CMSC424: Relational Algebra; JDBC; Remaining SQL

Instructor: Amol Deshpande
amol@cs.umd.edu
Today’s Class

- **Advanced SQL**
  - Accessing SQL From a Programming Language
    - Dynamic SQL: JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Integrity Constraints
  - Advanced Aggregation Features

- **Relational Algebra**
  - Formal Semantics of SQL (i.e., how to deal with duplicates)

- **Other things**
  - Exam Wednesday -- everything covered so far, including today
  - Project 3: JDBC; Some advanced SQL; Query Plans
    - Will post a iPython notebook on the last one in a couple of days
Client-server Architectures

Many different possibilities to build an end-to-end application, but often see 2-tier or 3-tier architectures

(a) Two-tier architecture

(b) Three-tier architecture

Figure 1.6 Two-tier and three-tier architectures.
Three-tier Architecture

**Presentation tier**
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

**Logic tier**
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

**Data tier**
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

---

e.g., Web servers

e.g., Ruby on Rails, Java EE, ASP.NET, PHP, ColdFusion, Perl or Python frameworks

e.g., PostgreSQL, Oracle, MySQL, etc…
Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
    - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Advanced Aggregation Features
  - Integrity Constraints
  - Recursion

- Relational Algebra

- Formal Semantics of SQL (i.e., how to deal with duplicates)
JDBC and ODBC

- API (application-program interface) for a program to interact with a database server
  - Application makes calls to
    - Connect with the database server
    - Send SQL commands to the database server
    - Fetch tuples of result one-by-one into program variables

- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
  - Other API’s such as ADO.NET sit on top of ODBC

- JDBC (Java Database Connectivity) works with Java
public static void JDBCexample(String dbid, String userid, String passwd) {
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    } catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
Update to database
try {
    stmt.executeUpdate("insert into instructor values(’77987’, ’Kim’, ’Physics’, 98000)");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple. " + sqle);
}

Execute query and fetch and print results
ResultSet rset = stmt.executeQuery("select dept_name, avg(salary) from instructor group by dept_name");
while (rset.next()) {
    System.out.println(rset.getString("dept_name") + " " + rset.getFloat(2));
}
Getting result fields:
- `rs.getString("dept_name")` and `rs.getString(1)` equivalent if `dept_name` is the first argument of select result.

Dealing with Null values:
- `int a = rs.getInt("a");`
  - `if (rs.wasNull()) System.out.println("Got null value");`
**PreparedStatement**

- `PreparedStatement pStmt = conn.prepareStatement("insert into instructor values(?,?,?,?)");`
  
  `pStmt.setString(1, "88877");`  
  `pStmt.setString(2, "Perry");`  
  `pStmt.setString(3, "Finance");`  
  `pStmt.setInt(4, 125000);`  
  `pStmt.executeUpdate();`  
  `pStmt.setString(1, "88878");`  
  `pStmt.executeUpdate();`

- For queries, use `pStmt.executeQuery()`, which returns a `ResultSet`
- **WARNING**: always use prepared statements when taking an input from the user and adding it to a query
  - **NEVER** create a query by concatenating strings which you get as inputs
String query = "select * from instructor where name = ‘" + name + "’" 

User enters: X’ or ’ Y’ = ’ Y

We execute:
  ◦ "select * from instructor where name = ’" + "X’ or ’ Y’ = ’ Y" + "’""
  ◦ which is: select * from instructor where name = ’ X’ or ’ Y’ = ’ Y’

Worse: user enters:
  • X’ ; update instructor set salary = salary + 10000; --

Prepared statement internally uses:
"select * from instructor where name = ’ X\’ or \’ Y\’ = \’ Y’"

Always use prepared statements, with user inputs as parameters
SQL Injection: XKCD

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR - DID HE BREAK SOMETHING?

IN A WAY-

DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
ResultSet metadata

E.g., after executing query to get a ResultSet rs:

- ResultSetMetaData rsmd = rs.getMetaData();
  for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
  }

- Look up the manual etc. for much more
The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.

