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.

• ANSI Standard in 1986, ISO in 1987
  – Structured Query Language

• Current standard is SQL:2011
  – SQL:2011 → Temporal DBs, Pipelined DML
  – SQL:2008 → TRUNCATE, Fancy ORDER
  – SQL:1999 → Regex, triggers, OO

• Most DBMSs at least support SQL-92
• System Comparison:
  – http://troels.arvin.dk/db/rdbms/

Today's Party

• SELECT/INSERT/UPDATE/DELETE
• Table Definition (DDL)
• NULLs
• String/Date/Time/Set/Bag Operations
• Output Redirection/Control
• Aggregates/Group By

Example Database

CUSTOMER

<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>

ACCOUNT

<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…

```
π bname (σ amt>1000 (account))
```

But not quite….

```
π bname (σ amt>1000 (account))
```

Downtown
Redwood
Perry

Downtown
Redwood
Perry

Duplicates

Now we get the same result as the relational algebra

```
SELECT DISTINCT bname FROM account WHERE amt > 1000
```

Why preserve duplicates?

```
π bname (σ amt>1000 (customer ⋈ account))
```

Same as

```
π cname, amt (σ amt>1000 (customer ⋈ account))
```

```
cname acctno
Georg Hegel A-123
Friedrich Engels A-456
Max Stirner A-789
```

```
acctno bname amt
A-123 Redwood 1800
A-789 Downtown 2000
A-123 Perry 1500
A-456 Downtown 1000
```

Basic SQL Query Grammar

```
SELECT [DISTINCT|ALL] target-list FROM relation-list [WHERE qualification]
```

• **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 ≥.
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>

SELECT Clause

- Use * to get all attributes

\[
\text{SELECT } * \text{ FROM account}
\]

\[
\text{SELECT account.} \ast \text{ FROM account}
\]

- Use DISTINCT to eliminate dupes

\[
\text{SELECT DISTINCT bname FROM account}
\]

- Target list can include expressions

\[
\text{SELECT bname, amt*1.05 FROM account}
\]

FROM Clause

- Binds tuples to variable names

\[
\text{SELECT } \ast \text{ FROM customer, account}
\]

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

- Define what kind of join to use

\[
\text{SELECT customer.} \ast, \text{ account.amt FROM customer LEFT OUTER JOIN account}
\]

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

- 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! *
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
  ```

- Find customers with an account in the “Downtown” branch with more than $100.
  ```sql
  SELECT C.cname, A.amt AS camt
  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
```

More SQL

- INSERT
- UPDATE
- DELETE
- TRUNCATE

INSERT

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

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

- Short-hand version:

```
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.

```
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
```

Today's Party

- SELECT/INSERT/UPDATE/DELETE
- Table Definition (DDL)
- NULLs
- String/Date/Time/Set/Bag Operations
- Output Redirection/Control
- Aggregates/Group By

Example Database

<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>Trump</td>
<td>trump@cs</td>
<td>45</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>21</td>
<td>3.9</td>
</tr>
<tr>
<td>53677</td>
<td>Tupac</td>
<td>shakuri@cs</td>
<td>26</td>
<td>3.5</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>53688</td>
<td>Reggae203</td>
<td>D</td>
</tr>
<tr>
<td>53688</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Massage105</td>
<td>D</td>
</tr>
</tbody>
</table>
Table Definition (DDL)

```
CREATE TABLE <table-name> (  
    [column-definition]*  
    [constraints]*  
) [table-options];
```

- **Column-Definition**: Comma separated list of column names with types.
- **Constraints**: Primary key, foreign key, and other meta-data attributes of columns.
- **Table-Options**: DBMS-specific options for the table (not SQL-92).

Table Definition Example

```
CREATE TABLE student (  
    sid   INT,  
    name  VARCHAR(16),  
    login VARCHAR(32),  
    age   SMALLINT,  
    gpa   FLOAT  
);  
CREATE TABLE enrolled (  
    sid   INT,  
    cid   VARCHAR(32),  
    grade CHAR(1)  
);  
```

Common Data Types

- **CHAR(n)**, **VARCHAR(n)**
- **TINYINT**, **SMALLINT**, **INT**, **BIGINT**
- **NUMERIC(p, d)**, **FLOAT**, **DOUBLE**, **REAL**
- **DATE**, **TIME**
- **BINARY(n)**, **VARBINARY(n)**, **BLOB**

Comment About BLOBs

- Don’t store large files in your database!
- Put the file on the filesystem and store a URI in the database.
- Many app frameworks will do this automatically for you.
- More information:
  - *To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?*
Useful Non-standard Types

- **TEXT**
- **BOOLEAN**
- **ARRAY**
- Geometric primitives
- **XML/JSON**
- Some systems also support user-defined types.

