CSE544
Data Management

Lectures 2: SQL
Announcements

• Lecture recordings are on zoom
• Wednesday, 1/10: review 1 is due
• Monday, 1/15: holiday, no class
• Wednesday, 1/17: canceled
• Friday, 1/19: makeup lecture, CSE2-371
• Also Friday, 1/19: review 2 is due
Recap

Relational model:

• Data is stored in flat relations
• No prescription of the physical storage
• Access to the data through high-level declarative language
SQL
SQL

• Introduced in the late 70s
• Standard has been continuously evolving into a huge language
• SQL systems support various subsets
• We will study a core supported by all systems; this is all you need
SQL

Two parts:

• Data Definition Language (DDL):
  – CREATE TABLE, …
  – You will mostly read on your own

• Data Manipulation Language (DML)
  – SELECT-FROM-WHERE
  – INSERT/DELETE/UPDATE
Relational Data Model

Supplier(sno, sname, scity, sstate)

Supply(sno, pno, qty, price)

Part(pno, pname, psize, pcolor)
Relational Data Model

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)
Relational Data Model

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)

SQL
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)

CREATE TABLE
SUPPLIER(sno int,
    sname text,
    scity text,
    sstate text);
CREATE TABLE Supplier(sno, sname, scity, sstate);
CREATE TABLE Supply(sno, pno, qty, price);
CREATE TABLE Part(pno, pname, psize, pcolor);
CREATE TABLE 
  Supplier(sno, sname, scity, sstate);
CREATE TABLE 
  Supply(sno, pno, qty, price);
CREATE TABLE 
  Part(pno, pname, psize, pcolor);
CREATE TABLE Supplier(sno int primary key, sname text, scity text, sstate text);

CREATE TABLE Part(pno int primary key, pname text, psize int, pcolor text);

CREATE TABLE Supply(sno int, pno int, qty int, price int);
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)

SQL

CREATE TABLE SUPPLIER(sno int primary key, sname text)
CREATE TABLE Part(pno int primary key, pname text, psize int, pcolor text)
CREATE TABLE Supply(sno int references Supplier, pno int references Part, qty int, price int)
CREATE TABLE SUPPLIER (sno int primary key, sname text, scity text, sstate text);

CREATE TABLE Part (pno int primary key, pname text, psize int, pcolor text);

CREATE TABLE Supply (sno int references Supplier, pno int references Part, qty int, price int, primary key (sno, pno));
SQL

INSERT INTO Supplier VALUES
(11,'ACME','Seattle','WA'),
(12,'Walmart','Portland','OR'),
(13,'Walmart','Seattle','WA');
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

SQL

SELECT ...columns...
FROM ...tables...
WHERE ...condition...
SQL

SELECT sname
FROM Supplier
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)

```
SELECT sname
FROM Supplier
```

<table>
<thead>
<tr>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
</tr>
<tr>
<td>Walmart</td>
</tr>
<tr>
<td>Costco</td>
</tr>
<tr>
<td>Walmart</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
SQL

```
SELECT sname
FROM Supplier
```

This is a bag

<table>
<thead>
<tr>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
</tr>
<tr>
<td>Walmart</td>
</tr>
<tr>
<td>Costco</td>
</tr>
<tr>
<td>Walmart</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Select distinct sname from Supplier

<table>
<thead>
<tr>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
</tr>
<tr>
<td>Walmart</td>
</tr>
<tr>
<td>Costco</td>
</tr>
<tr>
<td>Walmart</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
SQL

```
SELECT DISTINCT sname
FROM Supplier
```

Remove duplicates

Now it’s a set

<table>
<thead>
<tr>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
</tr>
<tr>
<td>Walmart</td>
</tr>
<tr>
<td>Costco</td>
</tr>
<tr>
<td>…</td>
</tr>
</tbody>
</table>
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

SQL

SELECT sname
FROM Supplier

SELECT sname, scity
FROM Supplier

SELECT *
FROM Supplier

What do these queries return?
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)

**SQL: WHERE**

```
SELECT *
FROM Supplier
WHERE sstate = 'WA'
```

Returns only suppliers in Washington State
Discussion

• Keywords, table/attribute names are case insensitive:
  – SELECT, select, sElEcT
  – Supplier, SUPPLIER, …