A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise **embedded** SQL.

The basic form of these languages follows that of the System R embedding of SQL into PL/I.

**EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

```
EXEC SQL <embedded SQL statement > END_EXEC
```

Note: this varies by language (for example, the Java embedding uses

```
# SQL { .... }; )
```

## Embedded SQL
Outline

- **Advanced SQL**
  - Accessing SQL From a Programming Language
    - Dynamic SQL
    - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints

- **Relational Algebra**

- **Formal Semantics of SQL** (i.e., how to deal with duplicates)
Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.

- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details

- Object-oriented aspects of these features are covered in Chapter 22 (Object Based Databases)
Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```sql
create function dept_count (dept_name varchar(20))
returns integer
begin
declare d_count integer;
select count (*) into d_count
from instructor
where instructor.dept_name = dept_name
return d_count;
end
```

Find the department name and budget of all departments with more that 12 instructors.

```sql
select dept_name, budget
from department
where dept_count (dept_name) > 1
```
SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

  ```sql
  create function dept_count (dept_name varchar(20))
  returns integer
  begin
    declare d_count integer;
    select count (*) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
  end
  ```

- Syntax doesn’t seem to work with PostgreSQL; see here for examples: http://www.postgresql.org/docs/9.1/static/sql-createfunction.html
Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```sql
create function instructors_of (dept_name char(20)
returns table (ID varchar(5),
               name varchar(20),
               dept_name varchar(20),
               salary numeric(8,2))

return table
(select ID, name, dept_name, salary
 from instructor
where instructor.dept_name = instructors_of.dept_name)
```

- Usage

```sql
select *
from table (instructors_of (‘Music’))
```
Procedural Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
  - Example:

    ```sql
    declare n integer default 0;
    for r as
        select budget from department
        where dept_name = 'Music'
    do
        set n = n - r.budget
    end for
    ```
Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
    - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints

- Relational Algebra

- Formal Semantics of SQL (i.e., how to deal with duplicates)
Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (  
    select course_id, prereq_id  
    from prereq

    union

    select rec_prereq.course_id, prereq.prereq_id  
    from rec_prereq, prereq  
    where rec_prereq.prereq_id = prereq.course_id
  )

select *  
from rec_prereq;
```

This example view, `rec_prereq`, is called the transitive closure of the `prereq` relation.
Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.

- Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of \textit{prereq} with itself.

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>BIO-399</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-319</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
<tr>
<td>EE-181</td>
<td>PHY-101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>Tuples in cl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(CS-301)</td>
</tr>
<tr>
<td>1</td>
<td>(CS-301), (CS-201)</td>
</tr>
<tr>
<td>2</td>
<td>(CS-301), (CS-201)</td>
</tr>
<tr>
<td>3</td>
<td>(CS-301), (CS-201), (CS-101)</td>
</tr>
<tr>
<td>4</td>
<td>(CS-301), (CS-201), (CS-101)</td>
</tr>
<tr>
<td>5</td>
<td>(CS-301), (CS-201), (CS-101)</td>
</tr>
</tbody>
</table>
Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
    - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints
- Relational Algebra
- Formal Semantics of SQL (i.e., how to deal with duplicates)
Ranking

- Rank instructors by salary.
  
  \[
  \text{select } *, \text{ rank()} \text{ over (order by salary desc) as s_rank from instructor;}
  \]

- An extra \textbf{order by} clause is needed to get them in sorted order
- Ranking may leave gaps (two with rank 5, none with rank 6)
- \textbf{Use dense_rank} to leave no gaps
- Can be done without using new keywords, but probably inefficient

  \[
  \text{select ID, (1 + (select count(*) from instructors i2 where i2.salary > i1.salary)) as s_rank from instructor i1 order by s_rank;}
  \]
Ranking can be done within partition of the data.

“Find the rank of instructors within each department.”