Integrity Constraints

```sql
CREATE TABLE student (
    sid   INT PRIMARY KEY,
    name  VARCHAR(16),
    login VARCHAR(32) UNIQUE,
    age   SMALLINT CHECK (age > 0),
    gpa   FLOAT
);

CREATE TABLE enrolled (
    sid   INT REFERENCES student (sid),
    cid   VARCHAR(32) NOT NULL,
    grade CHAR(1),
    PRIMARY KEY (sid, cid)
);
```

Primary Keys

- Single-column primary key:
  ```sql
  CREATE TABLE student (  
      sid   INT PRIMARY KEY,
      :    
  );
  ```

- Multi-column primary key:
  ```sql
  CREATE TABLE enrolled (  
      :    
      PRIMARY KEY (sid, cid)
  );
  ```

Foreign Key References

- Single-column reference:
  ```sql
  CREATE TABLE enrolled (  
      sid   INT REFERENCES student (sid),
      :    
  );
  ```

- Multi-column reference:
  ```sql
  CREATE TABLE enrolled (  
      :    
      FOREIGN KEY (sid, ...) REFERENCES student (sid, ...)
  );
  ```
Foreign Key References

• You can define what happens when the parent table is modified:
  – CASCADE
  – RESTRICT
  – NO ACTION
  – SET NULL
  – SET DEFAULT

Value Constraints

• Ensure one-and-only-one value exists:

CREATE TABLE student ( 
  login VARCHAR(32) UNIQUE,
)

• Make sure a value is not null:

CREATE TABLE enrolled ( 
  cid VARCHAR(32) NOT NULL,
)

• Make sure that an expression evaluates to true for each row in the table:

CREATE TABLE enrolled ( 
  age SMALLINT CHECK (age > 0),
)

• Can be expensive to evaluate, so tread lightly...
Auto-Generated Keys

• Automatically create a unique integer id for whenever a row is inserted \((last + 1)\).

• Implementations vary wildly:
  – SQL:2003 → IDENTITY
  – MySQL → AUTO_INCREMENT
  – Postgres → SERIAL
  – SQL Server → SEQUENCE
  – DB2 → SEQUENCE
  – Oracle → SEQUENCE

Conditional Table Creation

• **IF NOT EXISTS** prevents the DBMS from trying to create a table twice.

```
CREATE TABLE IF NOT EXISTS student (  
   sid   INT PRIMARY KEY  
   name  VARCHAR(16),  
   login VARCHAR(32) UNIQUE,  
   age   SMALLINT CHECK (age > 0),  
   gpa   FLOAT 
);  
```

Dropping Tables

• Completely removes a table from the database. Deletes everything related to the table (e.g., indexes, views, triggers, etc):

```
DROP TABLE student;  
```

• Can also use **IF EXISTS** to avoid errors:

```
DROP TABLE IF EXISTS student;  
```
Modifying Tables

- SQL lets you add/drop columns in a table after it is created:

  ```sql
  ALTER TABLE student
  ADD COLUMN phone VARCHAR(32) NOT NULL;

  ALTER TABLE student
  DROP COLUMN login;
  ```

- *This is really expensive!!! Tread lightly...*

---

Accessing Table Schema

- You can query the DBMS’s internal `INFORMATION_SCHEMA` catalog to get info about the database.
- ANSI standard set of read-only views that provide info about all of the tables, views, columns, and procedures in a database
- Every DBMS also have non-standard shortcuts to do this.

```sql
SELECT * FROM INFORMATION_SCHEMA.TABLES
WHERE table_catalog = '<db name>'
```

- Postgres
  ```
  \d;
  ```
  ```
  SHOW TABLES;
  ```
- MySQL
  ```
  .tables;
  ```
  ```
  SELECT * FROM INFORMATION_SCHEMA.TABLES
  WHERE table_catalog = '<db name>'
  ```
- SQLite
  ```
  .tables;
  ```

---
Accessing Table Schema

- List the column info for the student table:

  SELECT * FROM INFORMATION_SCHEMA.COLUMNS
  WHERE table_name = 'student'

  \d student; Postgres
  DESCRIBE student; MySQL
  .schema student; SQLite

Today's Party

- SELECT/INSERT/UPDATE/DELETE
- Table Definition (DDL)
- NULLs
- String/Date/Time/Set/Bag Operations
- Output Redirection/Control
- Aggregates/Group By

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>$9,000,000</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.

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

<table>
<thead>
<tr>
<th>bname</th>
<th>city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oakland</td>
<td>Pittsburgh</td>
<td>$9,000,000</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>
NULLs

• Find all branches that have null assets.

