CMSC424: Database Design
Week 3

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

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

- Relational Algebra

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

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…

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.
Outline

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

- 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
  ```java
  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
  ```java
  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()) Systems.out.println("Got null value");`
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
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;-- ?

Well, we've lost this year's student records. I hope you're happy.

Oh, yes. Little Bobby tables, we call him.

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  
```java
    # 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

- 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:
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

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

- 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

```sql
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 `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

- Relational Algebra

- Formal Semantics of SQL (i.e., how to deal with duplicates)
Rank instructors by salary.

```sql
select *, rank() over (order by salary desc) as s_rank
from instructor;
```

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

```sql
select ID, (1 + (select count(*)
from instructors i2
where i2.salary > i1.salary)) as s_rank
from instructor i1
order by s_rank;
```
Ranking (Cont.)

- 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
- 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.
**Select Operation**

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>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>

## $\Pi_{A,D}(r)$

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>7</td>
</tr>
<tr>
<td>α</td>
<td>7</td>
</tr>
<tr>
<td>β</td>
<td>3</td>
</tr>
<tr>
<td>β</td>
<td>10</td>
</tr>
</tbody>
</table>

## SQL Equivalent:

```sql
select distinct A, D
from r
```
**Set Union, Difference**

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>α</td>
<td>2</td>
</tr>
<tr>
<td>β</td>
<td>1</td>
</tr>
</tbody>
</table>

\( r \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>2</td>
</tr>
<tr>
<td>β</td>
<td>3</td>
</tr>
</tbody>
</table>

\( s \)

SQL Equivalent:

- \( r \cup s: \)
  
  ```sql
  select * from r
  union/except/intersect
  select * from s;
  ```

- \( r - s: \)
  
  ```sql
  select * from r
  - ( r - s);
  ```

This is one case where duplicates are removed.

Must be compatible schemas

What about intersection?

Can be derived

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

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>$\beta$</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>$\gamma$</td>
<td>10</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

$r \times s$:

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>$\alpha$</td>
<td>1</td>
<td>$\beta$</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>$\alpha$</td>
<td>1</td>
<td>$\beta$</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>2</td>
<td>$\alpha$</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>2</td>
<td>$\beta$</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>2</td>
<td>$\beta$</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>2</td>
<td>$\gamma$</td>
<td>10</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

