CMSC424: Database Design
Relational Model; SQL

Instructor: Amol Deshpande
amol@cs.umd.edu
Outline

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

- **SQL (Chapter 3)**
  - Setting up the PostgreSQL database
  - Data Definition (3.2)
  - Basics (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)
Context

- Data Models
  - Conceptual representation of the data

- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions

- Data Storage
  - How/where to store data, how to access it

- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies
Relational Data Model

Introduced by Ted Codd (late 60’s – early 70’s)

- Before = “Network Data Model” (Cobol as DDL, DML)
- Very contentious: Database Wars (Charlie Bachman vs. Ted Codd)

Relational data model contributes:

1. Separation of logical, physical data models (data independence)
2. Declarative query languages
3. Formal semantics
4. Query optimization (key to commercial success)

1st prototypes:

- Ingres → CA
- Postgres → Illustra → Informix → IBM
- System R → Oracle, DB2
Key Abstraction: Relation

Account =

<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>500</td>
</tr>
</tbody>
</table>

Terms:

- Tables (aka: Relations)

Why called Relations?

Closely correspond to mathematical concept of a relation
Account = 

<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>500</td>
</tr>
</tbody>
</table>

Considered equivalent to…

\{  (Downtown, A-101, 500),  
    (Brighton,   A-201, 900),  
    (Brighton,   A-217, 500) \}  

Relational database semantics defined in terms of mathematical relations
Relations

Account =

<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>500</td>
</tr>
</tbody>
</table>

Considered equivalent to...

\{(Downtown, A-101, 500),
   (Brighton, A-201, 900),
   (Brighton, A-217, 500)\}

Terms:

- Tables (aka: Relations)
- Rows (aka: tuples)
- Columns (aka: attributes)
- Schema (e.g.: Acct_Schema = (bname, acct_no, balance))
Definitions

**Relation Schema (or Schema)**
A list of attributes and their domains
E.g. `account` (account-number, branch-name, balance)

Programming language equivalent: A variable (e.g. x)

**Relation Instance**
A particular instantiation of a relation with actual values
Will change with time

<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>500</td>
</tr>
</tbody>
</table>

Programming language equivalent: Value of a variable
Definitions

**Domains of an attribute/column**
The set of permitted values

* e.g., `bname` must be String, `balance` must be a positive real number

We typically assume domains are **atomic**, i.e., the values are treated as indivisible (specifically: you can’t store lists or arrays in them)

**Null value**

A special value used if the value of an attribute for a row is:

* unknown (e.g., don’t know address of a customer)
* inapplicable (e.g., “spouse name” attribute for a customer)
* withheld/hidden

Different interpretations all captured by a single concept – leads to major headaches and problems
Tables in a University Database

classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(Id, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)
Outline

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

- Let $K \subseteq R$
- $K$ is a **superkey** of $R$ if values for $K$ are sufficient to identify a unique tuple of any possible relation $r(R)$
  - *Example:* $\{ID\}$ and $\{ID, name\}$ are both superkeys of instructor.
- Superkey $K$ is a **candidate key** if $K$ is **minimal** (i.e., no subset of it is a superkey)
  - *Example:* $\{ID\}$ is a candidate key for Instructor
- One of the candidate keys is selected to be the **primary key**
  - Typically one that is small and immutable (doesn’t change often)
- Primary key typically highlighted (e.g., underlined)
Tables in a University Database

classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
Tables in a University Database

takes(ID, course_id, sec_id, semester, year, grade)

What about ID, course_id?
   No. May repeat:
   ("1011049", "CMSC424", "102", "Fall", 2015, null)

What about ID, course_id, sec_id?
   May repeat:
   ("1011049", "CMSC424", "101", "Fall", 2015, null)

What about ID, course_id, sec_id, semester?
             ("1011049", "CMSC424", "101", "Spring", 2015, null)
Tables in a University Database

classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building,
        room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)
Foreign key: Primary key of a relation that appears in another relation
- \{ID\} from student appears in takes, advisor
- student called referenced relation
- takes is the referencing relation
- Typically shown by an arrow from referencing to referenced

Foreign key constraint: the tuple corresponding to that primary key must exist
- Imagine:
  - Tuple: (‘student101’, ‘CMSC424’) in takes
  - But no tuple corresponding to ‘student101’ in student
- Also called referential integrity constraint
Schema Diagram for the Banking Enterprise

branch
- branch-name
- branch-city
- assets

account
- account-number
- branch-name
- balance

depositor
- customer-name
- account-number

customer
- customer-name
- customer-city
- customer-street

loan
- loan-number
- branch-name
- amount

borrower
- customer-name
- loan-number
Examples

- Married(person1_ssn, person2_ssn, date_married, date_divorced)
- Account(cust_ssn, account_number, cust_name, balance, cust_address)
- RA(student_id, project_id, supervisor_id, appt_time, appt_start_date, appt_end_date)
- Person(Name, DOB, Born, Education, Religion, ...)
  - Information typically found on Wikipedia Pages
Examples

