Next….SQL!

- Defining relational schema
  - Table definition
  - Specify constraints and keys
- Getting started with MySQL
- SQL Queries
Relational Model Definitions
- A **relation** is a table with columns and rows.
- An **attribute** is a named column of a relation.
  - A **tuple** is a row of a relation.
- A **domain** is a set of allowable values for one or more attributes.
- A **relational database** is a collection of normalized relations with distinct relation names.
- Key: set of attributes that uniquely identify a tuple/row
  - No two rows can have the same key value
- Primary key: one of the keys to the table
- Foreign key: if an attribute in one table is the primary key in another table
  - Provides “link” between tables

Recall: Schema Design & Relational Integrity
- Integrity rules are used to insure the data is accurate.
- **Constraints** are rules or restrictions that apply to the database and limit the data values it may store.
  - DBMS checks the constraints
- Types of constraints:
  - **Domain constraint** - Every value for an attribute must be an element of the attribute's domain or be null.
    - **null** represents a value that is currently unknown or not applicable.
    - **null** is not the same as zero or an empty string.
  - **Entity integrity constraint** - In a base relation, no attribute of a primary key can be null.
  - **Key constraint** – every relation must have a key; one of them chosen as primary key
  - **Referential integrity constraint** - If a foreign key exists in a relation, then the foreign key value must match a primary key value of a tuple in the referenced relation or be null.
Referential integrity and Foreign Keys

- Only students listed in the Students relation should be allowed to enroll for courses.
- Sid in Enrolled is foreign key referencing students
  - Sid is key for Students table

<table>
<thead>
<tr>
<th>Enrolled</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>sid</td>
</tr>
<tr>
<td>53666</td>
<td>53666</td>
</tr>
<tr>
<td>Jazz101</td>
<td>Jones</td>
</tr>
<tr>
<td>53666</td>
<td>53688</td>
</tr>
<tr>
<td>Reggae203</td>
<td>Smith</td>
</tr>
<tr>
<td>53650</td>
<td>53688</td>
</tr>
<tr>
<td>Topology112</td>
<td>smith@eecs</td>
</tr>
<tr>
<td>53666</td>
<td>53650</td>
</tr>
<tr>
<td>History105</td>
<td>Smith</td>
</tr>
</tbody>
</table>

Next: SQL Module 1

- Specifying schema/table
- Specifying constraints in SQL
SQL: Structured Query Language

The standard language for relational data
- Invented by folks at IBM, esp. Don Chamberlin
- Actually not a great language…
- Beat a more elegant competing standard, QUEL, from Berkeley

Separated into a DML & DDL

SQL DML component based on relational algebra & calculus

- Data definition (DDL) – to define schema/tables
  - Define Schema
  - Define Constraints

SQL Basic Rules…read up on SQL syntax

- Some basic rules for SQL statements:
  - 1) There is a set of reserved words that cannot be used as names for database objects. (e.g. SELECT, FROM, WHERE)
  - 2) SQL is case-insensitive.
    Only exception is string constants. 'FRED' not the same as 'fred'.
  - 3) SQL is free-format and white-space is ignored.
  - 4) The semi-colon is often used as a statement terminator, although that is not always required.
  - 5) Date and time constants have defined format:
    - Dates: ‘YYYY-MM-DD’ e.g. ‘1975-05-17’
    - Times: ‘hh:mm:ss[f]’ e.g. ‘15:00:00’
    - Timestamp: ‘YYYY-MM-DD hh:mm:ss[f]’ e.g. ‘1975-05-17 15:00:00’
  - 6) Two single quotes ‘’ are used to represent a single quote character in a character constant. e.g. ‘Master’s’.
SQL Query Language: DML

To query and retrieve data from the tables we have a:
- SELECT clause
  - What attributes you want
  - What relations/tables to search
  - What condition/predicate to apply

SQL and Relational Algebra

- The SELECT statement can be mapped directly to relational algebra.

