>
<td>smith@uoc</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>5360</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

---

**FOREIGN Key**

**PRIMARY Key**
Primary Keys

- A set of fields is a superkey if:
  - No two distinct tuples can have same values in all key fields
- A set of fields is a key for a relation if:
  - It is a superkey
  - No subset of the fields is a superkey
- What if >1 key for a relation?
  - One of the keys is chosen (by DBA) to be the primary key. Other keys are called candidate keys.
- E.g.
  - sid is a key for Students.
  - What about name?
  - The set (sid, gpa) is a superkey.

Foreign Keys, Referential Integrity

- **Foreign key**: Set of fields in one relation that is used to ‘refer’ to a tuple in another relation.
  - Must correspond to the primary key of the other
  - Like a ‘logical pointer’.
- If all foreign key constraints are enforced, referential integrity is achieved (i.e., no dangling references.)

Integrity Constraints (ICs)

- **IC**: condition that must be true for any instance of the database; e.g., domain constraints.
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A legal instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

Basic SQL Query

- **relation-list**: List of relation names
  - possibly with a range variable after each name
- **target-list**: List of attributes of tables in relation-list
- **qualification**: Comparisons combined using AND, OR and NOT.
- **DISTINCT**: optional keyword indicating that the answer should not contain duplicates.

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```

Query Semantics

1. **FROM**: compute cross product of tables.
2. **WHERE**: Check conditions, discard tuples that fail.
3. **SELECT**: Delete unwanted fields.

*Note:* Probably the least efficient way to compute a query!
- Query optimizer will find more efficient ways to get the same answer.

Find sailors who've reserved at least one boat

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing S.sid by S.name in the SELECT clause?
- Would adding DISTINCT to this variant of the query make a difference?

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```
Arithmetic Expressions

SELECT S.age, S.age + S.age AS age1, 2*S.age AS age2
FROM Sailors S
WHERE S.name = 'dustin'

SELECT S1.name AS name1, S2.name AS name2
FROM Sailors S1, Sailors S2
WHERE S1.rating = S2.rating - 1

String Comparisons

SELECT S.name
FROM Sailors S
WHERE S.name LIKE '_KB'

* stands for any one character and % stands for 0 or more arbitrary characters.

SQL Overview

- CREATE TABLE <name> (<field> <domain>, ...)
- INSERT INTO <name> (<field names>) VALUES (<field values>)
- DELETE FROM <name>
  WHERE <condition>
- UPDATE <name>
  SET <field name> = <value>
  WHERE <condition>
- SELECT <fields>
  FROM <name>
  WHERE <condition>

Queries With GROUP BY

- The target-list contains
  - (i) list of column names &
  - (ii) terms with aggregate functions (e.g., MIN(S.age)).
- column name list (i) can contain only attributes from
  the grouping-list.
- To generate values for a column based on groups of
  rows, use aggregate functions in SELECT statements with
  the GROUP BY clause

GROUP BY grouping-list

Group By Examples

For each rating, find the average age of the sailors
SELECT S.rating, AVG(S.age)
FROM Sailors S
GROUP BY S.rating

For each rating find the age of the youngest
sailor with age >= 18
SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating

Conceptual Evaluation

- The cross-product of relation-list is computed, tuples
  that fail qualification are discarded, 'unnecessary'
  fields are deleted, and the remaining tuples are
  partitioned into groups by the value of attributes in
  grouping-list.
- One answer tuple is generated per qualifying group.
- If DISTINCT is specified, drop duplicate answer
  tuples.

SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
Queries With GROUP BY and HAVING

- Use the HAVING clause with the GROUP BY clause to restrict which group-rows are returned in the result set.

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

Conceptual Evaluation

- Form groups as before.
- The group-qualification is then applied to eliminate some groups.
  - Expressions in group-qualification must have a single value per group!
    - That is, attributes in group-qualification must be arguments of an aggregate op or must also appear in the grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Summary

- Relational model has well-defined query semantics
- SQL provides functionality close to basic relational model (some differences in duplicate handling, null values, set operators, …)
- Typically, many ways to write a query
  - DBMS figures out a fast way to execute a query, regardless of how it is written.

Formal Relational Query Languages

Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:

- **Relational Algebra**: More operational, very useful for representing execution plans.
- **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-procedural, declarative.)

- *Understanding Algebra & Calculus is key to understanding SQL, query processing!*

Relational Algebra: 5 Basic Operations

- **Selection** (σ) Selects a subset of rows from relation (horizontal).
- **Projection** (π) Retains only wanted columns from relation (vertical).
- **Cross-product** (×) Allows us to combine two relations.
- **Set-difference** (−) Tuples in r1, but not in r2.
- **Union** (∪) Tuples in r1 or in r2.

Since each operation returns a relation, operations can be composed! (Algebra is "closed".)

Compound Operator: Intersection

- In addition to the 5 basic operators, there are several additional "Compound Operators"
  - These add no computational power to the language, but are useful shorthands.
  - Can be expressed solely with the basic ops.
  - Intersection takes two input relations, which must be union-compatible.
  - How to express it using basic operators?
    - \( R \cap S = R - (R - S) \)
Compound Operator: Join

- Joins are compound operators involving cross product, selection, and (sometimes) projection.
- Most common type of join is a “natural join” (often just called “join”). \( R \bowtie S \) conceptually is:
  - Compute \( R \times S \)
  - Select rows where attributes that appear in both relations have equal values
  - Project all unique attributes and one copy of each of the common ones
- Note: Usually done much more efficiently than this.