- Married(person1_ssn, person2_ssn, date_married, date_divorced)

- Account(cust_ssn, account_number, cust_name, balance, cust_address)
  - If a single account per customer, then: cust_ssn
  - Else: (cust_ssn, account_number)
    - In the latter case, this is not a good schema because it requires repeating information

- RA(student_id, project_id, supervisor_id, appt_time, appt_start_date, appt_end_date)
  - Could be smaller if there are some restrictions – requires some domain knowledge of the data being stored

- Person(Name, DOB, Born, Education, Religion, ...)
  - Information typically found on Wikipedia Pages
  - Unclear what could be a primary key here: you could in theory have two people who match on all of those
Outline

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

- Example schema: \( R(A, B) \)
- Practical languages
  - SQL
    - select A from R where B = 5;
  - Datalog (sort of practical)
    - \( q(A) :- R(A, 5) \)
- Formal languages
  - Relational algebra
    - \( \pi_A ( \sigma_{B=5} (R) ) \)
  - Tuple relational calculus
    - \( \{ t : \{A\} | \exists s : \{A, B\} ( R(A, B) \land s.B = 5 ) \} \)
  - Domain relational calculus
    - Similar to tuple relational calculus
Relational Operations

- Some of the languages are “procedural” and provide a set of operations
  - Each operation takes one or two relations as input, and produces a single relation as output
  - Examples: SQL, and Relational Algebra

- The “non-procedural” (also called “declarative”) languages specify the output, but don’t specify the operations
  - Relational calculus
  - Datalog (used as an intermediate layer in quite a few systems today)
Select Operation

Choose a subset of the tuples that satisfies some predicate
Denoted by $\sigma$ in relational algebra

Relation $r$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\beta$</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\beta$</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\beta$</td>
<td>23</td>
<td>10</td>
</tr>
</tbody>
</table>

$\sigma_{A=B \land D > 5 \ (r)}$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\beta$</td>
<td>23</td>
<td>10</td>
</tr>
</tbody>
</table>
Choose a subset of the columns (for all rows)
Denoted by $\prod$ in relational algebra

Relation $r$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\beta$</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\beta$</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\beta$</td>
<td>23</td>
<td>10</td>
</tr>
</tbody>
</table>

$\prod_{A,D} (r)$

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>7</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>7</td>
</tr>
<tr>
<td>$\beta$</td>
<td>3</td>
</tr>
<tr>
<td>$\beta$</td>
<td>10</td>
</tr>
</tbody>
</table>

Relational algebra following “set” semantics – so no duplicates
SQL allows for duplicates – we will cover the formal semantics later
Set Union, Difference

Relation $r$, $s$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$\alpha$</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

$r$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

$s$

$\alpha$

$r \cup s$:   |   |   |
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$\alpha$</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

$r - s$:   |   |   |
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Must be compatible schemas

What about intersection?

Can be derived

$r \cap s = r - (r - s)$;
Cartesian Product

Combine tuples from two relations

If one relation contains N tuples and the other contains M tuples, the result would contain N*M tuples

The result is rarely useful – almost always you want pairs of tuples that satisfy some condition

Relation r, s

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
</tr>
<tr>
<td>β</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>β</td>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>γ</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>

r × s:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td>α</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>β</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>β</td>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>γ</td>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>α</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>β</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>β</td>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>γ</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>
**Joins**

Combine tuples from two relations if the pair of tuples satisfies some constraint

Equivalent to Cartesian Product followed by a Select

Relation \( r, s \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>1</td>
<td>( \alpha )</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>( \beta )</td>
<td>2</td>
<td>( \beta )</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>20</td>
<td>( \gamma )</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>1</td>
<td>( \alpha )</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>( \beta )</td>
<td>2</td>
<td>( \beta )</td>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>2</td>
<td>( \gamma )</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>

\[ r \bowtie_{A = C} s \]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>1</td>
<td>( \beta )</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>1</td>
<td>( \beta )</td>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>1</td>
<td>( \gamma )</td>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>( \beta )</td>
<td>2</td>
<td>( \alpha )</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>( \beta )</td>
<td>2</td>
<td>( \beta )</td>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>( \beta )</td>
<td>2</td>
<td>( \gamma )</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>
Natural Join

Combine tuples from two relations if the pair of tuples agree on the common columns (with the same name)

![Table](image)

**Figure 2.5** The department relation.

![Table](image)

**Figure 2.4** Unsorted display of the instructor relation.

![Table](image)