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

<table>
<thead>
<tr>
<th>bname</th>
<th>city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oakland</td>
<td>Pittsburgh</td>
<td>$9,000,000</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>

NULLs

• Arithmetic operations with NULL values is always NULL.

```
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.

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

<table>
<thead>
<tr>
<th>eq_bool</th>
<th>neq_bool</th>
<th>and_false</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

String Case | String Quotes
---|---
SQL-92 | Sensitive | Single Only
Postgres | Sensitive | Single Only
MySQL | Insensitive | Single/Double
SQLite | Sensitive | Single/Double
DB2 | Sensitive | Single Only
Oracle | Sensitive | Single Only

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 '%rum_'
```

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 abbrev_name
FROM student WHERE sid = 53688
```

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

Date/Time Operations

- Operations to manipulate and modify **DATE/TIME** attributes.
- Can be used in either output and predicates.
- **Support/syntax varies wildly…**

- Demo: Get the # of days since the beginning of the year.

Set/Bag Operations

- **Set Operations:**
  - UNION
  - INTERSECT
  - EXCEPT

- **Bag Operations:**
  - UNION ALL
  - INTERSECT ALL
  - EXCEPT ALL
Set Operations

(SELECT cname FROM customer)
(SELECT cname FROM account)

UNION
Returns names of customers with or without an account.

INTERSECT
Returns names of customers with an account.

EXCEPT
Returns names of customers without an account.

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.

SELECT DISTINCT cid INTO CourseIds SQL-92
FROM enrolled;

CREATE TABLE CourseIds (MySQL
SELECT DISTINCT cid FROM enrolled);

Today's Party

• SELECT/INSERT/UPDATE/DELETE
• Table Definition (DDL)
• NULLs
• String/Date/Time/Set/Bag Operations
• Output Redirection/Control
• Aggregates/Group By

Faloutsos/Pavlo

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 SQL-92
(SELECT DISTINCT cid FROM Enrolled);

MySQL
Output Control

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

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

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

**Aggregates**

• Functions that return a single value from a bag of tuples:
  – `AVG(col)` → Return the average col value.
  – `MIN(col)` → Return minimum col value.
  – `MAX(col)` → Return maximum col value.
  – `SUM(col)` → Return sum of values in col.
  – `COUNT(col)` → Return # of values for col.

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

```sql
SELECT COUNT(*) AS cnt
  FROM student WHERE cid = 'Pilates105'
ORDER BY grade
```

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

```sql
SELECT sid, name FROM student
WHERE login LIKE '%@cs'
LIMIT 10
```

```sql
SELECT sid, name FROM student
WHERE login LIKE '%@cs'
LIMIT 20 OFFSET 10
```

First 10 rows

Skip first 10 rows, 
Return the following 20
Aggregates

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

```
SELECT AVG(gpa), COUNT(sid)
FROM student
WHERE login LIKE '%@cs'
```

- Output of other columns outside of an aggregate is undefined:

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

- Unless...

```
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.

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

<table>
<thead>
<tr>
<th>e.sid</th>
<th>s.sid</th>
<th>s.gpa</th>
<th>e.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>53435</td>
<td>53435</td>
<td>2.25</td>
<td>Pilates101</td>
</tr>
<tr>
<td>53439</td>
<td>53439</td>
<td>2.70</td>
<td>Pilates101</td>
</tr>
<tr>
<td>53423</td>
<td>53423</td>
<td>2.98</td>
<td>Topology12</td>
</tr>
<tr>
<td>56023</td>
<td>56023</td>
<td>2.75</td>
<td>Reggae203</td>
</tr>
<tr>
<td>59439</td>
<td>59439</td>
<td>3.90</td>
<td>Reggae203</td>
</tr>
<tr>
<td>53961</td>
<td>53961</td>
<td>3.50</td>
<td>Reggae203</td>
</tr>
<tr>
<td>58345</td>
<td>58345</td>
<td>1.89</td>
<td>Massage105</td>
</tr>
</tbody>
</table>
**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
```

**X**

```
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.

```
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
```
All-in-One Example

Steps 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>$700</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

Step 6: ORDER BY

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

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

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

<table>
<thead>
<tr>
<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>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>Friedrich Engels</td>
<td>A-456</td>
<td>Downtown</td>
<td>2000</td>
</tr>
<tr>
<td>Max Stirner</td>
<td>A-789</td>
<td>Perry</td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>A-123</td>
<td>Downtown</td>
<td>1000</td>
</tr>
</tbody>
</table>

Stupid Joins Are Stupid

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

<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>

- Send customer to every node?
- Send account to every node?

Additional Information

- Online SQL validators:
  - http://developer.mimer.se/validator/
  - http://format-sql.com
- When in doubt, try it out!

Next Class

- Complex Joins
- Views
- Subqueries
- Common Table Expressions
- Window Functions
- Triggers
- Database Application Example