• Strings are case sensitive:
  – ‘WA’ different from ‘wa’

• WHERE conditions can use complex predicates
  – WHERE psize>15 and pcolor=‘red’ or pcolor=‘blue’

• SQL has lots of built-in predicates; look them up!
  – WHERE surname LIKE ‘%mart%’
SQL: Joins

Find all suppliers in ‘WA’ that supply ‘red’ parts
SQL: Joins

Find all suppliers in ‘WA’ that supply ‘red’ parts

```sql
SELECT *
FROM Supplier s
JOIN Supply s2 ON s.sno = s2.sno
JOIN Part p ON s2.pno = p.pno
WHERE s.sstate = 'WA' AND p.pcolor = 'red'
```
SQL: Joins

Find all suppliers in ‘WA’ that supply ‘red’ parts

```
SELECT            FROM Supplier x, Supply y, Part z
WHERE
```

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)
SQL: Joins

Find all suppliers in ‘WA’ that supply ‘red’ parts

```
SELECT   Supplier x, Supply y, Part z
FROM      Supplier x, Supply y, Part z
WHERE     x.sno = y.sno
    and y.pno = z.pno
```
SQL: Joins

Find all suppliers in ‘WA’ that supply ‘red’ parts

```
SELECT Supplier x, Supply y, Part z
FROM   Supplier x, Supply y, Part z
WHERE  x.sno = y.sno
       and y.pno = z.pno
       and x.sstate = 'WA'
       and z.pcolor = 'red';
```
SQL: Joins

Find all suppliers in ‘WA’ that supply ‘red’ parts

```
SELECT DISTINCT  z.pno, z.pname, x.scity
FROM Supplier x, Supply y, Part z
WHERE  x.sno = y.sno
   and y.pno = z.pno
   and x.sstate = 'WA'
   and z.pcolor = 'red';
```
SQL: Joins

Find all suppliers in ‘WA’ that supply ‘red’ parts

```
SELECT DISTINCT z.pno, z.pname, x.scity
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno
    and y.pno = z.pno
    and x.sstate = 'WA'
    and z.pcolor = 'red';
```

What happens if we don’t include these conditions?
Operations

- **Selection/filter**: return a subset of the rows:
  - `SELECT * FROM Supplier`  
  `WHERE scity = 'Seattle'`

- **Projection**: return subset of the columns:
  - `SELECT DISTINCT scity FROM Supplier`

- **Join**: refers to combining two or more tables
  - `SELECT * FROM Supplier, Supply, Part ...`
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y.pno
FROM        Supplier x, Supply y
WHERE       x.scity = 'Seattle'
            and x.scity = 'Portland'
            and x.sno = y.sno
```
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT y.pno
FROM Supplier x, Supply y
WHERE x.scity = 'Seattle'
  and x.scity = 'Portland'
  and x.sno = y.sno
```

This doesn’t work… Why?
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

SELECT DISTINCT y.pno
FROM Supplier x, Supply y
WHERE (x.scity = 'Seattle'
      or x.scity = 'Portland')
      and x.sno = y.sno

Does this work?
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT y.pno
FROM Supplier x, Supply y
WHERE (x.scity = 'Seattle'
    or x.scity = 'Portland')
    and x.sno = y.sno
```
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y1.pno
FROM       Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE      x1.scity = ‘Seattle’
    and x1.sno = y1.sno
    and x2.scity = ‘Portland’
    and x2.sno = y2.sno
    and y1.pno = y2.pno
```

Need TWO Suppliers and TWO Supplies
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT y1.pno
FROM   Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE  x1.scity = 'Seattle'
       and x1.sno = y1.sno
       and x2.scity = 'Portland'
       and x2.sno = y2.sno
       and y1.pno = y2.pno
```

Need TWO Suppliers and TWO Supplies

one in Seattle
the other in Portland
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

\[
\text{SELECT DISTINCT } y1.pno \\
\text{FROM } \text{Supplier x1, Supplier x2, Supply y1, Supply y2} \\
\text{WHERE } \text{x1.scity = ‘Seattle’} \\
\text{and x1.sno = y1.sno} \\
\text{and x2.scity = ‘Portland’} \\
\text{and x2.sno = y2.sno} \\
\text{and y1.pno = y2.pno}
\]
Discussion

SELECT-FROM-WHERE

• FROM clause: cartesian product
• WHERE clause: filter out
• SELECT clause: says what to return

The results can be described as a set of nested loops. Next.
Nested-Loop Semantics of SQL