```sql
select ID, dept_name,
  rank () over (partition by dept_name order by salary desc)
  as dept_rank
from instructor
order by dept_name, dept_rank;
```

Other ranking functions:

- `percent_rank` (within partition, if partitioning is done)
- `cume_dist` (cumulative distribution)
  - fraction of tuples with preceding values
- `row_number` (non-deterministic in presence of duplicates)
Windowing

- Used to smooth out random variations.
- E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”

**Window specification** in SQL:
- Given relation `sales(date, value)`
  ```sql
  select date, sum(value) over (order by date between rows 1 preceding and 1 following)
  from sales
  ````
- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between 10 preceding and current row**
    - All rows with values between current row value −10 to current value
  - **range interval 10 day preceding**
    - Not including current row
Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
    - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints

- Relational Algebra

- Formal Semantics of SQL (i.e., how to deal with duplicates)
IC’s

- Predicates on the database
  - Must always be true (checked whenever db gets updated)

There are the following 4 types of IC’s:

- **Key constraints** (1 table)
  - e.g., *2 accts can’t share the same acct_no*

- **Attribute constraints** (1 table)
  - e.g., *accts must have nonnegative balance*

- **Referential Integrity constraints** (2 tables)
  - E.g. *bnames associated w/ loans must be names of real branches*

- **Global Constraints** (*n* tables)
  - E.g., all *loans* must be carried by at least 1 *customer* with a savings acct
Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. **Primary Key:**
   
   ```sql
   CREATE TABLE branch(
     bname CHAR(15) PRIMARY KEY,
     bcity CHAR(20),
     assets INT);
   
   or
   
   CREATE TABLE depositor(
     cname CHAR(15),
     acct_no CHAR(5),
     PRIMARY KEY(cname, acct_no));
   ```

2. **Candidate Keys:**
   
   ```sql
   CREATE TABLE customer (  
     ssn CHAR(9) PRIMARY KEY,  
     cname CHAR(15),
     address CHAR(30),
     city CHAR(10),
     UNIQUE (cname, address, city));
   ```
Key Constraints

Effect of SQL Key declarations

PRIMAR Y (A1, A2, .., An) or
UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any inserted tuple. If found, reject insertion

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)

1. 1 primary key per table, several unique keys allowed.
2. Only primary key can be referenced by “foreign key” (ref integrity)
3. DBMS may treat primary key differently (e.g.: create an index on PK)

How would you implement something like this?
Attribute Constraints

- **Idea:**
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)

- **In SQL:**
  1. **NOT NULL**
     - e.g.: `CREATE TABLE branch(
               bname CHAR(15) NOT NULL,
               ....
             )`
     - Note: declaring bname as primary key also prevents null values
  2. **CHECK**
     - e.g.: `CREATE TABLE depositor(
               ....
               balance int NOT NULL,
               CHECK( balance >= 0),
               ....
             )`
     - affect insertions, update in affected columns
Domains: can associate constraints with DOMAINS rather than attributes

e.g: instead of:

```
CREATE TABLE depositor(
    ....
    balance INT NOT NULL,
    CHECK (balance >= 0)
)
```

One can write:

```
CREATE DOMAIN bank-balance INT (
    CONSTRAINT not-overdrawn CHECK (value >= 0),
    CONSTRAINT not-null-value CHECK( value NOT NULL));
```

```
CREATE TABLE depositor (  
    ....
    balance  bank-balance,
)
```

Advantages?
### Attribute Constraints

Advantage of associating constraints with domains:

1. can avoid repeating specification of same constraint for multiple columns

2. can name constraints
e.g.: CREATE DOMAIN bank-balance INT (CONSTRAINT not-overdrawn CHECK (value >= 0), CONSTRAINT not-null-value CHECK(value NOT NULL));

allows one to:
1. add or remove:
   ALTER DOMAIN bank-balance
   ADD CONSTRAINT capped CHECK(value <= 10000)
2. report better errors (know which constraint violated)
Idea: prevent “dangling tuples” (e.g.: a loan with a bname, *Kenmore*, when no *Kenmore* tuple in branch)

Ref Integrity:
ensure that:
foreign key value \(\rightarrow\) primary key value

(note: don’t need to ensure \(\leftarrow\), i.e., not all branches have to have loans)
Referential Integrity Constraints

In SQL:

```
CREATE TABLE branch(
    bname CHAR(15) PRIMARY KEY,
    ....)

