Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo

Lecture#6: Fun with SQL (part1)

General Overview - Rel. Model

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

"Intergalactic Standard"

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.
  - **S**tructured **E**nglish **Que**ry **L**anguage
  - Adopted by Oracle in the 1970s.
- ANSI Standard in 1986, ISO in 1987
  - **S**tructured **Q**uery **L**anguage

History

- Current standard is **SQL:2011**
  - **SQL:2011** → Temporal DBs, Pipelined DML
  - **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/

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

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

ACCOUNT

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

First SQL Example

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

Find the names of the branches that have an account with more than $1000.

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700
Redwood	L-450	3000

Similar to…

$\pi_{bname}(\sigma_{amt>1000}(account))$

bname
Downtown
Redwood
Perry

But not quite….

bname
Downtown
Redwood
Perry
Redwood

Duplicates

First SQL Example

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

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700
Redwood	L-450	3000

Now we get the same result as the relational algebra

bname
Downtown
Redwood
Perry

Why preserve duplicates?

Multi-Relation Queries

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

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

Same as

$\pi_{cname, amt}(\sigma_{amt>1000}(customer \bowtie account))$

cname	amt
Georg Hegel	1800
Max Stirner	2000
Georg Hegel	1500

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}

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

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!*

SELECT Clause

- Use * to get all attributes

```
SELECT * FROM account
```

```
SELECT account.* FROM account
```

- Use **DISTINCT** to eliminate dupes

```
SELECT DISTINCT bname FROM account
```

- Target list can include expressions

```
SELECT bname, amt*1.05 FROM account
```


FROM Clause

- Binds tuples to variable names

```
SELECT * FROM customer, account
WHERE customer.acctno = account.acctno
```

- Define what kind of join to use

```
SELECT customer.*, account.amt
 FROM customer LEFT OUTER JOIN account
WHERE customer.acctno = account.acctno
```

WHERE Clause

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

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

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

```
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.

```
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.

```
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.

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

```
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

STUDENT

sid	name	login	age	gpa
53666	Trump	trump@cs	45	4.0
53688	Bieber	jbieber@cs	21	3.9
53677	Tupac	shakur@cs	26	3.5

ENROLLED

sid	cid	grade
53831	Pilates101	C
53688	Reggae203	D
53688	Topology112	A
53666	Massage105	D

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)
);
```

Integer Range

Variable String Length

Fixed String Length

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

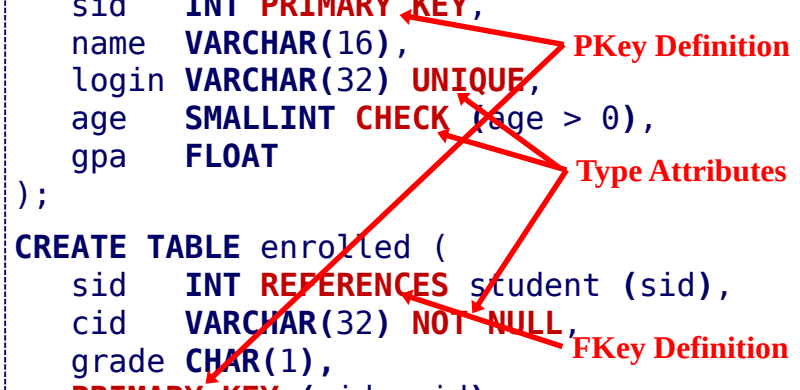
- Don't store large files in your database!
- Put the file on the filesystem and store a URI in the database.
- Many app frameworks will do this automatically for you.
- More information:
  - *To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?*

Useful Non-standard Types

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

Integrity Constraints

```
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:

```
CREATE TABLE student (
   sid    INT PRIMARY KEY,
   ⋮
```

- Multi-column primary key:

```
CREATE TABLE enrolled (
   ⋮
   PRIMARY KEY (sid, cid)
```


Foreign Key References

- Single-column reference:

```
CREATE TABLE enrolled (
   sid    INT REFERENCES student (sid),
   ⋮
```

- Multi-column reference:

```
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**

Foreign Key References

- Delete/update the enrollment information when a student is changed:

```
CREATE TABLE enrolled (
   ⋮
   FOREIGN KEY (sid)
     REFERENCES student (sid)
     ON DELETE CASCADE
     ON UPDATE CASCADE
```


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,
```


Value Constraints

- 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`**

Auto-Generated Keys

MySQL

```
CREATE TABLE student (
   sid   INT PRIMARY KEY AUTO_INCREMENT,
   ⋮
```

```
INSERT INTO student
  (sid, name, login, age, gpa)
VALUES
  (NULL, "Trump", "@cs", 45, 4.0);
```


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:

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

```
ALTER TABLE student DROP COLUMN login;
```

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

Modifying Tables

- You can also modify existing columns (rename, change type, change defaults, etc):

Postgres

```
ALTER TABLE student
  ALTER COLUMN name TYPE VARCHAR(32);
```

MySQL

```
ALTER TABLE student
  CHANGE COLUMN name name VARCHAR(32);
```


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.

