Lecture 2: Introduction to SQL
Announcements!

1. If you still have Jupyter trouble, let us know!

2. Enroll to Piazza!!!

3. People are looking for groups. Team up!

4. Enrollment should be finalized soon!

5. TA updates hopefully by Monday!
Lecture 2: Introduction to SQL
Today’s Lecture

1. SQL introduction & schema definitions
   • ACTIVITY: Table creation

2. Basic single-table queries
   • ACTIVITY: Single-table queries!

3. Multi-table queries
   • ACTIVITY: Multi-table queries!
1. SQL Introduction & Definitions
What you will learn about in this section

1. What is SQL?

2. Basic schema definitions

3. Keys & constraints intro

4. ACTIVITY: CREATE TABLE statements
SQL Motivation

• But why use SQL?

  • The relational model of data is the most widely used model today
    • Main Concept: the relation - essentially, a table

Remember: The reason for using the relational model is data independence!

Logical data independence: protection from changes in the logical structure of the data

SQL is a logical, declarative query language. We use SQL because we happen to use the relational model.
SQL Motivation

• Dark times 5 years ago.
  • Are databases dead?

• Now, as before: everyone sells SQL
  • Pig, Hive, Impala

• “Not-Yet-SQL?”
Basic SQL
SQL Introduction

• SQL is a standard language for querying and manipulating data

• SQL is a very high-level programming language
  • This works because it is optimized well!

• Many standards out there:
  • ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ....
  • Vendors support various subsets

Probably the world’s most successful parallel programming language (multicore?)
SQL is a...

• Data Definition Language (DDL)
  • Define relational *schemata*
  • Create/alter/delete tables and their attributes

• Data Manipulation Language (DML)
  • Insert/delete/modify tuples in tables
  • Query one or more tables – discussed next!
A **relation** or **table** is a multiset of tuples having the attributes specified by the schema.

Let’s break this definition down.

## Table: Product

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

---

**Product**
## Tables in SQL

### Product

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

### Multiset

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed)

- List: \([1, 1, 2, 3]\)
- Set: \(\{1, 2, 3\}\)
- Multiset: \(\{1, 1, 2, 3\}\)

i.e. no `next()`, etc. methods!
## Tables in SQL

**Product**

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

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation.

Attributes must have an **atomic** type in standard SQL, i.e. not a list, set, etc.
# Tables in SQL

## Product

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

A **tuple** or **row** is a single entry in the table having the attributes specified by the schema.

*Also referred to sometimes as a *record*.*
Tables in SQL

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

The number of tuples is the **cardinality** of the relation.

The number of attributes is the **arity** of the relation.
Data Types in SQL

• Atomic types:
  • Characters: CHAR(20), VARCHAR(50)
  • Numbers: INT, BIGINT, SMALLINT, FLOAT
  • Others: MONEY, DATETIME, ...

• Every attribute must have an atomic type
  • Hence tables are flat
Table Schemas

• The **schema** of a table is the table name, its attributes, and their types:

  Product(Pname: string, Price: float, Category: string, Manufacturer: string)

• A **key** is an attribute whose values are unique; we underline a key

  Product(Pname: string, Price: float, Category: string, Manufacturer: string)
Key constraints

A **key** is a **minimal subset of attributes** that acts as a unique identifier for tuples in a relation.

- A key is an implicit constraint on which tuples can be in the relation.
- i.e. if two tuples agree on the values of the key, then they must be the same tuple!

```sql
Students(sid:string, name:string, gpa: float)
```

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?
NULL and NOT NULL

• To say “don’t know the value” we use NULL
• NULL has (sometimes painful) semantics, more details later

Students(sid:string, name:string, gpa: float)

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Bob</td>
<td>3.9</td>
</tr>
<tr>
<td>143</td>
<td>Jim</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL, e.g., “name” in this table.
General Constraints

• We can actually specify arbitrary assertions
  • E.g. “There cannot be 25 people in the DB class”

• In practice, we don’t specify many such constraints. Why?
  • Performance!

