General Overview - Rel. Model

- Formal query languages
  - rel algebra and calculi
- Commercial query languages
  - SQL
  - Datalog
  - LINQ
  - Xquery
  - Pig (Hadoop)

Relational Languages

- A major strength of the relational model: supports simple, powerful querying of data.
- User only needs to specify the answer that they want, not how to compute it.
- The DBMS is responsible for efficient evaluation of the query.
  - Query optimizer: re-orders operations and generates query plan
Relational Languages

- Standardized **DML/DDL**
  - **DML** → Data Manipulation Language
  - **DDL** → Data Definition Language
- Also includes:
  - View definition
  - Integrity & Referential Constraints
  - Transactions

History

- Originally “SEQUEL” from IBM’s System R prototype.
  - Structured English Query Language
  - Adopted by Oracle in the 1970s.
  - Structured Query Language

History

- Current standard is **SQL:2011**
  - **SQL:2008** → TRUNCATE, Fancy ORDER
  - **SQL:2003** → XML, windows, sequences, auto-generated IDs.
  - **SQL:1999** → Regex, triggers, OO
- Most DBMSs at least support **SQL-92**
- System Comparison:
  - [http://troels.arvin.dk/db/rdbms/](http://troels.arvin.dk/db/rdbms/)
Overview

- **DML**
  - select, from, where, renaming
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries
- Other parts: **DDL**, embedded SQL, auth etc

Intro to SQL

- **SELECT**
- **FROM**
- **WHERE**
- Formal Semantics

Example Database

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>ACCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cname</strong></td>
<td><strong>acctno</strong></td>
</tr>
<tr>
<td>Georg Hegel</td>
<td>A-123</td>
</tr>
<tr>
<td>Friedrich Engels</td>
<td>A-456</td>
</tr>
<tr>
<td>Max Stirner</td>
<td>A-789</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>acctno</th>
<th>bname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>A-789</td>
<td>Downtown</td>
<td>2000</td>
</tr>
<tr>
<td>A-123</td>
<td>Perry</td>
<td>1500</td>
</tr>
<tr>
<td>A-456</td>
<td>Downtown</td>
<td>1000</td>
</tr>
</tbody>
</table>
### First SQL Example

**SELECT bname**

**FROM account**

**WHERE** amt > 1000

**Similar to…**

\[ \pi_{bname} (\sigma_{amt>1000}(\text{account})) \]

**But not quite…**

\[ bname \]

- Downtown
- Redwood
- Perry

<table>
<thead>
<tr>
<th>bname</th>
<th>lno</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-170</td>
<td>3000</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
</tr>
<tr>
<td>Perry</td>
<td>L-260</td>
<td>1700</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-450</td>
<td>3000</td>
</tr>
</tbody>
</table>

---

### First SQL Example

**SELECT DISTINCT bname**

**FROM loan**

**WHERE** amt > 1000

**Now we get the same result as the relational algebra**

\[ \pi_{bname} (\sigma_{amt>1000}(\text{account})) \]

<table>
<thead>
<tr>
<th>bname</th>
<th>lno</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-170</td>
<td>3000</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
</tr>
<tr>
<td>Perry</td>
<td>L-260</td>
<td>1700</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-450</td>
<td>3000</td>
</tr>
</tbody>
</table>

**Why preserve duplicates?**

- Eliminating them is costly
- Users often don’t care.