**Figure 2.12** Result of natural join of the instructor and department relations.
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)
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 relation.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.
CREATE TABLE <name> ( <field> <domain>, ... )

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

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
  )
CREATE TABLE <name> ( <field> <domain>, ... )

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

create table instructor ( 

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

- 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.
### SQL Constructs: Insert/Delete/Update Tuples

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

#### Instructor relation

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>dept_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>65000</td>
<td>Comp. Sci.</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>90000</td>
<td>Finance</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>40000</td>
<td>Music</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>95000</td>
<td>Physics</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>60000</td>
<td>History</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>87000</td>
<td>Physics</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>75000</td>
<td>Comp. Sci.</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>62000</td>
<td>History</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>80000</td>
<td>Finance</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>72000</td>
<td>Biology</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>92000</td>
<td>Comp. Sci.</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>80000</td>
<td>Elec. Eng.</td>
</tr>
</tbody>
</table>

#### 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
SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>

  ```sql
  delete from department where budget < 80000;
  ```

- create table instructor
  ```sql
  (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
    
    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
UPDATE <name> SET <field name> = <value> WHERE <condition>

- Increase all salaries’s over $100,000 by 6%, all other receive 5%.
- Write two update statements:
  
  update instructor
  set salary = salary * 1.06
  where salary > 100000;

  update instructor
  set salary = salary * 1.05
  where salary ≤ 10000;

- The order is important
- Can be done better using the case statement
UPDATE <name> SET <field name> = <value> WHERE <condition>

- Increase all salaries’s over $100,000 by 6%, all other receive 5%.
- Can be done better using the case statement

```
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

- Follow the instructions posted on the course website to set up the University database in PostgreSQL

https://github.com/umddb/cmsc424-fall2015/tree/master/postgresql-setup
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

\[ \text{select } A_1, A_2, \ldots, A_n \]
\[ \text{from } r_1, r_2, \ldots, r_m \]
\[ \text{where } P \]

- Attributes or expressions
- Relations (or queries returning tables)
- Predicates

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

Apply some filters (predicates):
\[ \text{select name} \]
\[ \text{from instructor} \]
\[ \text{where salary} > 80000 \text{ and dept_name} = 'Finance'; \]

Remove duplicates:
\[ \text{select distinct name} \]
\[ \text{from instructor} \]

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

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

Select all attributes:
\[
\text{select *} \\
\text{from instructor}
\]

Expressions in the select clause:
\[
\text{select name, salary < 100000} \\
\text{from instructor}
\]

More complex filters:
\[
\text{select name} \\
\text{from instructor} \\
\text{where (dept\_name !='Finance' and salary > 75000)} \\
\text{or (dept\_name = 'Finance' and salary > 85000)};
\]

A filter with a subquery:
\[
\text{select name} \\
\text{from instructor} \\
\text{where dept\_name in (select dept\_name from department where budget < 100000)};
\]
Find the names of all instructors:

```sql
select name
from instructor
```

Renaming tables or output column names:

```sql
select i.name, i.salary * 2 as double_salary
from instructor i
where i.salary < 80000 and i.name like '%g_';
```

More complex expressions:

```sql
select concat(name, concat(' ', 'dept_name'))
from instructor;
```

Careful with NULLs:

```sql
select name
from instructor
where salary < 100000 or salary >= 100000;
```

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

Cartesian product:

```
select * 
from instructor, department
```

Use predicates to only select “matching” pairs:

```
select *
from instructor i, department d
where i.dept_name = d.dept_name;
```

Identical (in this case) to using a natural join:

```
select *
from instructor natural join department;
```

Natural join does an equality on common attributes – doesn’t work here:

```
select *
from instructor natural join advisor;
```

Instead can use “on” construct (or where clause as above):

```
select *
from instructor join advisor on (i_id = id);
```
3-Table Query to get a list of instructor-teaches-course information:

```sql
select i.name as instructor_name, c.title as course_name
from instructor i, course c, teaches
where i.ID = teaches.ID and c.course_id = teaches.course_id;
```

Beware of unintended common names (happens often)
You may think the following query has the same result as above – it doesn’t

```sql
select name, title
from instructor natural join course natural join teaches;
```

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.
Set operations

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:

\[
\begin{align*}
&\text{(select course_id from section where semester = 'Fall' and year = 2009)} \\
&\text{except} \\
&\text{(select course_id from section where semester = 'Spring' and year = 2010)};
\end{align*}
\]
Set operations: Duplicates

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 $r$ union all $s$
- min$(m,n)$ times in $r$ intersect all $s$
- max$(0, m - n)$ times in $r$ except all $s$
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 $r$ union all $s$
- $\min(m,n)$ times in $r$ intersect all $s$
- $\max(0, m - n)$ times in $r$ except all $s$
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)
SQL: Nulls

The “dirty little secret” of SQL
(major headache for query optimization)

Can be a value of any attribute

e.g: branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

What does this mean?

