Today

- Join Syntax (cont)
- Aggregation
- Set Ops

Assignments

- Quiz 4 Tues
- HW4 out
- Exam 1 Thurs
Basic SQL Queries: Example Tables

**account**

<table>
<thead>
<tr>
<th>number</th>
<th>owner</th>
<th>balance</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>J. Smith</td>
<td>1000.00</td>
<td>checking</td>
</tr>
<tr>
<td>102</td>
<td>W. Wei</td>
<td>2000.00</td>
<td>checking</td>
</tr>
<tr>
<td>103</td>
<td>J. Smith</td>
<td>5000.00</td>
<td>savings</td>
</tr>
<tr>
<td>104</td>
<td>M. Jones</td>
<td>1000.00</td>
<td>checking</td>
</tr>
<tr>
<td>105</td>
<td>H. Martin</td>
<td>10000.00</td>
<td>checking</td>
</tr>
</tbody>
</table>

**deposit**

<table>
<thead>
<tr>
<th>account</th>
<th>transaction_id</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>1</td>
<td>10/22/11</td>
<td>500.00</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>10/29/11</td>
<td>200.00</td>
</tr>
<tr>
<td>104</td>
<td>3</td>
<td>10/29/11</td>
<td>1000.00</td>
</tr>
<tr>
<td>105</td>
<td>4</td>
<td>11/2/11</td>
<td>10000.00</td>
</tr>
</tbody>
</table>

**check**

<table>
<thead>
<tr>
<th>account</th>
<th>check_number</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>924</td>
<td>10/23/11</td>
<td>125.00</td>
</tr>
<tr>
<td>101</td>
<td>925</td>
<td>10/24/11</td>
<td>23.98</td>
</tr>
</tbody>
</table>

**FK's:**

- deposit.account → account.number
- check.account → account.number
Exercise: Rewire the following queries using JOIN syntax

Query 1:

```sql
SELECT a.name, a.balance
FROM account a, deposit d
WHERE a.number = d.account AND a.balance > 1000;
```

- In this case: ON a.number = d.account

Query 2:

```sql
SELECT a1.owner, a1.number, a1.balance, a2.number, a2.balance
FROM account a1, account a2
WHERE a1.owner = a2.owner AND a1.balance > a2.balance
```

- In this case: USING (owner)
Multiple Joins

\[
\begin{align*}
\text{SELECT} & \; * \\
\text{FROM} & \; \text{deposit d JOIN check c USING (account)} \\
& \; \text{JOIN account a ON c.account = a.number}
\end{align*}
\]

Q: Rewrite this query using only “comma” joins ... 

\[
\begin{align*}
\text{SELECT} & \; * \\
\text{FROM} & \; \text{deposit d, check c, account a} \\
& \; \text{WHERE} \; \text{d.check = c.check AND c.account = a.number}
\end{align*}
\]

Natural Joins

\[
\begin{align*}
\text{SELECT} & \; * \\
\text{FROM} & \; \text{deposit NATURAL JOIN check}
\end{align*}
\]

- joins on same-named attributes between two tables
- only outputs unique attribute names

Cross Joins

\[
\begin{align*}
\text{SELECT} & \; * \\
\text{FROM} & \; \text{deposit CROSS JOIN check}
\end{align*}
\]

- this is the same as:

\[
\begin{align*}
\text{SELECT} & \; * \\
\text{FROM} & \; \text{deposit, check}
\end{align*}
\]

We’ll look at outer joins later ...
More SQL: Aggregation (SELECT)

SQL provides 5 aggregate operators

- COUNT, SUM, MIN, MAX, AVG

```sql
SELECT MIN(balance), MAX(balance), AVG(balance)
FROM account
WHERE type = 'checking';
```

<table>
<thead>
<tr>
<th>MIN(balance)</th>
<th>MAX(balance)</th>
<th>AVG(balance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10000</td>
<td>3500</td>
</tr>
</tbody>
</table>

```sql
SELECT SUM(balance), COUNT(balance)
FROM account
WHERE type = 'checking';
```

<table>
<thead>
<tr>
<th>SUM(balance)</th>
<th>COUNT(balance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>14000</td>
<td>4</td>
</tr>
</tbody>
</table>

If an aggregate operator appears in a SELECT clause ...

- all entries in the select clause must be aggregate operators
- unless the query includes a GROUP BY clause (more later)

For example:

```sql
SELECT owner, AVG(balance)
FROM account;
```

- this query is not well formed ... but MySQL allows it
Q: What does MySQL do?