---

### Multi-Relation Queries

**SELECT cname, amt**

**FROM customer, account**

**WHERE**

\[ \text{customer.acctno} = \text{account.acctno} \]

**AND** account.amt > 1000

**Same as**

\[ \pi_{cname, amt}(\sigma_{\text{amt}>1000}(\text{customer} \bowtie \text{account})) \]

<table>
<thead>
<tr>
<th>cname</th>
<th>acctno</th>
</tr>
</thead>
<tbody>
<tr>
<td>Georg Hegel</td>
<td>A-123</td>
</tr>
<tr>
<td>Friedrich Engels</td>
<td>A-456</td>
</tr>
<tr>
<td>Max Stirner</td>
<td>A-789</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>acctno</th>
<th>bname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>A-789</td>
<td>Downtown</td>
<td>2000</td>
</tr>
<tr>
<td>A-122</td>
<td>Perry</td>
<td>1500</td>
</tr>
<tr>
<td>A-456</td>
<td>Downtown</td>
<td>1000</td>
</tr>
</tbody>
</table>
Basic SQL Query Grammar

- **Relation-List**: A list of relation names
- **Target-List**: A list of attributes from the tables referenced in relation-list
- **Qualification**: Comparison of attributes or constants using operators =, ≠, <, >, ≤, and ≥.

SELECT Clause

- Use `*` to get all attributes
  
  ```sql
  SELECT * FROM account
  ```

- Use `DISTINCT` to eliminate dupes
  
  ```sql
  SELECT DISTINCT bname FROM account
  ```

- Target list can include expressions
  
  ```sql
  SELECT bname, amt*1.05 FROM account
  ```

FROM Clause

- Binds tuples to variable names
  
  ```sql
  SELECT * FROM depositor, account
  WHERE depositor.acctno = account.acctno
  ```

- Define what kind of join to use
  
  ```sql
  SELECT depositor.*, account.amt
  FROM depositor LEFT OUTER JOIN account
  WHERE depositor.acctno = account.acctno
  ```
**WHERE Clause**

- Complex expressions using **AND**, **OR**, and **NOT**

```sql
SELECT * FROM account
WHERE amt > 1000
AND (bname = "Downtown" OR
NOT bname = "Perry")
```

- Special operators **BETWEEN**, **IN**:

```sql
SELECT * FROM account
WHERE (amt BETWEEN 100 AND 200)
AND bname IN ("Leon", "Perry")
```

---

**Renaming**

- The **AS** keyword can also be used to rename tables and columns in **SELECT** queries.
- Allows you to target a specific table instance when you reference the same table multiple times.

---

**Renaming – Table Variables**

- Find customers with an account in the "Downtown" branch with more than $100.

```sql
SELECT customer.cname, account.amt
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.bname = "Downtown"
AND account.amt > 1000
```
Renaming – Table Variables

• Find customers with an account in the “Downtown” branch with more than $100.

```sql
SELECT C.cname, A.amt AS cmnt
FROM customer AS C, account AS A
WHERE C.acctno = A.acctno
AND A.bname = "Downtown"
AND A.amt > 1000
```

Renaming – Self-Join

• Find all unique accounts that are open at more than one branch.

<table>
<thead>
<tr>
<th>acctno</th>
<th>bname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>A-789</td>
<td>Downtown</td>
<td>2000</td>
</tr>
<tr>
<td>A-123</td>
<td>Perry</td>
<td>1500</td>
</tr>
<tr>
<td>A-456</td>
<td>Downtown</td>
<td>1000</td>
</tr>
</tbody>
</table>

```sql
SELECT DISTINCT a1.acctno
FROM account AS a1, account AS a2
WHERE a1.acctno = a2.acctno
AND a1.bname != a2.bname
```

Renaming – Theta-Join

• Find all unique accounts that are open at more than one branch and have an amount greater than $1600.

<table>
<thead>
<tr>
<th>acctno</th>
<th>bname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>A-789</td>
<td>Downtown</td>
<td>2000</td>
</tr>
<tr>
<td>A-123</td>
<td>Perry</td>
<td>1500</td>
</tr>
<tr>
<td>A-456</td>
<td>Downtown</td>
<td>1000</td>
</tr>
</tbody>
</table>

```sql
SELECT DISTINCT a1.acctno
FROM account AS a1, account AS a2
WHERE a1.acctno = a2.acctno
AND a1.bname != a2.bname
AND a1.amt > 1600
```
Formal Semantics of SQL