```
SELECT a_1, a_2, ..., a_k
FROM  R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE Conditions
```
Nested-Loop Semantics of SQL

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

Answer = {}
for x_1 in R_1 do
    for x_2 in R_2 do
        ...
        for x_n in R_n do
            if Conditions
                then Answer = Answer \cup \{(a_1,\ldots,a_k)\}
return Answer
```
Nested-Loop Semantics of SQL

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

Answer = {}
for $x_1$ in $R_1$ do
    for $x_2$ in $R_2$ do
        .....  
        for $x_n$ in $R_n$ do
            if Conditions
                then Answer = Answer $\cup \{(a_1,\ldots,a_k)\}$
        return Answer

This SEMANTICS!
It is NOT how the engine computes the query!
Discussion

Data independence:
• SQL engines do NOT compute the query using nested loop semantics
• Instead they choose between a variety of execution plans

How would you execute it?

```
SELECT DISTINCT z.pno, z.pname, x.scity
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno
  and y.pno = z.pno
  and x.sstate = 'WA'
  and z.pcolor = 'red';
```
NULLs in SQL

• A NULL value means missing, or unknown, or undefined, or inapplicable
Boolean predicate:
- Atomic: Expr1 op Expr2
- AND / OR / NOT

price < 100 and (pcolor='red' or psize=2)

How do we compute the predicate when values are NULL?
Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - False or True when both A, B are not null
  - Unknown otherwise
- AND, OR, NOT are min, max.
- Return only tuples whose condition is True
Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - False or True when both A, B are not null
  - Unknown otherwise
- AND, OR, NOT are \textbf{min}, \textbf{max}.
- Return only tuples whose condition is \textbf{True}

```sql
select *
from Part
where price < 100
  and (psize=2 or pcolor='red')
```
Three-Valued Logic

• False=0, Unknown=0.5, True=1
• A op B is
  – False or True when both A, B are not null
  – Unknown otherwise
• AND, OR, NOT are min, max.
• Return only tuples whose condition is True

select *
from Part
where price < 100
and (psize=2 or pcolor='red')
### Three-Valued Logic

- **False** = 0, **Unknown** = 0.5, **True** = 1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```sql
select *
from Part
where price < 100
  and (psize=2 or pcolor='red')
```

<table>
<thead>
<tr>
<th>pno</th>
<th>pname</th>
<th>price</th>
<th>psize</th>
<th>pcolor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>iPad</td>
<td>500</td>
<td>13</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>Scooter</td>
<td>99</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>Charger</td>
<td>NULL</td>
<td>NULL</td>
<td>red</td>
</tr>
<tr>
<td>1</td>
<td>iPad</td>
<td>50</td>
<td>2</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Three-Valued Logic

- **False**=0, **Unknown**=0.5, **True**=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```sql
select * 
from Part 
where price < 100 
and (psize=2 or pcolor='red')
```
Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - False or True when both A, B are not null
  - Unknown otherwise
- AND, OR, NOT are min, max.
- Return only tuples whose condition is True

```
select *
from Part
where price < 100
  and (psize=2 or pcolor='red')
```
Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - False or True when both A, B are not null
  - Unknown otherwise
- AND, OR, NOT are min, max.
- Return only tuples whose condition is True

-- problem: (A or not(A)) ≠ true
-- does NOT return all Products
select *
from Product
where (price <= 100) or (price > 100)
Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - False or True when both A, B are not null
  - Unknown otherwise
- AND, OR, NOT are min, max.
- Return only tuples whose condition is True

-- problem: (A or not(A)) ≠ true
-- does NOT return all Products
```sql
select *
from Product
where (price <= 100) or (price > 100)
```

-- returns ALL Products
```sql
select *
from Product
where (price <= 100) or (price > 100)
  or isNull(price)
```
Discussion

• So far we have seen only SELECT-FROM-WHERE queries
• Most data analysis requires some form of aggregation
• An aggregate operator takes a set of values and returns a single value
  – sum, min, max, avg, count
SELECT count(*)
FROM Part
More SQL: Aggregates

```sql
SELECT count(*)
FROM Part
```

```sql
SELECT x.scity, avg(psize)
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
GROUP BY x.scity
```
More SQL: Aggregates

```
SELECT count(*)
FROM Part

SELECT x.scity, avg(psize)
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
GROUP BY x.scity

SELECT x.scity, avg(psize)
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
GROUP BY x.scity
HAVING count(*) > 200
```
Discussion

