CMSC424: Database Design
Introduction/Overview

Professor: Pete Keleher
keleher@cs.umd.edu

What we will cover...

- We will mainly discuss structured data
  - That can be represented in tabular forms (called Relational data)
  - We will spend some time on XML
  - We will also spend some time on Mapreduce-like stuff

- Still the biggest and most important business (?)
  - Well defined problem with really good solutions that work
    - Contrast XQuery for XML vs SQL for relational
  - Solid technological foundations

- Many of the basic techniques however are directly applicable
  - E.g. reliable data storage etc.
  - Cf. Many recent attempts to add SQL-like capabilities, transactions to Mapreduce and related technologies
    - E.g., Spark DataFrames
What we will cover…

- representing information
  - data modeling
  - semantic constraints
- languages and systems for querying data
  - complex queries & query semantics
  - over massive data sets
- concurrency control for data manipulation
  - ensuring transactional semantics
- reliable data storage
  - maintain data semantics even if you pull the plug
  - fault tolerance
Summary

Why study databases?
- Shift from computation to information
  - Always true in corporate domains
  - Increasing true for personal and scientific domains
- Need has exploded in recent years
  - Data is growing quickly
- Solving data management problems is key

Database Management Systems provide
- Data abstraction: Key in evolving systems
- Guarantees about data integrity
  - In presence of concurrent access, failures…
- Speed !!

Logistics

Professor: Peter Keleher
- 4157 AV Williams Bldg
- keleher@cs.umd.edu
- Class Webpage:
  - http://www.cs.umd.edu/class
  - Or http://424.kelehers.me,

Email to me: write CMSC424 in the title.
Logistics

- Grading
  - Approximate cut-offs
    - 85+: A
    - 75+: B
    - 65+: C

- Most had 40/50 on non-exams last two times
  - Exams are usually somewhat harder (no curves)
  - We would enforce a minimum passing grade on the total exam score

Some To-Dos

- Sign up for Piazza!

- Set up the computing environment (project0), and make sure you can run Vagrant+VirtualBox, PostgreSQL, IPython, etc.

- Upcoming:
  - Reading Quiz 1 (Due Monday)
  - Project 1: SQL (Sept 15)
Overview of modeling

Relational Model (Chapter 2)
- Basics
- Keys
- Relational operations
- Relational algebra basics

SQL (Chapter 3)
- Basic Data Definition (3.2)
- Setting up the PostgreSQL database
- Basic Queries (3.3-3.5)
- Null values (3.6)
- Aggregates (3.7)
History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

- Several alternative syntaxes to write the same queries

Different Types of Constructs

- **Data definition language** (DDL): Defining/modifying schemas
  - **Integrity constraints**: Specifying conditions the data must satisfy
  - **View definition**: Defining views over data
  - **Authorization**: Who can access what
- **Data-manipulation language** (DML): Insert/delete/update tuples, queries
- **Transaction control**:
- **Embedded SQL**: Calling SQL from within programming languages
- **Creating indexes, Query Optimization control…**
Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- Also: other information such as
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

SQL Constructs: Data Definition Language

- CREATE TABLE <name> ( <field> <domain>, … )

```sql
create table department
    (dept_name varchar(20),
     building varchar(15),
     budget numeric(12,2) check (budget > 0),
     primary key (dept_name)
    );
```

```sql
create table instructor
    (ID char(5),
     name varchar(20) not null,
     dept_name varchar(20),
     salary numeric(8,2),
     primary key (ID),
     foreign key (dept_name) references department
    )
```
SQL Constructs: Data Definition Language

- CREATE TABLE <name> ( <field> <domain>, … )

```sql
create table department
(dept_name varchar(20) primary key,
building varchar(15),
budget numeric(12,2) check (budget > 0))
```

```sql
create table instructor (
ID char(5) primary key,
name varchar(20) not null,
department_name varchar(20),
salary numeric(8,2),
foreign key (department_name) references department)
```