- To express SQL, must extend to a bag algebra:
  - A bag is like a set, but can have duplicates
  - Example: \{4, 5, 4, 6\}

<table>
<thead>
<tr>
<th>acctno</th>
<th>bname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>A-789</td>
<td>Downtown</td>
<td>2000</td>
</tr>
<tr>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>A-456</td>
<td>Downtown</td>
<td>1000</td>
</tr>
</tbody>
</table>

Formal Semantics of SQL

- A SQL query is defined in terms of the following evaluation strategy:
  1. Execute **FROM** clause
     Compute cross-product of all tables
  2. Execute **WHERE** clause
     Check conditions, discard tuples
  3. Execute **SELECT** clause
     Delete unwanted columns.

• Probably the worst way to compute!

Execution Example

- Find the students that got a “D” grade in any course.

```sql
SELECT S.name, E.cid
FROM students AS S, enrolled AS E
WHERE S.sid = E.sid AND E.grade = "D"
```
Step 1 – Cross Product

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Faloutsos</td>
<td>christos@cs</td>
<td>45</td>
<td>1.8</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>21</td>
<td>3.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Pilates101</td>
<td>C</td>
</tr>
<tr>
<td>53832</td>
<td>Reggae203</td>
<td>D</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Massage105</td>
<td>D</td>
</tr>
</tbody>
</table>

Step 2 – Discard Tuples

```
SELECT S.name, E.cid
FROM Students AS S, Enrolled AS E
WHERE S.sid = E.sid AND E.grade = "D"
```

Step 3 – Discard Columns

```
SELECT S.name, E.cid
FROM Students AS S, Enrolled AS E
WHERE S.sid = E.sid AND E.grade = "D"
```
More SQL

- INSERT
- UPDATE
- DELETE
- TRUNCATE

INSERT

- Provide target table, columns, and values for new tuples:

  ```sql
  INSERT INTO account
  (acctno, bname, amt)
  VALUES
  ("A-999", "Pittsburgh", 1000);
  ```

- Short-hand version:

  ```sql
  INSERT INTO account VALUES
  ("A-999", "Pittsburgh", 1000);
  ```

UPDATE

- **UPDATE** must list what columns to update and their new values (separated by commas).
- Can only update one table at a time.
- **WHERE** clause allows query to target multiple tuples at a time.

  ```sql
  UPDATE account
  SET bname = "Compton",
      amt = amt + 100
  WHERE acctno = "A-999"
  AND bname = "Pittsburgh"
  ```
DELETE

- Similar to single-table SELECT statements.
- The WHERE clause specifies which tuples will deleted from the target table.
- The delete may cascade to children tables.

```
DELETE FROM account WHERE amt < 0
```

TRUNCATE

- Remove all tuples from a table.
- This is usually faster than DELETE, unless it needs to check foreign key constraints.

```
TRUNCATE account
```

Even More SQL

- NULLs
- String Operations
- Output Redirection
- Set/Bag Operations
- Output Control
- Aggregates
NULLs

- The “dirty little secret” of SQL, since it can be a value for any attribute.

<table>
<thead>
<tr>
<th>bname</th>
<th>city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oakland</td>
<td>Pittsburgh</td>
<td>$2,000,000</td>
</tr>
<tr>
<td>Compton</td>
<td>Los Angeles</td>
<td>NULL</td>
</tr>
<tr>
<td>Long Beach</td>
<td>Los Angeles</td>
<td>$400,000</td>
</tr>
<tr>
<td>Harlem</td>
<td>New York</td>
<td>$1,700,000</td>
</tr>
</tbody>
</table>

- What does this mean?
  - We don’t know Compton assets?
  - Compton has no assets?