CREATE TABLE loan (  
    .......
    FOREIGN KEY bname REFERENCES branch);
```

Affects:
1) Insertions, updates of referencing relation
2) Deletions, updates of referenced relation
what happens when we try to delete this tuple?

Ans: 3 possibilities

1) reject deletion/ update

2) set \( t_i[c], t_j[c] = NULL \)

3) propagate deletion/update
   
   DELETE: delete \( t_i, t_j \)
   
   UPDATE: set \( t_i[c], t_j[c] \) to updated values
what happens when we try to delete this tuple?

CREATE TABLE A (   .....  
       FOREIGN KEY c REFERENCES B action  
............ )

Action:  1) left blank (deletion/update rejected)

2) ON DELETE SET NULL/ ON UPDATE SET NULL
   sets  ti[c] = NULL, tj[c] = NULL

3) ON DELETE CASCADE
   deletes ti, tj
   ON UPDATE CASCADE
   sets ti[c], tj[c] to new key values
Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)
   ◦ E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

2) multiple relations: CREATE ASSERTION

SQL examples:
   1) single relation: All Bkln branches must have assets > 5M

      CREATE TABLE branch ( 
      ........
      bcity CHAR(15),  
      assets INT,  
      CHECK (NOT(bcity = ‘Bkln’) OR assets > 5M))

   Affects: 
   insertions into branch
   updates of bcity or assets in branch
Global Constraints

SQL example:

2) Multiple relations: every loan has a borrower with a savings account

CHECK (NOT EXISTS (  
  SELECT  * 
  FROM loan AS L 
  WHERE NOT EXISTS(  
    SELECT  * 
    FROM borrower B, depositor D, account A 
    WHERE B.cname = D.cname AND  
      D.acct_no = A.acct_no AND  
      L.lno = B.lno)))

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:

CREATE ASSERTION loan-constraint
  CHECK( ..... )

Checked with EVERY DB update! very expensive.....
## Summary: Integrity Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Where declared</th>
<th>Affects...</th>
<th>Expense</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Constraints</strong></td>
<td>CREATE TABLE (PRIMARY KEY, UNIQUE)</td>
<td>Insertions, Updates</td>
<td>Moderate</td>
</tr>
<tr>
<td><strong>Attribute Constraints</strong></td>
<td>CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)</td>
<td>Insertions, Updates</td>
<td>Cheap</td>
</tr>
<tr>
<td><strong>Referential Integrity</strong></td>
<td>Table Tag (FOREIGN KEY .... REFERENCES ....)</td>
<td>1. Insertions into referencing rel’n</td>
<td>1,2: like key constraints. Another reason to index/sort on the primary keys</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Updates of referencing rel’n of relevant attrs</td>
<td>3,4: depends on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Deletions from referenced rel’n</td>
<td>a. update/delete policy chosen</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. Update of referenced rel’n</td>
<td>b. existence of indexes on foreign key</td>
</tr>
<tr>
<td><strong>Global Constraints</strong></td>
<td>Table Tag (CHECK) or outside table (CREATE ASSERTION)</td>
<td>1. For single rel’n constraint, with insertion, deletion of relevant attrs</td>
<td>1. cheap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. For assertions w/ every db modification</td>
<td>2. very expensive</td>
</tr>
</tbody>
</table>
Outline

- Advanced SQL
- Relational Algebra
Relational Algebra

- Procedural language

- Six basic operators
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename

- The operators take one or more relations as inputs and give a new relation as a result.
Relation r

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>α</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>α</td>
<td>β</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>β</td>
<td>β</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>α</td>
<td>α</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>β</td>
<td>β</td>
<td>23</td>
<td>10</td>
</tr>
</tbody>
</table>

**SQL Equivalent:**
```
select *
from r
where A = B and D > 5
```

*Unfortunate naming confusion*
Project

Relation $r$

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\beta$</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\beta$</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\beta$</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

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

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

SQL Equivalent:

```
select distinct A, D
from r
```
## Set Union, Difference

<table>
<thead>
<tr>
<th>Relation r, s</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>r</td>
</tr>
<tr>
<td>α</td>
<td>2</td>
<td>s</td>
</tr>
<tr>
<td>β</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r ∪ s:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r − s:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Must be compatible schemas

What about intersection?

Can be derived

\[ r \cap s = r - (r - s); \]

**SQL Equivalent:**

- select * from r
- union/except/intersect
- select * from s;

This is one case where duplicates are removed.
Cartesian Product

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>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>

SQL Equivalent:

```
select distinct *
from r, s
```

Does not remove duplicates.
Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.

Example:

\[ \rho_x (E) \]

returns the expression \( E \) under the name \( X \)

If a relational-algebra expression \( E \) has arity \( n \), then

\[ \rho_x (A_1, A_2, ..., A_n) (E) \]

returns the result of expression \( E \) under the name \( X \), and with the attributes renamed to \( A_1, A_2, ..., A_n \).
Relational Algebra

- Those are the basic operations

- What about SQL Joins?
  - Compose multiple operators together
    \[ \sigma_{A=C}(r \times s) \]

- Additional Operations
  - Set intersection
  - Natural join
  - Division
  - Assignment
Additional Operators

- **Set intersection (∩)**
  - \( r \cap s = r - (r - s); \)
  - SQL Equivalent: intersect

- **Assignment (←)**
  - A convenient way to write complex RA expressions
  - Essentially for creating “temporary” relations
    - \( temp1 ← \prod_{R-S}(r) \)
  - SQL Equivalent: “create table as...”
Additional Operators: Joins

- **Natural join (⋈)**
  - A Cartesian product with equality condition on common attributes
  - Example:
    - if \( r \) has schema \( R(A, B, C, D) \), and if \( s \) has schema \( S(E, B, D) \)
    - Common attributes: \( B \) and \( D \)
    - Then:
      \[
      r \bowtie s = \prod_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))
      \]

- **SQL Equivalent:**
  - select \( r.A, r.B, r.C, r.D, s.E \) from \( r, s \) where \( r.B = s.B \) and \( r.D = s.D \), OR
  - select * from \( r \) natural join \( s \)
Additional Operators: Joins

- **Equi-join**
  - A join that only has equality conditions

- **Theta-join** \((\bowtie_\theta)\)
  - \(r \bowtie_\theta s = \sigma_\theta(r \times s)\)

- **Left outer join** \((\bowtie)\)
  - Say \(r(A, B), s(B, C)\)
  - We need to somehow find the tuples in \(r\) that have no match in \(s\)
  - Consider: \((r \setminus \pi_{r.A, r.B}(r \bowtie s))\)
    - We are done:
      \[
      (r \bowtie s) \cup \rho_{temp}(A, B, C) \left( (r \setminus \pi_{r.A, r.B}(r \bowtie s)) \times \{(NULL)\} \right)
      \]
# Additional Operators: Join Variations

- **Tables:** \( r(A, B), s(B, C) \)

<table>
<thead>
<tr>
<th>name</th>
<th>Symbol</th>
<th>SQL Equivalent</th>
<th>RA expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross product</td>
<td>×</td>
<td>select * from ( r, s );</td>
<td>( r \times s )</td>
</tr>
<tr>
<td>natural join</td>
<td>⋈</td>
<td>natural join</td>
<td>( \pi_{r.A, r.B, s.C} \sigma_{r.B = s.B}(r \times s) )</td>
</tr>
<tr>
<td>theta join</td>
<td>⋈_θ</td>
<td>from .. where ( \theta );</td>
<td>( \sigma_\theta(r \times s) )</td>
</tr>
<tr>
<td>equi-join</td>
<td>⋈_θ</td>
<td>(theta must be equality)</td>