Whenever we do something ugly (or avoid doing something convenient) it’s for the sake of performance
Summary of Schema Information

• Schema and Constraints are how databases understand the semantics (meaning) of data

• They are also useful for optimization

• SQL supports general constraints:
  • Keys and foreign keys are most important
  • We’ll give you a chance to write the others
ACTIVITY: Activity-2-1.ipynb
2. Single-table queries
What you will learn about in this section

1. The SFW query

2. Other useful operators: LIKE, DISTINCT, ORDER BY

3. ACTIVITY: Single-table queries
SQL Query

- Basic form (there are many many more bells and whistles)

```
SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>
```

Call this a SFW query.
Simple SQL Query: Selection

**Selection** is the operation of filtering a relation’s tuples on some condition.

```
SELECT * FROM Product WHERE Category = 'Gadgets'
```

<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>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

<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>
</tbody>
</table>
Simple SQL Query: Projection

**Projection** is the operation of producing an output table with tuples that have a subset of their prior attributes.

SELECT Pname, Price, Manufacturer
FROM Product
WHERE Category = 'Gadgets'

<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>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
Notation

Input schema: Product(PName, Price, Category, Manufacturer)

Output schema: Answer(PName, Price, Manufacturer)

SELECT Pname, Price, Manufacturer
FROM Product
WHERE Category = ‘Gadgets’
A Few Details

• **SQL commands** are case insensitive:
  • Same: SELECT, Select, select
  • Same: Product, product

• **Values** are **not**:
  • Different: ‘Seattle’, ‘seattle’

• Use single quotes for constants:
  • ‘abc’ - yes
  • “abc” - no
LIKE: Simple String Pattern Matching

- `s LIKE p`: pattern matching on strings
- `p` may contain two special symbols:
  - `%` = any sequence of characters
  - `_` = any single character

```
SELECT * FROM Products WHERE PName LIKE '%gizmo%'
```
DISTINCT: Eliminating Duplicates

SELECT DISTINCT Category
FROM Product

Versus

SELECT Category
FROM Product

<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>
ORDER BY: Sorting the Results

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Category='gizmo' AND Price > 50
ORDER BY Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.
ACTIVITY: Activity-2-2.ipynb
3. Multi-table queries
What you will learn about in this section

1. Foreign key constraints

2. Joins: basics

3. Joins: SQL semantics

4. ACTIVITY: Multi-table queries
Foreign Key constraints

• Suppose we have the following schema:

```plaintext
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)
```

• And we want to impose the following constraint:
  • ‘Only bona fide students may enroll in courses’ i.e. a student must appear in the Students table to enroll in a class

<table>
<thead>
<tr>
<th>Students</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td>gpa</td>
</tr>
<tr>
<td>101</td>
<td>Bob</td>
<td>3.2</td>
</tr>
<tr>
<td>123</td>
<td>Mary</td>
<td>3.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enrolled</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>student_id</td>
<td>cid</td>
<td>grade</td>
</tr>
<tr>
<td>123</td>
<td>564</td>
<td>A</td>
</tr>
<tr>
<td>123</td>
<td>537</td>
<td>A+</td>
</tr>
</tbody>
</table>

We say that student_id is a **foreign key** that refers to Students
Declaring Foreign Keys

```
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)

CREATE TABLE Enrolled(
    student_id  CHAR(20),
    cid         CHAR(20),
    grade       CHAR(10),
    PRIMARY KEY (student_id, cid),
    FOREIGN KEY (student_id) REFERENCES Students(sid)
)
```
Foreign Keys and update operations

Students(\textit{sid}: string, \textit{name}: string, \textit{gpa}: float)

Enrolled(\textit{student\_id}: string, \textit{cid}: string, \textit{grade}: string)

• What if we insert a tuple into Enrolled, but no corresponding student?
  • INSERT is rejected (foreign keys are constraints)!