• Find all branches that have null assets.

```sql
SELECT * FROM branch WHERE assets IS NULL
```

<table>
<thead>
<tr>
<th>bname</th>
<th>city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compton</td>
<td>Los Angeles</td>
<td>NULL</td>
</tr>
</tbody>
</table>

• Find all branches that have null assets.

```sql
SELECT * FROM branch WHERE assets = NULL
```

<table>
<thead>
<tr>
<th>bname</th>
<th>city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compton</td>
<td>Los Angeles</td>
<td>NULL</td>
</tr>
</tbody>
</table>
**NULLs**

- Arithmetic operations with **NULL** values is always **NULL**.

```sql
SELECT 1+NULL AS add_null,
       1-NULL AS sub_null,
       1*NULL AS mul_null,
       1/NULL AS div_null;
```

<table>
<thead>
<tr>
<th>add_null</th>
<th>sub_null</th>
<th>mul_null</th>
<th>div_null</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**NULLs**

- Comparisons with **NULL** values varies.

```sql
SELECT true = NULL AS eq_bool,
      true != NULL AS neq_bool,
      true AND NULL AS and_bool,
      NULL = NULL AS eq_null,
      NULL IS NULL AS is_null;
```

<table>
<thead>
<tr>
<th>eq_bool</th>
<th>neq_bool</th>
<th>and_bool</th>
<th>eq_null</th>
<th>is_null</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

**String Operations**

<table>
<thead>
<tr>
<th>String Case</th>
<th>String Quotes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQL-92</strong></td>
<td>Sensitive</td>
</tr>
<tr>
<td>Postgres</td>
<td>Sensitive</td>
</tr>
<tr>
<td>MySQL</td>
<td>Insensitive</td>
</tr>
<tr>
<td>SQLite</td>
<td>Sensitive</td>
</tr>
<tr>
<td>DB2</td>
<td>Sensitive</td>
</tr>
<tr>
<td>Oracle</td>
<td>Sensitive</td>
</tr>
</tbody>
</table>

```sql
WHERE UPPER(name) = 'EURKEL'  SQL-92
WHERE name = "EURKEL"  MySQL
```
String Operations

- **LIKE** is used for string matching.
- String-matching operators
  - “%” Matches any substring (incl. empty).
  - “_” Match any one character

```sql
SELECT * FROM enrolled AS e
  WHERE e.cid LIKE 'Pilates%'
```

```sql
SELECT * FROM student AS s
  WHERE s.name LIKE '%louso'
```

String Operations

- **SQL-92** defines string functions.
  - Many DBMSs also have their own unique functions
  - Can be used in either output and predicates:

```sql
SELECT SUBSTRING(name, 0, 5) AS abbr_name
  FROM student WHERE sid = 53688
```

```sql
SELECT * FROM student AS s
  WHERE UPPER(s.name) LIKE 'FALOU%'
```

Output Redirection

- Store query results in another table:
  - Table must not already be defined.
  - Table will have the same # of columns with the same types as the input.

```sql
SELECT DISTINCT cid INTO CourseIds SQL-92
  FROM Enrolled;
```

```sql
CREATE TABLE CourseIds (MySQL
  SELECT DISTINCT cid FROM Enrolled);
```
Output Redirection

- Insert tuples from query into another table:
  - Inner `SELECT` must generate the same columns as the target table.
  - DBMSs have different options/syntax on what to do with duplicates.

```
INSERT INTO CourseIds
(SELECT DISTINCT cid FROM Enrolled);
```

Set/Bag Operations

- Set Operations:
  - `UNION`
  - `INTERSECT`
  - `EXCEPT`

- Bag Operations:
  - `UNION ALL`
  - `INTERSECT ALL`
  - `EXCEPT ALL`

Set Operations

```
(SELECT cname FROM depositor)
UNION
(SELECT cname FROM borrower)
```

- `UNION` Returns names of customers with saving accts, loans, or both.
- `INTERSECT` Returns names of customers with saving accts AND loans.
- `EXCEPT` Returns names of customers with saving accts but NOT loans.
Bag Operations