<td></td>
</tr>
<tr>
<td>left outer join</td>
<td>( r \bowtie s )</td>
<td>left outer join (with “on”)</td>
<td>(see previous slide)</td>
</tr>
<tr>
<td>full outer join</td>
<td>( r \bowtie s )</td>
<td>full outer join (with “on”)</td>
<td>–</td>
</tr>
<tr>
<td>(left) semijoin</td>
<td>( r \bowtie s )</td>
<td>none</td>
<td>( \pi_{r.A, r.B}(r \bowtie s) )</td>
</tr>
<tr>
<td>(left) antijoin</td>
<td>( r \triangleright s )</td>
<td>none</td>
<td>( r - \pi_{r.A, r.B}(r \bowtie s) )</td>
</tr>
</tbody>
</table>
Additional Operators: Division

- Suitable for queries that have “for all”
  - $r \div s$

- Think of it as “opposite of Cartesian product”
  - $r \div s = t \iff t \times s \subseteq r$

<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>$\alpha$</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>$\beta$</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>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>

\[ \begin{array}{c|c} A & B \\ \hline \alpha & 1 \\ \beta & 2 \end{array} \] \[ \begin{array}{c|c|c} C & D & E \\ \hline \alpha & 10 & a \\ \beta & 10 & a \\ \beta & 20 & b \\ \gamma & 10 & b \end{array} \]
Example Query

- Find the largest salary in the university
  - Step 1: find instructor salaries that are less than some other instructor salary (i.e. not maximum)
    - using a copy of instructor under a new name $d$
    - $\prod_{instructor.salary} (\sigma_{instructor.salary < d,salary} (instructor \times \rho_d(instructor)))$
  - Step 2: Find the largest salary
    - $\prod_{salary(instructor)} - \prod_{instructor.salary} (\sigma_{instructor.salary < d,salary} (instructor \times \rho_d(instructor)))$
Example Queries

- Find the names of all instructors in the Physics department, along with the course_id of all courses they have taught

  - Query 1
    \[ \Pi_{\text{instructor.ID}, \text{course_id}} (\sigma_{\text{dept_name} = \text{"Physics"}} (\sigma_{\text{instructor.ID} = \text{teaches.ID}} (\text{instructor} \times \text{teaches}))) \]

  - Query 2
    \[ \Pi_{\text{instructor.ID}, \text{course_id}} (\sigma_{\text{instructor.ID} = \text{teaches.ID}} (\sigma_{\text{dept_name} = \text{"Physics"}} (\text{instructor} \times \text{teaches}))) \]
Outline

- SQL Basics
- Relational Algebra
- Formal Semantics of SQL
By definition, *relations are sets*
- So $\rightarrow$ No duplicates allowed

**Problem:**
- Not practical to remove duplicates after every operation
- Why?

So...
- SQL by default does not remove duplicates

**SQL follows bag semantics, not set semantics**
- Implicitly we keep count of number of copies of each tuple
RA can only express \textit{SELECT DISTINCT} queries

- To express SQL, must extend RA to a \textit{bag} algebra
  \[\text{Bags (aka: multisets) like sets, but can have duplicates}\]

\[\text{e.g: } \{5, 3, 3\}\]

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

<table>
<thead>
<tr>
<th>c name</th>
<th>c city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson Smith</td>
<td>Brighton Perry</td>
</tr>
<tr>
<td>Johnson Smith</td>
<td>Brighton R.H.</td>
</tr>
</tbody>
</table>

- Next: will define RA*: a \textbf{bag} version of RA
1. $\sigma^*_{p}(r)$: preserves copies in $r$

   e.g.: $\sigma^*_{\text{city} = \text{Brighton}}(\text{homes}) =

   \begin{array}{|c|c|}
   \hline
   \text{cname} & \text{ccity} \\
   \hline
   \text{Johnson} & \text{Brighton} \\
   \text{Johnson} & \text{Brighton} \\
   \hline
   \end{array}