• What if we delete a student?
  1. Disallow the delete
  2. Remove all of the courses for that student
  3. \textit{SQL allows a third via Null} (not yet covered)

\textit{DBA chooses (syntax in the book)}
Keys and Foreign Keys

### 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>

What is a foreign key vs. a key here?

### 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>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
Joins

**Product**(`PName`, `Price`, `Category`, `Manufacturer`)

**Company**(`CName`, `StockPrice`, `Country`)

Ex: 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

\[ \text{Product}(\text{PName}, \text{Price}, \text{Category}, \text{Manufacturer}) \]
\[ \text{Company}(\text{CName}, \text{StockPrice}, \text{Country}) \]

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

\[
\begin{align*}
\text{SELECT} & \quad \text{PName, Price} \\
\text{FROM} & \quad \text{Product, Company} \\
\text{WHERE} & \quad \text{Manufacturer} = \text{CName} \\
& \quad \text{AND Country='Japan'} \\
& \quad \text{AND Price} \leq 200
\end{align*}
\]

A join between tables returns all unique combinations of their tuples which meet some specified join condition.
Joins

Several equivalent ways to write a basic join in SQL:

<table>
<thead>
<tr>
<th>SELECT</th>
<th>PName, Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Product, Company</td>
</tr>
<tr>
<td>WHERE</td>
<td>Manufacturer = CName AND Country='Japan' AND Price &lt;= 200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SELECT</th>
<th>PName, Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Product</td>
</tr>
<tr>
<td>JOIN</td>
<td>Company ON Manufacturer = Cname AND Country='Japan'</td>
</tr>
<tr>
<td>WHERE</td>
<td>Price &lt;= 200</td>
</tr>
</tbody>
</table>

A few more later on...
### Joins

#### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

#### Company

<table>
<thead>
<tr>
<th>Cname</th>
<th>Stock</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GWorks</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
```
Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)
Company(name, address)

SELECT DISTINCT name, address
FROM Person, Company
WHERE worksfor = name

Which “address” does this refer to?
Which “name”s??
Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)
Company(name, address)

SELECT DISTINCT Person.name, Person.address
FROM Person, Company
WHERE Person.worksfor = Company.name

SELECT DISTINCT p.name, p.address
FROM Person p, Company c
WHERE p.worksfor = c.name

Both equivalent ways to resolve variable ambiguity
Meaning (Semantics) of SQL Queries

```
SELECT  x_1.a_1, x_1.a_2, ..., x_n.a_k
FROM     R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE    Conditions(x_1,..., x_n)
```

```
Answer = {}
for x_1 in R_1 do
    for x_2 in R_2 do
        ...
        for x_n in R_n do
            if Conditions(x_1,..., x_n)
                then Answer = Answer ∪ {(x_1.a_1, x_1.a_2, ..., x_n.a_k)}

return Answer
```

Almost never the fastest way to compute it!

Note: this is a multiset union
An example of SQL semantics

```sql
SELECT R.A
FROM R, S
WHERE R.A = S.B
```

Output:

```
A 3 3
```

Cross Product:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Apply Selections / Conditions:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
Note the **semantics** of a join

1. Take **cross product**: 
   \[ X = R \times S \]

2. Apply **selections / conditions**: 
   \[ Y = \{(r, s) \in X \mid r.A = s.B\} \]

3. Apply **projections** to get final output: 
   \[ Z = (y.A, ) \text{ for } y \in Y \]

Recall: Cross product \((A \times B)\) is the set of all unique tuples in \(A\),\(B\)

Ex: \(\{a,b,c\} \times \{1,2\} = \{(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)\}\)

= Filtering!

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on...)
Note: we say “semantics” not “execution order”

• The preceding slides show what a join means

• Not actually how the DBMS executes it under the covers