(unknown) We don’t know Waltham’s assets?
(inapplicable) Waltham has a special kind of account without assets
(withheld) We are not allowed to know
Arithmetic Operations with **Null**

\[ n + \text{NULL} = \text{NULL} \]

(similarly for all *arithmetic ops*: +, −, *, /, mod, …)

```
e.g: branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

```
SELECT bname, assets * 2 as a2 =
FROM branch

<table>
<thead>
<tr>
<th>bname</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>18M</td>
</tr>
<tr>
<td>Perry</td>
<td>3.4M</td>
</tr>
<tr>
<td>Mianus</td>
<td>.8M</td>
</tr>
<tr>
<td>Waltham</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```
### SQL: Nulls

**Boolean Operations with Null**

\[ n < \text{NULL} = \text{UNKNOWN} \quad (\text{similarly for all boolean ops: } >, \leq, \geq, \neq, =, \ldots) \]

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

```sql
SELECT * =
FROM branch
WHERE assets = NULL
```

Counter-intuitive: \( \text{NULL} * 0 = \text{NULL} \)

Counter-intuitive: select * from movies
where length >= 120 or length <= 120
**SQL: Nulls**

**Boolean Operations with Null**

\[ n < \text{NULL} = \text{UNKNOWN} \]  
(similarly for all boolean ops: \(>, \leq, \geq, \not=, =, \ldots\))

\[ \text{e.g.: } \text{branch} = \]

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**SELECT** *
**FROM** branch
**WHERE** assets IS NULL
Boolean Operations with Unknown

n < NULL = UNKNOWN  (similarly for all boolean ops: >, <=, >=, <>, =, ...)

FALSE OR UNKNOWN = UNKNOWN
TRUE AND UNKNOWN = UNKNOWN

Intuition: substitute each of TRUE, FALSE for unknown. If different answer results, results is unknown

UNKNOWN OR UNKNOWN = UNKNOWN
UNKNOWN AND UNKNOWN = UNKNOWN
NOT (UNKNOWN) = UNKNOWN

Can write:

SELECT ...  
FROM ...  
WHERE booleanexp IS UNKNOWN

UNKNOWN tuples are not included in final result
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)
Other common aggregates: \( \text{max, min, sum, count, stdev, ...} \)

Find the average salary of instructors in the Computer Science
\[
\text{select } \text{avg}(\text{salary}) \\
\text{from instructor} \\
\text{where dept\_name} = \text{‘Comp. Sci’};
\]

Find the average salary of instructors in the Computer Science
\[
\text{select count (distinct ID)} \\
\text{from teaches} \\
\text{where semester} = \text{‘Spring’ and year} = 2010
\]

Find the average salary of instructors in the Computer Science
\[
\text{Can specify aggregates in any query.} \\
\text{Find max salary over instructors teaching in S’10} \\
\text{select max(salary)} \\
\text{from teaches natural join instructor} \\
\text{where semester} = \text{‘Spring’ and year} = 2010;
\]

Find instructors with max salary:
\[
\text{select *} \\
\text{from instructor} \\
\text{where salary} = (\text{select max(salary) from instructor});
\]
Aggregate result can be used as a scalar. Find instructors with max salary:

```sql
select * 
from instructor 
where salary = (select max(salary) from instructor);
```

Following doesn’t work:

```sql
select * 
from instructor 
where salary = max(salary);
```

```sql
select name, max(salary) 
from instructor 
where salary = max(salary);
```
Split the tuples into groups, and computer the aggregate for each group

```sql
select dept_name, avg(salary)
from instructor
group by dept_name;
```
Attributes in the select clause must be aggregates, or must appear in the group by clause. Following wouldn’t work

```sql
select dept_name, ID, avg(salary)
from instructor
group by dept_name;
```

“having” can be used to select only some of the groups.

```sql
select dept_name, ID, avg(salary)
from instructor
group by dept_name
having avg(salary) > 42000;
```
Given

\[ \text{branch} = \]

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Aggregate Operations

\[
\text{SELECT SUM (assets) } = \\
\text{FROM branch}
\]

<table>
<thead>
<tr>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 M</td>
</tr>
</tbody>
</table>

**NULL is ignored for SUM**

*Same for AVG (3.7M), MIN (0.4M), MAX (9M)*

Also for COUNT(assets) -- returns 3

**But COUNT (*) returns**

<table>
<thead>
<tr>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Given

branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
</table>

SELECT SUM (assets) = SUM
FROM branch

• Same as AVG, MIN, MAX
• But COUNT (assets) returns COUNT 0
Summary

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

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