SQL Equivalent:

```sql
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** ($\cap$)
  - $r \cap s = r - (r - s)$;
  - SQL Equivalent: `intersect`

- **Assignment** (←)
  - A convenient way to right complex RA expressions
  - Essentially for creating “temporary” relations
    - $temp1 \leftarrow \prod_{R-S} (r)$
  - SQL Equivalent: “create table as...”
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:
    \[
    r \bowtie s = \prod_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \land r.D = s.D} (r \times s))
    \]

SQL Equivalent:

- \( \text{select } r.A, r.B, r.C, r.D, s.E \text{ from } r, s \text{ where } r.B = s.B \text{ and } r.D = s.D \), OR
- \( \text{select } * \text{ from } r \text{ natural join } s \)
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: $(r - \pi_{r.A, r.B}(r \bowtie s))$
  - We are done:
    $$(r \bowtie s) \cup \rho_{\text{temp}}(A, B, C) ((r - \pi_{r.A, r.B}(r \bowtie s)) \times \{(NULL)\})$$
## 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>$\times$</td>
<td>select * from $r, s$;</td>
<td>$r \times s$</td>
</tr>
<tr>
<td>natural join</td>
<td>$\bowtie$</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>$\bowtie_{\theta}$</td>
<td>from .. where $\theta$;</td>
<td>$\sigma_{\theta}(r \times s)$</td>
</tr>
<tr>
<td>equi-join</td>
<td>$\bowtie_{\theta}$</td>
<td>(theta must be equality)</td>
<td></td>
</tr>
<tr>
<td>left outer join</td>
<td>$r \leftarrow s$</td>
<td>left outer join (with “on”)</td>
<td>(see previous slide)</td>
</tr>
<tr>
<td>full outer join</td>
<td>$r \rightarrow s$</td>
<td>full outer join (with “on”)</td>
<td>$-$</td>
</tr>
<tr>
<td>(left) semijoin</td>
<td>$r \Leftarrow s$</td>
<td>none</td>
<td>$\pi_{r.A, r.B} (r \Leftarrow s)$</td>
</tr>
<tr>
<td>(left) antijoin</td>
<td>$r \Rightarrow s$</td>
<td>none</td>
<td>$r - \pi_{r.A, r.B} (r \Leftarrow s)$</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>

\[ \begin{array}{cc}
\alpha & 1 \\
\beta & 2 \\
\end{array} \] \[ \div \] \[ \begin{array}{c}
\alpha \\
\beta \\
\end{array} \] = \[ \begin{array}{ccc}
C & D & E \\
\alpha & 10 & a \\
\beta & 10 & b \\
\beta & 20 & b \\
\gamma & 10 & b \\
\end{array} \]
Relational Algebra Examples

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, …)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)
Relational Algebra Examples
Relational Algebra Examples

Find all loans of over $1200

\[ \sigma_{\text{amount} > 1200} (\text{loan}) \]

Find the loan number for each loan of an amount greater than $1200

\[ \Pi_{\text{loan-number}} (\sigma_{\text{amount} > 1200} (\text{loan})) \]

Find the names of all customers who have a loan, an account, or both, from the bank

\[ \Pi_{\text{customer-name}} (\text{borrower}) \cup \Pi_{\text{customer-name}} (\text{depositor}) \]
Relational Algebra Examples

Find the names of all customers who have a loan and an account at bank.

\[ \Pi_{\text{customer-name}} (\text{borrower}) \cap \Pi_{\text{customer-name}} (\text{depositor}) \]

Find the names of all customers who have a loan at the Perryridge branch.

\[ \Pi_{\text{customer-name}} (\sigma_{\text{branch-name} = \text{“Perryridge”}} ( \sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower x loan}))) \]

Find the largest account balance

1. Rename the account relation a \( d \)

\[ \Pi_{\text{balance}} (\text{account}) - \Pi_{\text{account.balance}} (\sigma_{\text{account.balance} < d.\text{balance}} (\text{account x } \rho_{d} (\text{account}))) \]
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 $d$
    - $\prod_{instructor\_salary} (\sigma_{instructor\_salary < d,salary}
      (instructor \ast \rho_d (instructor)))$
  - Step 2: Find the largest salary
    - $\prod_{salary\_instructor} (\sigma_{instructor\_salary < d,salary}
      (instructor \ast \rho_d (instructor)))$
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
  \[ \Pi_{\text{instructor.ID, course_id}} (\sigma_{\text{dept_name} = \text{"Physics"}} (\sigma_{\text{instructor.ID} = \text{teaches.ID}} (\text{instructor} \times \text{teaches}))) \]

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

- SQL Basics
- Relational Algebra
- Formal Semantics of SQL
Duplicates

- By definition, *relations are sets*
  - So → 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 \texttt{SELECT DISTINCT} queries

To express SQL, must extend RA to a \textbf{bag} algebra

\textbf{Bag}s (aka: \textit{multisets}) like sets, but can have duplicates

\textit{e.g.} \{5, 3, 3\}

\textit{e.g.} \texttt{homes =}

\begin{tabular}{|l|l|}
\hline
\texttt{cname} & \texttt{ccity} \\
\hline
Johnson Smith & Brighton Perry \\
Johnson Smith & Brighton R.H. \\
\hline
\end{tabular}

Next: will define \textbf{RA*: a bag} version of RA
Formal Semantics of SQL: RA*

1. $\sigma^*_p (r)$: preserves copies in $r$

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

<table>
<thead>
<tr>
<th>cname</th>
<th>ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
<td>Brighton</td>
</tr>
<tr>
<td>Johnson</td>
<td>Brighton</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
</tr>
<tr>
<td>Smith</td>
</tr>
<tr>
<td>Johnson</td>
</tr>
<tr>
<td>Smith</td>
</tr>
</tbody>
</table>
3. $r \cup^* s$: 

additive union

\[
\begin{array}{cc}
A & B \\
1 & \alpha \\
1 & \alpha \\
2 & \beta \\
\end{array} \quad \cup^* \quad \begin{array}{cc}
A & B \\
2 & \beta \\
3 & \alpha \\
1 & \alpha \\
\end{array} = \begin{array}{cc}
A & B \\
1 & \alpha \\
1 & \alpha \\
2 & \beta \\
2 & \beta \\
3 & \alpha \\
1 & \alpha \\
\end{array}
\]

4. $r -^* s$: 

bag difference

\[e.g.: \quad r -^* s = \begin{array}{cc}
A & B \\
1 & \alpha \\
\end{array} \quad \quad s -^* r = \begin{array}{cc}
A & B \\
3 & \alpha \\
\end{array}\]
5. \( r \times^* s: \) cartesian product

\[
\begin{array}{ccc}
A & B & C \\
1 & \alpha & + \\
1 & \alpha & - \\
2 & \beta & - \\
\end{array}
\]

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

Query:
SELECT $a_1, \ldots, a_n$
FROM $r_1, \ldots, r_m$
WHERE $p$

Semantics:
$\pi^{*}_{A_1, \ldots, A_n} (\sigma^{*}_{p} (r_1 \times^{*} \ldots \times^{*} r_m))$  \hspace{1cm} (1)

Query:
SELECT DISTINCT $a_1, \ldots, a_n$
FROM $r_1, \ldots, r_m$
WHERE $p$

Semantics: What is the only operator to change in (1)?
$\pi^{*}_{A_1, \ldots, A_n} (\sigma^{*}_{p} (r_1 \times^{*} \ldots \times^{*} r_m))$  \hspace{1cm} (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</th>
<th>Order</th>
<th>Semantics (RA/RA*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT [(DISTINCT)] FROM</td>
<td>4</td>
<td>1</td>
<td>(\pi) (or (\pi^*))</td>
</tr>
<tr>
<td>WHERE</td>
<td>1</td>
<td>2</td>
<td>(\times^*)</td>
</tr>
<tr>
<td>INTO</td>
<td>2</td>
<td>3</td>
<td>(\sigma^*)</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>7</td>
<td>4</td>
<td>(\leftarrow)</td>
</tr>
<tr>
<td>HAVING</td>
<td>3</td>
<td>5</td>
<td>Extended relational operator (g)</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>6</td>
<td>6</td>
<td>Can’t express: requires ordered sets, bags</td>
</tr>
<tr>
<td>AS</td>
<td>-</td>
<td>7</td>
<td>(\rho)</td>
</tr>
<tr>
<td>UNION ALL</td>
<td>8</td>
<td>8</td>
<td>(\mathbb{U}^*)</td>
</tr>
<tr>
<td>UNION</td>
<td></td>
<td></td>
<td>(\mathbb{U})</td>
</tr>
<tr>
<td>(similarly intersection, except)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>