Other Types of Joins

- **Condition Join** (or “theta-join”): Performs a join based on any specified condition \( \theta \) (hence the name, “theta-join”)
- **Result schema** is the same as that of performing a cross-product.
- **Eq-join**: condition \( c \) contains only the conjunction of equalities.
- May have fewer tuples than cross-product.

Summary

- **Relational Algebra**: a small set of operators mapping relations to relations
  - Operational, in the sense that you specify the explicit order of operations
  - A closed set of operators: Can mix and match.
- Basic ops includes: \( \sigma, \pi, \times, \cup, \mid \)
- Important compound ops: \( \cap, \bowtie \)

Relational Calculus

- **Query** has the form: \( \{ T \mid \rho(T) \} \)
  - \( \rho(T) \) is a formula containing \( T \)
- **Answer** = tuples \( T \) for which \( \rho(T) = \text{true} \).

Formulae

- **Atomic formulae**:
  - \( R \in \text{Relation} \)
  - \( R.a \ op \ S.b \)
  - \( R.a \ op \ \text{constant} \)
  - \( \ldots \ op \) is one of \( <, >, =, \leq, \geq, \neq \)
- A formula can be:
  - an atomic formula
  - \( \neg p, p \land q, p \lor q, p \Rightarrow q \)
  - \( \exists R(p(R)) \)
  - \( \forall R(p(R)) \)

Free and Bound Variables

- **Quantifiers**: \( \exists \) and \( \forall \)
- Use of \( \exists X \) or \( \forall X \) **binds** \( X \)
  - A variable that is **not bound** is free.
- Recall our definition of a **query**:
  - \( \{ T \mid \rho(T) \} \)
- **Important restriction**:
  - \( T \) must be the only free variable in \( \rho(T) \).
  - all other variables must be bound using a quantifier.
Simple Queries

- Find all sailors with rating above 7
  \[\{ S \mid S \in Sailors \land S.\text{rating} > 7\}\]

- Find names and ages of sailors with rating above 7.
  \[\{ S \mid \exists S1 \in Sailors(S1.\text{rating} > 7 \land S.\text{name} = S1.\text{name} \land S.\text{age} = S1.\text{age})\}\]

  Note: \(S\) is a variable of 2 fields (i.e., \(S\) is a projection of \(\text{Sailors}\)).

Joins

- Find sailors rated > 7 who’ve reserved boat #103
  \[\{ S \mid S \in Sailors \land S.\text{rating} > 7 \land \exists R(\exists R \in Reserves \land R.\text{sid} = S.\text{sid} \land R.\text{bid} = 103)\}\]

Joins (continued)

- Find sailors rated > 7 who’ve reserved a red boat
  - This may look cumbersome, but it’s not so different from SQL!
  \[\{ S \mid S \in Sailors \land S.\text{rating} > 7 \land \exists R(\exists R \in Reserves \land R.\text{sid} = S.\text{sid} \land \exists B(\exists B \in Boats \land B.\text{bid} = R.\text{bid} \land B.\text{color} = \text{‘red’})\}\]

Universal Quantification

- Find sailors who’ve reserved all boats
  \[\{ S \mid S \in Sailors \land \forall B (\exists R \in Reserves \land (S.\text{sid} = R.\text{sid} \land B.\text{bid} = R.\text{bid}))\}\]

A trickier example…

- Find sailors who’ve reserved all Red boats
  \[\{ S \mid S \in Sailors \land \forall B (\exists R \in Reserves \land (S.\text{sid} = R.\text{sid} \land B.\text{bid} = R.\text{bid}))\}\]

A trickier example…

- Alternatively...
  \[\{ S \mid S \in Sailors \land \forall B (B.\text{color} \neq \text{‘red’} \lor \exists R(\exists R \in Reserves \land (S.\text{sid} = R.\text{sid} \land B.\text{bid} = R.\text{bid}))\}\]\
\[ a \implies b \text{ is the same as } \neg a \lor b \]

\[
\begin{array}{c|c|c}
   & T & F \\
\hline
T & T & F \\
F & T & T \\
\end{array}
\]

A Remark: Unsafe Queries

- \exists \text{ syntactically correct calculus queries that have an infinite number of answers! Unsafe queries.}
- e.g., \[ |S \setminus \{S \in \text{Sailors}\} \]
- Solution???? Don't do that!

Expressive Power

- Expressive Power (Theorem due to Codd):
  - Every query that can be expressed in relational algebra can be expressed as a safe query in relational calculus; the converse is also true.
- Relational Completeness:
  - Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.
  - (actually, SQL is more powerful, as we will see...)

Summary

- Formal query languages — simple and powerful.
  - Relational algebra is operational
    - used as internal representation for query evaluation plans.
  - Relational calculus is "declarative"
    - query = "what you want", not "how to compute it"
  - Same expressive power
    - \[ \implies \] relational completeness.
- Several ways of expressing a given query
  - a query optimizer should choose the most efficient version.

Coming Attractions...

- Be prepared for disks and files, indexing, and sorting (Alex) …
- Followed by implementation of relational operators, query optimization, and physical design (Nathan)

- But first, a quick review of information retrieval.
- Hooray for text search!