- SELECT $A_1, A_2, \ldots, A_n$ /* this is projection
- FROM $R_1, R_2, \ldots, R_m$ /* this is the cartesian prod
- WHERE $P$ /* this is selection op

is equivalent to:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(R_1 \times R_2 \times \ldots \times R_m))$$

More on this later…
SQL DDL
- SQL data definition language (DDL) allows users to:
  - add, modify, and drop tables
  - define and enforce integrity constraints
  - enforce security restrictions
  - Create views

SQL Identifiers and Data types...standard definitions you've seen before in other languages
- **Identifiers** are used to identify objects in the database such as tables, views, and columns.
  - The identifier is the name of the database object.
  - Rules for SQL identifiers…read notes
  - Note: Quoted or **delimited identifiers** enclosed in double quotes allow support for spaces and other characters. E.g. “select”
- Data types: each attribute has associated domain of values – i.e., each column has data type
  - The DBMS can perform implicit data type conversion when necessary
  - Can also do explicit conversion using CAST and CONVERT
- SQL also supports **user defined data types**
  - CREATE DOMAIN
  - Similar to typedef in C?
### SQL Data Types...similar to prog lang

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>TRUE or FALSE</td>
</tr>
<tr>
<td>CHAR</td>
<td>Fixed length string (padded with blanks) e.g. CHAR(10)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>Variable length string e.g. VARCHAR(50)</td>
</tr>
<tr>
<td>BIT</td>
<td>Bit string e.g. BIT(4) can store '0101'</td>
</tr>
<tr>
<td>NUMERIC or DECIMAL</td>
<td>Exact numeric data type e.g. NUMERIC(7,2) has a precision (max. digits) of 7 and scale of 2 (# of decimals) e.g. 12345.67</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integer data only</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Smaller space than INTEGER</td>
</tr>
<tr>
<td>FLOAT or REAL</td>
<td>Approximate numeric data types.</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>Precision dependent on implementation.</td>
</tr>
<tr>
<td>DATE</td>
<td>Stores YEAR, MONTH, DAY</td>
</tr>
<tr>
<td>TIME</td>
<td>Stores HOUR, MINUTE, SECOND</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Stores date and time data.</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>Time interval.</td>
</tr>
<tr>
<td>CHARACTER LARGE OBJECT</td>
<td>Stores a character array (e.g. for a document)</td>
</tr>
<tr>
<td>BINARY LARGE OBJECT</td>
<td>Stores a binary array (e.g. for a picture, movie)</td>
</tr>
</tbody>
</table>

### COMPANY Database Schema

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
<th>Frame</th>
<th>Minit</th>
<th>Lname</th>
<th>Ssn</th>
<th>Bdate</th>
<th>Address</th>
<th>Sex</th>
<th>Salary</th>
<th>Super_sen</th>
<th>Dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT</td>
<td>Dname</td>
<td>Dnumber</td>
<td>Mgr_ssn</td>
<td>Mgr_start_date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPT_LOCATIONS</td>
<td>Dnumber</td>
<td>Dlocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROJECT</td>
<td>Pname</td>
<td>Pnumber</td>
<td>Plocation</td>
<td>Dnum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORKS_ON</td>
<td>Eain</td>
<td>Pho</td>
<td>Hours</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPENDENT</td>
<td>Eain</td>
<td>Dependent_name</td>
<td>Sex</td>
<td>Bdata</td>
<td>Relationship</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Referential Integrity Constraints for COMPANY database

From FK to PK ex: from Dno in EMP to Dnumber in DEPT

Example Schema
- Relational database schema:

  employee (ssn, fname, Minit, Lname, bdate, address, Gender, salary, superssn, Dno)
  project (pnumber, pname, Plocation, Dnum)
  department (dnumber, dname, mgrssn, Mgr_start_date)
  workson (essn, pno, hours)
**SQL CREATE TABLE**

- The `CREATE TABLE` command is used to create a table in the database. A table consists of a table name, a set of fields with their names and data types, and specified constraints.

- The general form is:

```sql
CREATE TABLE tableName (  
    attr1Name attr1Type  [attr1_constraints],  
    attr2Name attr2Type  [attr2_constraints],  
    ...  
    attrMName attrMType  [attrM_constraints],  
    [primary and foreign key constraints]  
) ;
```

**SQL CREATE TABLE Example**

- The `CREATE TABLE` command for the `Emp` relation:

```sql
CREATE TABLE employee (  
    ssn CHAR(9),  
    fname VARCHAR(15) NOT NULL,  
    minit CHAR(1),  
    lname CHAR(15),  
    bdate DATE,  
    sex CHAR(1),  
    salary DECIMAL(10,2),  
    superssn CHAR(9),  
    dno INT(4),  
) ;
```
SQL Constraints - Entity Integrity

- Entity Integrity constraint - The primary key of a table must contain a unique, non-null value for each row. The primary key is specified using the PRIMARY KEY clause.
  - e.g. PRIMARY KEY (ssn) (for Emp relation)
  - e.g. PRIMARY KEY (essn, pno) (for WorksOn relation)
  - It is also possible to use PRIMARY KEY right after defining the attribute in the CREATE TABLE statement.