• SQL Aggregates = simple data analytics

• Semantics:
  1. FROM-WHERE (nested-loop semantics)
  2. Group answers by GROUP BY attrs
  3. Apply HAVING predicates on groups
  4. Apply SELECT aggregates on groups

• Aggregate functions:
  – count, sum, min, max, avg

• DISTINCT same as GROUP BY
Outer Joins

• A join returns only those outputs that have a tuple from each of the input tables

• Sometimes we want to include tuples from one table without a match from the other table:

Outer Join
Outer joins

Retrieve all product names, categories, and stores where they were purchased. Include products that never sold.
Outer joins

Retrieve all product names, categories, and stores where they were purchased. Include products that never sold.

```
SELECT x.name, x.category, y.store 
FROM   Product x, Purchase y 
WHERE  x.name = y.prodName
```
Product (name, category)
Purchase (prodName, store)

Outer joins

Retrieve all product names, categories, and stores where they were purchased.
Include products that never sold

```
SELECT x.name, x.category, y.store
FROM Product x, Purchase y
WHERE x.name = y.prodName
```

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>Category</strong></td>
<td><strong>ProdName</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Camera</td>
</tr>
</tbody>
</table>
Outer joins

Retrieve all product names, categories, and stores where they were purchased. Include products that never sold.

```sql
SELECT x.name, x.category, y.store
FROM Product x, Purchase y
WHERE x.name = y.prodName
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Category</td>
<td>ProdName</td>
</tr>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Camera</td>
</tr>
</tbody>
</table>

prodName is foreign key

missing
Outer joins

Retrieve all product names, categories, and stores where they were purchased. Include products that never sold.

```
SELECT x.name, x.category, y.store
FROM Product x LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Category</td>
<td>ProdName</td>
</tr>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Camera</td>
</tr>
</tbody>
</table>

prodName is foreign Key

Now it's present
Left Outer Join (Details)

from R left outer join S on C1 where C2

1. Compute cross product R×S
2. Filter on C1
3. Add all R records without a match
4. Filter on C2
Left Outer Join (Details)

```
select ...
from   R left outer join S on C1
where   C2
```

```python
Tmp = {}
for x in R do  // left outer join using C1
    for y in S do
        if C1 then Tmp = Tmp ∪ {(x,y)}
    for x in R do
        if not (x in Tmp) then Tmp = Tmp ∪ {(x,NULL)}

Answer = {}  // apply condition C2
for (x,y) in Tmp if C2 then Answer = Answer ∪ {(x,y)}
return Answer
```
ON v.s. WHERE

• Outer join condition in the **ON** clause
• Different from the **WHERE** clause
• Compare:

```
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
AND y.price < 10
```

```
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
WHERE y.price < 10
```
Product(name, category)
Purchase(prodName, store, price)

ON v.s. WHERE

• Outer join condition in the ON clause
• Different from the WHERE clause
• Compare:

```sql
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
AND y.price < 10
```

Includes products that were never purchased with price < 10

```sql
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
WHERE y.price < 10
```

prodName is foreign Key

**Product**

**Purchase**
ON v.s. WHERE

- Outer join condition in the **ON** clause
- Different from the **WHERE** clause
- Compare:

```sql
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
AND y.price < 10
```

Includes products that were never purchased with price < 10

```sql
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
WHERE y.price < 10
```

Includes products that were never purchased, then checks price < 10

Product (name, category)
Purchase (prodName, store, price)
ON v.s. WHERE

- Outer join condition in the **ON** clause
- Different from the **WHERE** clause
- Compare:

```sql
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
AND y.price < 10
```

Includes products that were never purchased with price < 10

```sql
SELECT x.name, y.store
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.name = y.prodName
WHERE y.price < 10
```

Includes products that were never purchased, then checks price <10

Same as inner join!
Joins

- **Inner join** = includes only matching tuples (i.e. regular join)
- **Left outer join** = includes everything from the left
- **Right outer join** = includes everything from the right
- **Full outer join** = includes everything