- drop table student
- delete from student
  - Keeps the empty table around
- alter table
  - alter table student add address varchar(50);
  - alter table student drop tot_cred;
SQL Constructs: Insert/Delete/Update Tuples

- **INSERT INTO** `<name>` `<field names>`) VALUES `<field values>`
  - `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`
  - `insert into instructor (name, ID) values ('Smith', '10211');`
    - -- NULL for other two
  - `insert into instructor (ID) values ('10211');`
    - -- FAIL

- **DELETE FROM** `<name>` WHERE `<condition>`
  - `delete from department where budget < 80000;`
    - Syntax is fine, but this command **may be rejected** because of referential integrity constraints.

- **DELETE FROM** `<name>` WHERE `<condition>`
  - `delete from department where budget < 80000;`

**Figure 2.5** The `department` relation.

We can choose what happens:
1. Reject the delete, or
2. Delete the rows in Instructor (may be a cascade), or
3. Set the appropriate values in Instructor to NULL
DELETE FROM <name> WHERE <condition>

```sql
create table instructor
    (ID varchar(5),
     name varchar(20) not null,
     dept_name varchar(20),
     salary numeric(8,2) check (salary > 29000),
     primary key (ID),
     foreign key (dept_name) references department
     on delete set null
    );
```

We can choose what happens:
(1) Reject the delete (nothing), or
(2) Delete the rows in Instructor (on delete cascade), or
(3) Set the appropriate values in Instructor to NULL (on delete set null)

DELETE FROM <name> WHERE <condition>

◦ Delete all classrooms with capacity below average
  ```sql
  delete from classroom where capacity <
  (select avg(capacity) from classroom);
  ```

◦ Problem: as we delete tuples, the average capacity changes

◦ Solution used in SQL:
  • First, compute avg capacity and find all tuples to delete
  • Next, delete all tuples found above (without recomputing avg or retesting the tuples)

◦ E.g. consider the query: delete the smallest classroom
SQL Constructs: Insert/Delete/Update Tuples

- UPDATE <name> SET <field name> = <value> WHERE <condition>
  - Increase salaries over $100,000 by 6%, others 5%.
  - What is wrong with:

    update instructor set salary = salary * 1.05 where salary ≤ 10000;
    update instructor set salary = salary * 1.06 where salary > 100000;

  - Order can be important

  - Can be done better using the case statement

```sql
update instructor
set salary =
  case
    when salary > 100000
      then salary * 1.06
    when salary <= 100000
      then salary * 1.05
  end;
```
Outline

- Overview of modeling
- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Basic Data Definition (3.2)
  - Setting up the PostgreSQL database
  - Basic Queries (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)

Setting up the PostgreSQL database

- Done. Bam!
Outline

- Overview of modeling
- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Basic Data Definition (3.2)
  - Setting up the PostgreSQL database
  - Basic Queries (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)

Basic Query Structure

\[
\text{select } A_1, A_2, \ldots, A_n \quad \text{Attributes or expressions} \\
\text{from } r_1, r_2, \ldots, r_m \quad \text{Relations (or queries returning tables)} \\
\text{where } P \quad \text{Predicates}
\]

- Find the names of all instructors:
  \[
  \text{select name} \\
  \text{from instructor}
  \]

- Remove duplicate rows:
  \[
  \text{select distinct name} \\
  \text{from instructor}
  \]

- Order the output:
  \[
  \text{select distinct name} \\
  \text{from instructor} \\
  \text{order by name asc}
  \]

- Apply some filters (predicates):
  \[
  \text{select name} \\
  \text{from instructor} \\
  \text{where salary > 80000 and dept\_name = 'Finance';}
  \]
Basic Query Constructs

Find the names of all instructors:
```
select name
from instructor
```

Find the names of all instructors:
```
select name, salary < 100000
from instructor
```