- There can only be one primary key per relation, other candidate keys can be specified using UNIQUE:
  - e.g. UNIQUE (lname)

Another Example... 'mini-banner'

- Create Students table
  - Info on students
- Takes table holds information about courses that students take.
  - Is sid same field in the two tables??

CREATE TABLE Students
(sid: CHAR(20),
name: CHAR(20),
PRIMARY KEY (sid)) ;

CREATE TABLE Takes
(sid: CHAR(20),
cid: CHAR(20),
grade: CHAR(2))
Specifying constraints on Takes table

- A reasonable condition/constraint: “For a given student and course, there is a single grade”

```
CREATE TABLE Enrolled
    (sid: CHAR(20),
    cid: CHAR(20),
    grade: CHAR(2))
```

Does this schema have any problems?

```
CREATE TABLE Enrolled2
    (sid CHAR(20),
    cid CHAR(20),
    grade CHAR(2),
    PRIMARY KEY (sid),
    UNIQUE (cid, grade)
)`
Effect of incorrect constraints...

- Enrolled1: “For a given student and course, there is a single grade.” vs. Enrolled 2: “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”
- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

SQL Constraints - Referential Integrity

- **Referential integrity constraint** - Defines a foreign key that references the primary key of another table.
  - If a foreign key contains a value that is not NULL, that value must be present in some tuple in the relation containing the referenced primary key.

- Example: Workson contains two foreign keys:
  - workson.essn references employee.ssn
  - workson.pno references project.pnumber

- Specify foreign keys using **FOREIGN KEY syntax**:

  ```sql
  FOREIGN KEY (essn) REFERENCES employee(ssn)
  ```
SQL Referential Integrity

- The CREATE TABLE command for the workson relation:

```sql
CREATE TABLE workson (
    essn    CHAR(9),
    pno     INT(4),
    hours   DECIMAL(4,1),
    PRIMARY KEY (essn, pno),
    FOREIGN KEY (essn) REFERENCES employee(ssn),
    FOREIGN KEY (pno) REFERENCES project(pnumber)
);```

SQL Referential Integrity and Updates

- When you try to INSERT or UPDATE a row in a relation containing a foreign key (e.g. workson) that operation is rejected if it violates referential integrity.

- When you UPDATE or DELETE a row in the primary key relation (e.g. emp or proj), you have the option on what happens to the values in the foreign key relation (workson):
  1) CASCADE - Delete (update) values in foreign key relation when primary key relation has rows deleted (updated).
  2) SET NULL - Set foreign key fields to NULL when corresponding primary key relation row is deleted.
  3) SET DEFAULT - Set foreign key values to their default value (if defined).
  4) NO ACTION - Reject the request on the parent table.
SQL Referential Integrity Example (2)

CREATE TABLE workson (  
    essn CHAR(9),  
    pno INT(4),  
    hours DECIMAL(4,1),  
    PRIMARY KEY (essn,pno),  
    FOREIGN KEY (essn) REFERENCES employee(ssn)  
        ON DELETE NO ACTION  
        ON UPDATE CASCADE,  
    FOREIGN KEY (pno) REFERENCES project(pnumber)  
        ON DELETE NO ACTION  
        ON UPDATE CASCADE  
);  

You don’t want to delete an employee who is still Working on a project…delete from WorksOn first

SQL CREATE TABLE Example

- The CREATE TABLE command for the Emp relation:

```
CREATE TABLE employee (  
    ssn CHAR(9),  
    lname VARCHAR(15) NOT NULL,  
    ...  
    superssn CHAR(9),  
    dno INT(4),  
    PRIMARY KEY (eno),  
    FOREIGN KEY (dno) REFERENCES department(dnum)  
        ON DELETE SET NULL ON UPDATE CASCADE,  
    FOREIGN KEY (superssn) REFERENCES employee(ssn)  
        ON DELETE SET DEFAULT ON UPDATE CASCADE,  
);  
```