- calculate the average for each owner? NO!!!
- pick an owner and then average balance across all owners YES!!!

Note: This is confusing. So beware!

The `COUNT` aggregate can take multiple attributes

```sql
SELECT COUNT(*)
FROM account;
```

Q: What does this return? ... number of rows in account

```sql
SELECT COUNT(DISTINCT balance, type)
FROM account;
```

- This is OK ... returns number of unique pairs of (balance, type) values

```sql
SELECT COUNT(balance, type)
FROM account;
```

- This is not a well-formed query ... even in MySQL
- Better to use COUNT(*) unless using DISTINCT ... why?
Using DISTINCT with aggregates

Q: Can you guess what the difference is between these 2 queries?

```sql
SELECT SUM(balance)
FROM account;

SELECT SUM(DISTINCT balance)
FROM account;
```

- sum computed over all balances vs. only distinct balances
- first query returns 19000
- second returns 18000

Q: Can you guess what the difference is between these 2?

```sql
SELECT MIN(balance)
FROM account;

SELECT MIN(DISTINCT balance)
FROM account;
```

- Nothing!
More SQL: SELECT expressions and comparators

Arithmetic expressions can be used in the SELECT clause

```sql
SELECT a.owner, d.date, (a.balance + d.amount) AS new_bal
FROM account a, deposit d
WHERE a.number = d.account
```

MySQL also allows comparison operators

```sql
SELECT owner, balance > 5000 as large_balance
FROM account
WHERE type = 'checking'
```

<table>
<thead>
<tr>
<th>owner</th>
<th>large_balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>J. Smith</td>
<td>0</td>
</tr>
<tr>
<td>W. Wei</td>
<td>0</td>
</tr>
<tr>
<td>H. Martin</td>
<td>1</td>
</tr>
<tr>
<td>M. Jones</td>
<td>0</td>
</tr>
</tbody>
</table>

Query Answer:
• SQL largely ignores NULL values when computing aggregates

   – Consider the following table

   Cars
<table>
<thead>
<tr>
<th>id</th>
<th>retail_price</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
<td>economy</td>
</tr>
<tr>
<td>2</td>
<td>NULL</td>
<td>sports</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>luxury</td>
</tr>
</tbody>
</table>

   – The query:

   ```
   SELECT AVG(retail_price)
   FROM Cars;
   ```

   – Returns:

<table>
<thead>
<tr>
<th>AVG(retail_price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7500</td>
</tr>
</tbody>
</table>

   – And the query:

   ```
   SELECT COUNT(retail_price)
   FROM Cars;
   ```

   – Returns:

<table>
<thead>
<tr>
<th>COUNT(retail_price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
- Whereas the query:

```
SELECT COUNT(*)
FROM Cars;
```

- Returns:

| COUNT(*) | 3 |
Union, intersection, difference

Set-based operators

- ∪  union
- ∩  intersect
- \ or −  difference

Consider the two sets:

- $S_1 = \{1, 3, 5, 7\}$
- $S_2 = \{1, 2, 3, 4\}$

Q: What do these return?

- $S_1 \cap S_2 = \{1, 3\}$
- $S_1 \cup S_2 = \{1, 2, 3, 4, 5, 6, 7\}$
- $S_1 - S_2 = \{5, 6\}$
- $S_2 - S_1 = \{2, 4\}$

These operators can only be used with "union compatible" relations

- Two relations are union compatible if
  - they have the same arity
  - the corresponding attributes have the same domains

Q: Are these relations union compatible?

- Checking(c_num: int, c_owner: varchar(80), c_balance: float)
- Savings(s_num: int, s_owner: varchar(80), s_balance: float)

Q: What changes would make them no longer union compatible?
The **UNION** keyword in SQL

```
(SELECT owner
 FROM account
 WHERE balance >= 10000)
UNION
(SELECT owner
 FROM account
 WHERE balance <= 100);
```

**Note we can add literals to help distinguish results**

```
(SELECT owner, 'high balance' as at_risk_type
 FROM account
 WHERE balance >= 10000)
UNION
(SELECT owner, 'low balance' as at_risk_type
 FROM account
 WHERE balance <= 100);
```

<table>
<thead>
<tr>
<th>owner</th>
<th>at_risk_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>H. Martin</td>
<td>high balance</td>
</tr>
<tr>
<td>J. Smith</td>
<td>low balance</td>
</tr>
</tbody>
</table>