- There are \( m \) copies of \( a \) in table \( R \) and \( n \) copies of \( a \) in table \( S \).
- How many copies of \( a \) in…
  - \( R \) **UNION ALL** \( S \) \( \rightarrow m + n \)
  - \( R \) **INTERSECT ALL** \( S \) \( \rightarrow \min(m, n) \)
  - \( R \) **EXCEPT ALL** \( S \) \( \rightarrow \max(0, m-n) \)

Output Control

- **ORDER BY** <column*> [ASC|DESC]
  - Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
WHERE cid = 'Pilates105'
ORDER BY grade
```

```
SELECT sid FROM enrolled
WHERE cid = 'Pilates105'
ORDER BY grade DESC, sid ASC
```

Output Control

- **LIMIT** <count> [offset]
  - Limit the # of tuples returned in output.
  - Can set an offset to return a “range”

```
SELECT sid, name FROM Student
WHERE login LIKE '%cs%'
LIMIT 10  
First 10 rows
```

```
SELECT sid, name FROM Student
WHERE login LIKE '%cs5'
LIMIT 20 OFFSET 10  
Skip first 10 rows, Return the following 20
```
Aggregates

- Functions that return a single value from a bag of tuples:
  - \( \text{AVG}(\text{col}) \) → Return the average \( \text{col} \) value.
  - \( \text{MIN}(\text{col}) \) → Return minimum \( \text{col} \) value.
  - \( \text{MAX}(\text{col}) \) → Return maximum \( \text{col} \) value.
  - \( \text{SUM}(\text{col}) \) → Return sum of values in \( \text{col} \).
  - \( \text{COUNT}(\text{col}) \) → Return # of values for \( \text{col} \).

Aggregates

- Functions can only be used in the SELECT attribute output list.
- Get the number of students with a @cs login:

  \[
  \text{SELECT COUNT(login) AS cnt FROM student WHERE login LIKE '%@cs'}
  \]

  cnt
  12

Aggregates

- Can use multiple functions together at the same time.
- Get the number of students and their GPA that have a @cs login.

  \[
  \text{SELECT AVG(gpa), COUNT(sid) FROM student WHERE login LIKE '%@cs'}
  \]

  AVG(gpa)  COUNT(sid)
  3.25  12
Aggregates

- **COUNT, SUM, AVG** support **DISTINCT**
  - Get the number of unique students that have an @cs login.

```sql
SELECT COUNT(DISTINCT login) FROM student WHERE login LIKE '%@cs'
```

Output of other columns outside of an aggregate is undefined:

- **Unless...**

```sql
SELECT AVG(s.gpa), e.cid FROM enrolled AS e, student AS s WHERE e.sid = s.sid
GROUP BY e.cid
```

GROUP BY

- Project tuples into subsets and calc aggregates against each subset.

```sql
SELECT AVG(s.gpa), e.cid FROM enrolled AS e, student AS s WHERE e.sid = s.sid
GROUP BY e.cid
```
GROUP BY

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

SELECT AVG(s.gpa), e.cid, s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid

GROUP BY

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

SELECT AVG(s.gpa), e.cid, s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid, s.name

HAVING

- Filters output results
- Like a WHERE clause for a GROUP BY

SELECT AVG(s.gpa) AS avg_gpa, e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
HAVING avg_gpa > 2.75;
All-in-One Example

- Store the total balance of the cities that have branches with more than $1m in assets and where the total balance is more than $700, sorted by city name in descending order.