If a department is deleted, do not fire the employee IF supervisor is deleted, set to default supervisor
Domain Constraints SQL

- Name should not be NULL
- Age > 10 (restrict values in that domain)
- Other constraints…
  - Can specify SQL query

```sql
CREATE TABLE Students
(sid CHAR(20),
 name: CHAR(20) NOT NULL,
 login CHAR(10),
 age INTEGER,
 gpa: REAL,
 CHECK (age > 10) )  ;
```

**SQL CREATE TABLE Full Syntax**

- Full syntax of `CREATE TABLE` statement:

```sql
CREATE TABLE tableName (
  { attrName attrType [NOT NULL] [UNIQUE] [PRIMARY KEY]
   [DEFAULT value] [CHECK (condition)] }
  [PRIMARY KEY (colList)]
  [{FOREIGN KEY (colList) REFERENCES tbl [(colList)],
    [ON UPDATE action]
    [ON DELETE action] }]
  [CHECK (condition)]
);
```

Important: MySQL does not support CHECK operator
Implement this using TRIGGERS
- Will return to this in a few weeks
Database Updates

- Database updates such as inserting rows, deleting rows, and updating rows are performed using their own statements.
- INSERT
- UPDATE
- DELETE

Database Updates

- Insert is performed using the INSERT command:
  
  \[
  \text{INSERT INTO } \text{tableName} [\text{(column list)}] \text{VALUES} \text{(data value list)}
  \]

- Examples:
  
  ```
  INSERT INTO employee VALUES ('James','E','Borg','888665555','1927-11-10',
  '450 Stone, Houston, TX','M',55000,null,null);
  INSERT INTO project (pno, pname)
  VALUES ('P6','Programming');
  ```

Note: If column list is omitted, values must be specified in order they were created in the table. If any columns are omitted from the list, they are set to NULL.
Changing/Deleting Tables/Schema...Read on your own

- The **ALTER TABLE** command can be used to change an existing table. This is useful when the table already contains data and you want to add or remove a column or constraint.
  - DB vendors may support only parts of **ALTER TABLE** or may allow additional changes including changing the data type of a column.
- The command **DROP TABLE** is used to delete the table definition and all data from the database:
  ```sql
  DROP TABLE tableName [RESTRICT | CASCADE];
  ```

DDL Summary

- SQL contains a data definition language that allows you to **CREATE**, **ALTER**, and **DROP** database objects such as tables, triggers, indexes, schemas, and views.
- Constraints are used to preserve the integrity of the database:
  - **CHECK** can be used to validate attribute values.
  - **Entity Integrity constraint** - The primary key of a table must contain a unique, non-null value for each row.
  - **Referential Integrity constraint** - Defines a foreign key that references a unique key of another table.
- **INSERT, DELETE, and UPDATE** commands modify the data stored within the database.
The web server processed web page request, runs PHP scripts, and returns HTML content.

The database server reads and writes data from/to the database.

The database itself is often stored as files on a hard drive, but it doesn’t necessarily have to be.

### Connecting to mySQL on gwupyterhub

- **Use your GW netID to connect to the gwupyterhub.seas.gwu.edu server**
  
  ```
  ssh -Y GWnetID@gwupyterhub.seas.gwu.edu
  ```

- **Login into MySQL**
  
  ```
  mysql -u GWnetID -p
  ```

- **Reset your password**
  
  ```
  SET PASSWORD FOR 'GWNetID''localhost'='NEWPASSWORD';
  ```

- **NOTE:** use your GW NetID, WITH the password `CSCI2541_sp20`
MySQL Database

- An existing database is available for your use

  ```sql
  show databases;
  ```

- To use your database:

  ```sql
  use database_name;
  ```

  **NOTE:** use your GW NetID for database name
MySQL table creation

- Syntax to create a table example

CREATE TABLE works_on (  
  essn char(9),  
  pno int(4),  
  hours decimal(4,1),  
  primary key (essn,pno),  
  foreign key (essn) references employee(ssn),  
  foreign key (pno) references project(pnumber)  
);  