Find the names of all instructors:
```
select name, salary * 2 as double_salary
from instructor i
where i.salary < 80000 and i.name like '%g_';
```

Filters:
```
select name
from instructor
where (dept_name !='Finance' and salary > 75000)
or (dept_name = 'Finance' and salary > 85000);
```

Expressions in the select clause:
```
select name, salary < 100000
from instructor
```

A filter with a subquery:
```
select name
from instructor
where dept_name in (select dept_name from department
where budget < 100000);
```

Careful with NULLs:
```
select name
from instructor
where salary < 100000 or salary >= 100000;
```

Wouldn't return the instructor with NULL salary (if any)
Multi-table Queries

Use predicates to only select “matching” pairs:
\[
\begin{align*}
\text{select } & \ast \\
\text{from } & \text{instructor } i, \text{ department } d \\
\text{where } & i.\text{dept}_\text{name} = d.\text{dept}_\text{name};
\end{align*}
\]

Cartesian product:
\[
\begin{align*}
\text{select } & \ast \\
\text{from } & \text{instructor, department}
\end{align*}
\]

Similar (in this instance) to using a natural join:
\[
\begin{align*}
\text{select } & \ast \\
\text{from } & \text{instructor natural join department}
\end{align*}
\]

Natural join does an equality on common attributes – doesn’t work here:
\[
\begin{align*}
\text{select } & \ast \\
\text{from } & \text{instructor natural join advisor}
\end{align*}
\]

Instead can use “on” construct (or where clause as above):
\[
\begin{align*}
\text{select } & \ast \\
\text{from } & \text{instructor join advisor on } (i.\text{id} = id)
\end{align*}
\]

3-Table Query to get a list of instructor-teaches-course information:
\[
\begin{align*}
\text{select } & i.\text{name as instructor}_\text{name}, c.\text{title as course}_\text{name} \\
\text{from } & \text{instructor } i, \text{ course } c, \text{ teaches} \\
\text{where } & i.\text{ID} = \text{teaches.ID and c.course}_\text{id} = \text{teaches.course}_\text{id};
\end{align*}
\]

Beware of unintended common names (happens often)
You may think the following query has the same result as above – it doesn’t
\[
\begin{align*}
\text{select } & \text{name, title} \\
\text{from } & \text{instructor natural join course natural join teaches;}
\end{align*}
\]

I prefer avoiding “natural joins” for that reason

Note: On the small dataset, the above two have the same answer, but not on the large dataset. Large dataset has cases where an instructor teaches a course from a different department.
Find courses that ran in Fall 2009 or Spring 2010
\[
\begin{align*}
\text{(select course_id from section where semester = 'Fall' and year = 2009)} \\
\text{union} \\
\text{(select course_id from section where semester = 'Spring' and year = 2010)};
\end{align*}
\]

In both:
\[
\begin{align*}
\text{(select course_id from section where semester = 'Fall' and year = 2009)} \\
\text{intersect} \\
\text{(select course_id from section where semester = 'Spring' and year = 2010)};
\end{align*}
\]

In Fall 2009, but not in Spring 2010:
\[
\begin{align*}
\text{(select course_id from section where semester = 'Fall' and year = 2009)} \\
\text{except} \\
\text{(select course_id from section where semester = 'Spring' and year = 2010)};
\end{align*}
\]

Union/Intersection/Except eliminate duplicates in the answer (the other SQL commands don't) (e.g., try 'select dept_name from instructor').

Can use “union all” to retain duplicates.

NOTE: The duplicates are retained in a systematic fashion (for all SQL operations)

Suppose a tuple occurs \( m \) times in \( r \) and \( n \) times in \( s \), then, it occurs:

- \( m + n \) times in \( r \text{ union all } s \)
- \( \min(m, n) \) times in \( r \text{ intersect all } s \)
- \( \max(0, m - n) \) times in \( r \text{ except all } s \)