2. $\pi^*_{A_1, \ldots, A_n}(r)$: no duplicate elimination

   e.g.: $\pi^*_{\text{cname}}(\text{homes}) =

   \begin{array}{|c|}
   \hline
   \text{cname} \\
   \hline
   \text{Johnson} \\
   \text{Smith} \\
   \text{Johnson} \\
   \text{Smith} \\
   \hline
   \end{array}
3. \( r \cup^* s : \) \textit{additive union}

\[
\begin{array}{ccc}
A & B \\
1 & \alpha \\
1 & \alpha \\
2 & \beta \\
\end{array}
\cup^*

\begin{array}{ccc}
A & B \\
2 & \beta \\
3 & \alpha \\
1 & \alpha \\
\end{array}

\begin{array}{ccc}
A & B \\
1 & \alpha \\
1 & \alpha \\
2 & \beta \\
2 & \beta \\
3 & \alpha \\
1 & \alpha \\
\end{array}

4. \( r -* s : \) \textit{bag difference}

\[ r -* s = \begin{array}{ccc}
A & B \\
1 & \alpha \\
\end{array} \]

\[ s -* r = \begin{array}{ccc}
A & B \\
3 & \alpha \\
\end{array} \]
Formal Semantics of SQL: RA*

5. \( r \times^* s: \) cartesian product

\[
\begin{array}{|c|c|}
\hline
A & B \\
\hline
1 & \alpha \\
1 & \alpha \\
2 & \beta \\
\hline
\end{array}
\]

\times^*

\[
\begin{array}{|c|}
\hline
C \\
\hline
+ \\
- \\
\hline
\end{array}
\]

= 

\[
\begin{array}{|c|c|c|}
\hline
A & B & C \\
\hline
1 & \alpha & + \\
1 & \alpha & - \\
1 & \alpha & + \\
2 & \beta & + \\
2 & \beta & - \\
\hline
\end{array}
\]
Formal Semantics of SQL

Query: 
```
SELECT a_1, ..., a_n
FROM r_1, ..., r_m
WHERE p
```

Semantics: 
\[
\pi^*_{A_1, ..., A_n} (\sigma^*_p (r_1 \times * ... \times * r_m)) \quad (1)
\]

Query: 
```
SELECT DISTINCT a_1, ..., a_n
FROM r_1, ..., r_m
WHERE p
```

Semantics: 
What is the only operator to change in (1)? 
\[
\pi_{A_1, ..., A_n} (\sigma^*_p (r_1 \times * ... \times * r_m)) \quad (2)
\]
Set/Bag Operations Revisited

- **Set Operations**
  - UNION \( \equiv U \)
  - INTERSECT \( \equiv \cap \)
  - EXCEPT \( \equiv - \)

- **Bag Operations**
  - UNION ALL \( \equiv U^* \)
  - INTERSECT ALL \( \equiv \cap^* \)
  - EXCEPT ALL \( \equiv -* \)

**Duplicate Counting:**

Given \( m \) copies of \( t \) in \( r \), \( n \) copies of \( t \) in \( s \), how many copies of \( t \) in:

- \( r \) UNION ALL \( s \)?
  - A: \( m + n \)

- \( r \) INTERSECT ALL \( s \)?
  - A: \( \min (m, n) \)

- \( r \) EXCEPT ALL \( s \)?
  - A: \( \max (0, m-n) \)
## SQL: Summary

<table>
<thead>
<tr>
<th>Clause</th>
<th>Eval Order</th>
<th>Semantics (RA/RA*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT [(DISTINCT)] FROM WHERE INTO GROUP BY HAVING ORDER BY AS UNION ALL UNION (similarly intersection, except)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>$\pi$ (or $\pi^*$)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>$\times^*$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$\sigma^*$</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>$\Leftarrow$</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Extended relational operator $g$</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>$\sigma^*$</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Can’t express: requires ordered sets, bags</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>$\rho$</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>$U^*$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$U$</td>
</tr>
</tbody>
</table>
Is that it?
  ◦ Unfortunately No
  ◦ SQL 3 standard is several hundreds of pages (if not several thousands)
  ◦ And expensive too..

We will discuss a few more constructs along the way
  E.g. *Embedded SQL*, *creating indexes* etc

Again, this is what the reference books are for; you just need to know where to look in the reference book