MySQL basic Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char(n)</td>
<td>Fixed length character string of length n (max 255)</td>
</tr>
<tr>
<td>varchar(n)</td>
<td>Variable length character string (max 255)</td>
</tr>
<tr>
<td>date</td>
<td>holds a date field (28-Jan-2013)</td>
</tr>
<tr>
<td>decimal(n,d)</td>
<td>real numbers occupying up to n spaces with d digits after the decimal point</td>
</tr>
<tr>
<td>int(n)</td>
<td>integer with up to n digits</td>
</tr>
</tbody>
</table>
MySQL Table Creation - Example

```
CREATE TABLE project (
  -> pname  varchar(25) not null,
  -> pnumber int(3),
  -> plocation varchar(15),
  -> dname  varchar(25) not null,
  -> primary key (pnumber),
  -> unique (pname),
  -> foreign key (dname) references department(dnumber)
  ->);
```

**MySQL table operations:**

- **Show structure**
  
  ```
  describe tableName;
  ```

- **Modify structure**
  
  ```
  ALTER TABLE employee ADD (bdate date);
  ALTER TABLE project DROP column plocation;
  ALTER TABLE department MODIFY COLUMN dname varchar(2);
  ALTER TABLE employee
  ADD foreign key (dno) references department(dnumber);
  ```

- **Remove table**
  
  ```
  DROP TABLE locations;
  ```
MySQL: INSERT

The SQL keywords INSERT INTO begin the statement. The name of the table... The next part is a list of your table column names, separated by commas.

`INSERT INTO table_name (column_name1, column_name2, ...)
VALUES ('value1', 'value2', ...)`

More quoted values follow, with no comma after the last one.

This next part is a list of the values to be inserted, separated by commas. The single quotes are correct. Use them whenever you're inserting text, even if it's a single character!

IMPORTANT: these need to be in the same order as the column names.

MySQL: INSERT - Example

```
mysql> describe employee;
+ Field + Type + Null | Key | Default | Extra |
+--------+-------+--------+------|--------+-------|
| id     | int   | YES    |    | NULL   |       |
| name   | varchar(35) | NO | | NULL | |
| idate  | date  | YES    | NO | NULL   |       |
| address| varchar(50) | YES | | NULL | |
| phone  | varchar(10) | YES | | NULL | |
| email  | varchar(50) | YES | | NULL | |
| age    | int    | YES    | NO | NULL   |       |
| salary | int    | YES    | NO | NULL   |       |
| city   | varchar(25) | YES | | NULL | |
| zip    | varchar(10) | YES | | NULL | |
| state  | varchar(2) | YES | | NULL | |
| country| varchar(25) | YES | | NULL | |

10 rows in set (0.00 sec)
```

```
mysql> INSERT INTO employee VALUES
    -> ("James","E","Borg","88665555","1927-11-15","450 Stone, Houston, TX","M",55000, null, null);
Query OK, 1 row affected (0.01 sec)
```

mysql>
Check that data was added: SELECT

Follow SELECT with a list of the columns you want data for.

A SELECT always take place with respect to a specific table, not a database in general.

The FROM part of a SELECT statement is how SELECT knows what table we’ll be selecting data from.

In class exercise:

- Create all tables for the employee schema described in Module 2
  - For each table creation record the create statements and show the description
- Run the script provided to populate the tables.
- Show all the entries in each table.

NOTE:

Because of the connectivity of these tables, when creating them, run the following command:

```sql
SET FOREIGN_KEY_CHECKS = 0;
```

After table creation:

```sql
SET FOREIGN_KEY_CHECKS = 1;
```

You will also need to alter the tables and add the foreign keys relationship after the tables dependent are created.

https://classroom.github.com/a/wBGgagip
Next: Module 3- Querying in SQL

- Querying the database
  - SQL Data Manipulation language

- Today….Simple commands
  - Equivalent to Relational Algebra

---

### Basic SQL Query

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE qualification/predicate:
```

- **relation-list** A list of relation names (possibly with a **range-variable**, i.e., **tuple variable**, after each name).
- **attribute-list** A list of attributes of relations in **relation-list**
- **Qualification/predicate** Comparisons (Attr op const or Attr1 op Attr2, where op is one of `<`, `>`, `=`, `≤`, `≥`, `≠`) combined using AND, OR and NOT.
- **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!
- To select all attributes in result, we use *
**SQL and Relational Algebra**

- The `SELECT` statement can be mapped directly to relational algebra.