Accessing Table Schema

- List all of the tables in the current database:

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

Postgres

```
\d;
```

MySQL

```
SHOW TABLES;
```

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.

bname	city	assets
Oakland	Pittsburgh	$9,000,000
Compton	Los Angeles	NULL
Long Beach	Los Angeles	$400,000
Harlem	New York	$1,700,000

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

NULLs

- Find all branches that have null assets.

bname	city	assets
Oakland	Pittsburgh	$9,000,000
Compton	Los Angeles	NULL
Long Beach	Los Angeles	$400,000
Harlem	New York	$1,700,000

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

bname	city	assets

NULLs

- Find all branches that have null assets.

bname	city	assets
Oakland	Pittsburgh	$9,000,000
Compton	Los Angeles	NULL
Long Beach	Los Angeles	$400,000
Harlem	New York	$1,700,000

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

bname	city	assets
Compton	Los Angeles	NULL

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;
```

add_null	sub_null	mul_null	div_null
NULL	NULL	NULL	NULL

NULLs

- Comparisons with **NULL** values varies.

```
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;
```

eq_bool	neq_bool	and_false	eq_null	is_null
NULL	NULL	NULL	NULL	TRUE

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

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

```
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:

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

```
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.

Today's Party

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

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-92
```
SELECT DISTINCT cid INTO CourseIds
  FROM enrolled;
```

MySQL
```
CREATE TABLE CourseIds (
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.

SQL-92
```
INSERT INTO CourseIds
(SELECT DISTINCT cid FROM Enrolled);
```

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
```

sid	grade
53123	A
53334	A
53650	B
53666	D

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

sid
53666
53650
53123
53334

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 '%@cs'
 LIMIT 20 OFFSET 10
```

Skip first 10 rows, Return the following 20

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.

Aggregates

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

```
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.

```
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.

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

COUNT(DISTINCT login)
10

Aggregates

- 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
```

AVG(s.gpa)	e.cid
3.5	???

- *Unless…*

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
```

e.sid	s.sid	s.gpa	e.cid
53435	53435	2.25	Pilates101
53439	53439	2.70	Pilates101
53423	53423	2.98	Topology112
56023	56023	2.75	Reggae203
59439	59439	3.90	Reggae203
53961	53961	3.50	Reggae203
58345	58345	1.89	Massage105

AVG(s.gpa)	e.cid
2.46	Pilates101
3.39	Reggae203
2.98	Topology112
1.89	Massage105

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;
```

AVG(s.gpa)	e.cid
2.46	Pilates101
3.39	Reggae203
2.98	Topology112
1.89	Massage105

avg_gpa	e.cid
3.39	Reggae203
2.98	Topology112

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`

b.bname	b.city	b.assets	a.bname	a.acct_no	a.balance
Downtown	Boston	$9,000,000	Downtown	A-101	$500
Compton	Los Angeles	$2,100,000	Compton	A-215	$700
Long Beach	Los Angeles	$1,400,000	Long Beach	A-102	$400
Harlem	New York	$7,000,000	Harlem	A-202	$350
Marcy	New York	$2,100,000	Marcy	A-305	$900
Marcy	New York	$2,100,000	Marcy	A-217	$750

All-in-One Example

Step 3: `GROUP BY`
Step 4: `SELECT`

b.city	totalbalance
Boston	500
Los Angeles	1100
New York	2000

Step 5: `HAVING`

b.city	totalbalance
Los Angeles	1100
New York	2000

Step 6: `ORDER BY`

b.city	totalbalance
New York	2000
Los Angeles	1100

Step 7: `INTO` *< Store in new table >*

Summary

Clause	Evaluation Order	Semantics (RA)
`SELECT[DISTINCT]`	4	p* (or p)
`FROM`	1	X*
`WHERE`	2	s*
`INTO`	7	←
`GROUP BY`	3	Cannot Express
`HAVING`	5	s*
`ORDER BY`	6	Cannot Express

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
```

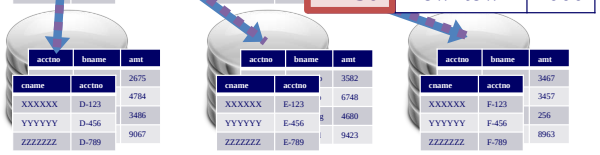
Distributed Execution

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

Query Request

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

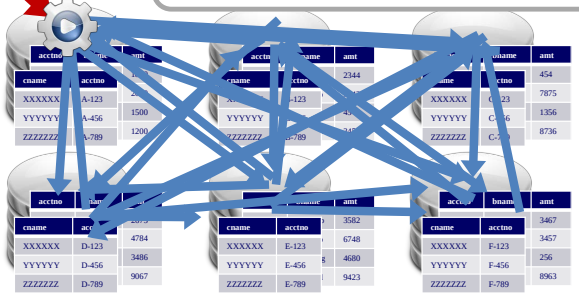


Stupid Joins Are Stupid

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

Query Request

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