```sql
SELECT bcity, SUM(balance) AS totalbalance
INTO BranchAcctSummary
FROM branch AS b, account AS a
WHERE b.bname=a.bname AND assets > 1000000
GROUP BY bcity
HAVING totalbalance >= 700
ORDER BY bcity DESC
```

Step 1, 2: FROM, WHERE

<table>
<thead>
<tr>
<th>b.name</th>
<th>b.city</th>
<th>b.assets</th>
<th>a.name</th>
<th>a.acct_no</th>
<th>a.balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>$9,000,000</td>
<td>Downtown</td>
<td>A-101</td>
<td>$500</td>
</tr>
<tr>
<td>Compton</td>
<td>Los Angeles</td>
<td>$2,100,000</td>
<td>Compton</td>
<td>A-215</td>
<td>$300</td>
</tr>
<tr>
<td>Long Beach</td>
<td>Los Angeles</td>
<td>$1,400,000</td>
<td>Long Beach</td>
<td>A-102</td>
<td>$400</td>
</tr>
<tr>
<td>Harlem</td>
<td>New York</td>
<td>$7,000,000</td>
<td>Harlem</td>
<td>A-202</td>
<td>$350</td>
</tr>
<tr>
<td>Marcy</td>
<td>New York</td>
<td>$2,100,000</td>
<td>Marcy</td>
<td>A-305</td>
<td>$900</td>
</tr>
<tr>
<td>Marcy</td>
<td>New York</td>
<td>$2,100,000</td>
<td>Marcy</td>
<td>A-217</td>
<td>$750</td>
</tr>
</tbody>
</table>

Step 3: GROUP BY

Step 4: SELECT

<table>
<thead>
<tr>
<th>b.city</th>
<th>totalbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>500</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>1100</td>
</tr>
<tr>
<td>New York</td>
<td>2000</td>
</tr>
</tbody>
</table>

Step 5: HAVING

<table>
<thead>
<tr>
<th>b.city</th>
<th>totalbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>1100</td>
</tr>
<tr>
<td>New York</td>
<td>2000</td>
</tr>
</tbody>
</table>

Step 6: ORDER BY

<table>
<thead>
<tr>
<th>b.city</th>
<th>totalbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>2000</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>1100</td>
</tr>
</tbody>
</table>

Step 7: INTO < Store in new table >
Summary

<table>
<thead>
<tr>
<th>Clause</th>
<th>Evaluation Order</th>
<th>Semantics (RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT(DISTINCT)</td>
<td>4</td>
<td>p* (or p)</td>
</tr>
<tr>
<td>FROM</td>
<td>1</td>
<td>X*</td>
</tr>
<tr>
<td>WHERE</td>
<td>2</td>
<td>s*</td>
</tr>
<tr>
<td>INTO</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>GROUP BY</td>
<td>3</td>
<td>Cannot Express</td>
</tr>
<tr>
<td>HAVING</td>
<td>5</td>
<td>s*</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>6</td>
<td>Cannot Express</td>
</tr>
</tbody>
</table>

Advantages of SQL

- Write once, run everywhere (in theory…)
  - Different DBMSs
  - Single-node DBMS vs. Distributed DBMS

```sql
SELECT cname, amt
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.amt > 1000
```

Distributed Execution

```sql
SELECT (cname, acctno)
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.amt > 1000
```

<table>
<thead>
<tr>
<th>Query</th>
<th>cname</th>
<th>acctno</th>
<th>bname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Georg Hegel</td>
<td>A-123</td>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>Max Stirner</td>
<td>A-789</td>
<td>A-123</td>
<td>Perry</td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>A-456</td>
<td>A-789</td>
<td>Downtown</td>
<td>1000</td>
</tr>
</tbody>
</table>
NoSQL

- NoSQL really means non-relational
  - Many NoSQL DBMSs are just key-value stores
    - Queries are often written in procedural code.
- Relax the guarantees of the relational model to gain better horizontal scalability.

NoSQL: Not Only SQL!

- Many NoSQL systems now support a SQL-like dialect.
  - Facebook’s Hive (http://bit.ly/qId8np)
  - Cassandra CQL (http://bit.ly/nGJLtX)
- Other systems support declarative languages:
Additional Information

- Online SQL validators:
  - http://developer.mimer.se/validator/
  - http://format-sql.com
- Links to Postgres, MySQL, and SQLite documentation will be posted.
- *When in doubt, try it out!*

Next Class

- DDLs
- Complex Joins
- Views
- Nested Subqueries
- Triggers
- Stored Procedures