```
SELECT A_1, A_2, ... , A_n  \text{ this is projection } \pi
FROM  R_1, R_2, ... , R_m  \text{ this is Cartesian product } \times
WHERE  P  \text{ this is the selection op } \sigma
```

- is equivalent to:

```
\Pi_{A_1, A_2, ..., A_n}(\sigma_P(R_1 \times R_2 \times ... \times R_m))
```

- If we don’t want to project, then `SELECT *`

---

**Conceptual Evaluation Strategy**

- *Semantics* of an SQL query defined in terms of the following conceptual evaluation strategy:

  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *predicate qualifications*.
  - Delete attributes that are not in *target attribute-list*.
    - If *DISTINCT* is specified, eliminate duplicate rows.
    - SQL allows duplicates in relations (unlike Rel. Algebra)
  - This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*. 
Algorithm: Scan each tuple in table and check if matches condition in WHERE clause.
Duplicates in SQL

SQL returns ‘bag of words’
  duplicates allowed in contrast to relational algebra

To remove duplicates use DISTINCT clause:

SELECT DISTINCT title
FROM emp;

Eliminating Duplicates

- **SELECT DISTINCT category**
  FROM Product

Compare to:

- **SELECT category**
  FROM Product
Expressions and Strings

- Illustrates use of arithmetic expressions and string pattern matching: Find students whose name includes ‘Sam’.
- LIKE is used for string matching. ‘_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.

Find students whose name begins with S and at least three characters:
  - Replace with ‘S_ _ %’

Ordering Result Data

- The query result returned is not ordered on any attribute by default. We can order the data using the ORDER BY clause:
- STUDENTS [ sid, name]

- Replace with ‘S_ _ %’
Ordering Result Data using multiple attributes

- Can define sort on major and minor sort keys
  - Corresponds to “filing order” of k-tuples

emp: [ssn, ename, salary]

SELECT ename, salary
FROM emp
WHERE salary > 30000
ORDER BY salary DESC, ename ASC

- The order of sorted attributes is significant. The first attribute specified is sorted on first, then the second attribute is used to break any ties, etc.

Bank Database Schema

<table>
<thead>
<tr>
<th>Branch</th>
<th>Assets</th>
<th>Branch_City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch_Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CustID</td>
<td>LoanNo</td>
<td>Amount</td>
</tr>
<tr>
<td>Deposit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CustID</td>
<td>AccNo</td>
<td>Balance</td>
</tr>
<tr>
<td>Customer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CustID</td>
<td>Name</td>
<td>Street</td>
</tr>
</tbody>
</table>
**Schema of Bank DB**

- **Customer** (`CustID`, Name, street, city, zip)
  - Key is `CustID` of type int (?)
  - Name, street, city can vary in length – varchar(20)
  - `CustID` and `zip` can be integer

- **Deposit** (`CustID`, `Acct-num`, balance, `Branch-name`)
  - `Acct-Num` is key, type int (?)
  - `CustID` foreign key references `Customer`
  - `Branch-name` is varchar(20) references `Branch`
  - Balance is a real number, i.e., decimal
  - Customer can have many accounts; cannot have joint accounts

- **Loan** (`CustID`, `Loan-num`, Amount, `Branch-name`)

- **Branch** (`Branch-name`, assets, `Branch-city`)

**IN CLASS EXERCISE**

- Follow the instructions in your GitHub repository to populate the Bank database.
- TODO: move 4 queries here
Joins in SQL

- Multiple tables can be queried in a single SQL statement by listing them in the `FROM` clause.
  - Note that if you do not specify any join condition to relate them in the `WHERE` clause, you get a *cross product* of the tables.

```sql
Joins

Product (pname, price, category, manufacturer)

Company (cname, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

```

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
AND Price <= 200
```
Joins

Find all products under $200 manufactured in Japan; return their names and prices.

### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Threshold</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

### Company

<table>
<thead>
<tr>
<th>CName</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

### SQL Query

```sql
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan' AND Price <= 200
```

Renaming and Aliasing

- Does the job of rename operator \( \rho \) in relational algebra
- Often it is useful to be able to rename an attribute in the final result (especially when using calculated fields). Renaming is accomplished using the keyword `AS`:

```sql
SELECT lname, salary AS pay
FROM employee
WHERE dno=5;
```

**Result**

<table>
<thead>
<tr>
<th>lname</th>
<th>pay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lee</td>
<td>100000.00</td>
</tr>
<tr>
<td>Smith</td>
<td>60000.50</td>
</tr>
<tr>
<td>Lee</td>
<td>90000.00</td>
</tr>
</tbody>
</table>

Note: AS keyword is optional.
Renaming…Using Tuple/Range variables

- Concept of tuple/range variables borrowed from relational calculus
  - Tuple $t$ of type $R$: $t \in R$
  - What about $x \in R$, $y \in R$
- It performs the job of the rename operator from relational algebra
  - One variable with name $x$ and one with name $y$, BOTH of type $R$
- Need to worry about scope of tuple variables when we have nested queries

Aliasing to remove ambiguity…The easy case:

Person($\text{pname}$, address, worksfor)
Company($\text{cname}$, address)

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT} \quad \text{pname, address} \\
\text{FROM} & \quad \text{Person, Company} \\
\text{WHERE} & \quad \text{worksfor} = \text{cname}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT} \quad \text{Person.pname, Company.address} \\
\text{FROM} & \quad \text{Person, Company} \quad /*\text{named field notation} \\
\text{WHERE} & \quad \text{Person.worksfor} = \text{Company.cname}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT} \quad x.\text{pname}, y.\text{address} \\
\text{FROM} & \quad \text{Person AS } x, \text{ Company AS } y \quad /*\text{ aliasing} \\
\text{WHERE} & \quad x.\text{worksfor} = y.\text{cname}
\end{align*}
\]
Tuple Variables

Person(pname, address, worksfor)
Company(cname, address)

SELECT DISTINCT P.pname, C.address
FROM Person P, Company C
WHERE P.worksfor = C.cname;

x is a copy of Person, y is a copy of Company
P is a variable of 'type' Person C is a variable of 'type' Company

Renaming: Joining table with itself

- Aliases/Tuple variables must be used when relation has to be 'joined' with itself – i.e., two or more copies of the same table are needed. Using aliases allows you to uniquely identify what table you are talking about.

Example: Return last names of employees and their managers.

SELECT E.lname, M.lname
FROM employee E, employee M
WHERE E.superssn = M.ssn;

- E is a variable of type Employee, and denotes an employee
- M is a variable of type Employee, and denotes (will bind to) values of supervisor
Meaning (Semantics) of SQL Queries with tuple variables

```
SELECT a1, a2, ..., ak 
FROM R1 x1, R2 x2, ..., Rn xn 
WHERE Conditions
```

```
Answer = {} 
for x1 in R1 do 
  for x2 in R2 do 
    ....
      for xn in Rn do 
        if Conditions then Answer = Answer \ { (a1,...,ak) }
  return Answer
```

Tuple variables
- Find students who are taking the same course as Sam with sid=1234.
- Need to access Takes table twice
  - Once to extract courses (X) taken by Sam with ID=1234
  - Second time to find students who are taking these X courses
- Define two “variables” A, B of ‘type’ Takes
  - B is variable that corresponds ID 1234 and its cid field is equal to “X”
  - A is a variable whose CID is equal to “X”
- SELECT B.sid
- FROM Takes A, Takes B
- WHERE A.cid = B.cid;
More SQL
- Set operations
- Aggregate operators
- GroupBy
- .......
- Next week

Next: Module 4- Test your querying skills!
- Step 1: 15 minutes
  - Do NOT code...
  - Work at your table to discuss solutions/queries – do not write down code
- Step 2: Code your queries and demo to the instructors
  - And submit solutions/code
Questions:
1. Find all rows in deposit where balance is greater than $1000
2. Find names of customers whose name is the same as the street they live on.
3. Find names and IDs of customers, ordered by name, whose name begins with a C
4. Find IDs of all customers who have Loans between 1200 and 2500
5. Find names of all customers who have Loans between 1200 and 2500.
6. Find the names and IDs of customers who live on the same street as customer(s) named Lennon
7. Find names of customers who have an account at the same branch as a customer named Lennon.