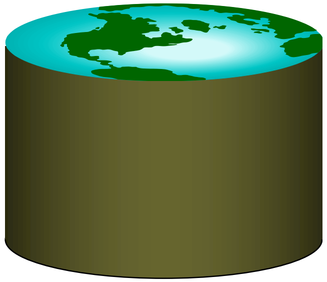
SQL: The Query Language (Part 2)

Fall 2016

Life is just a bowl of queries.

-Anon



Recall: SQL

- **SQL is a *very* high-level language.**
  - Say "what to do" rather than "how to do it."
  - Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java.
- **Database management system figures out "best" way to execute query.**
  - Called "query optimization."

Recall: Basic SQL Query

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

- *relation-list* : A list of relation names
  - possibly with a *range-variable* after each name
- *target-list* : A list of attributes of tables in *relation-list*
- *qualification* : Comparisons combined using AND, OR and NOT.
  - Comparisons are Attr *op* const or Attr1 *op* Attr2, where *op* is one of $= \neq < > \leq \geq$
- *DISTINCT*: optional keyword indicating that the answer should not contain duplicates.
  - In SQL SELECT, the default is that duplicates are *not* eliminated! (Result is called a "multiset")

Recall: Basic SQL Query

DISTINCT: optional. Answer should not contain duplicates.
SQL default: duplicates are *not* eliminated! (Result a "multiset")

target-list : List of expressions over attributes of tables in *relation-list*

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

qualification : Comparisons combined using AND, OR and NOT. Comparisons are Attr *op* const or Attr1 *op* Attr2, where *op* is one of =,<,>,≠, etc.

relation-list : List of relation names, possibly with a *range-variable* "AS" clause after each name

Query Semantics

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

1. FROM : compute cross product of tables.
2. WHERE : Check conditions, discard tuples that fail.
3. SELECT : Delete unwanted fields.
4. DISTINCT (optional) : eliminate duplicate rows.

- **Note: likely a terribly inefficient strategy!**
  - Query optimizer will find more efficient plans.

Essential SQL Statements

```
CREATE DATABASE database_name
```

```
DROP DATABASE database_name
```

```
CREATE TABLE table_name
(
column_name1 data_type,
column_name2 data_type,
column_name3 data_type,
...
)
```

```
DROP TABLE table_name
DELETE FROM table_name
DELETE * FROM table_name
```

```
CREATE INDEX index_name
ON table_name (column_name)
```

```
DROP INDEX index_name
```

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name)
```

```
INSERT INTO table_name
VALUES (value1, value2,....)
INSERT INTO table_name
(column1, column2,...)
VALUES (value1, value2,....)
```

```
DELETE FROM table_name
WHERE some_column=some_value
```


Example: Relation (Table)

Row/Tuple/Record

Column/Attribute/Field

name	birth	gpa	grad
Anderson	1987-10-22	3.9	2009
Jones	1990-4-16	2.4	2012
Hernandez	1989-8-12	3.1	2011
Chen	1990-2-4	3.2	2011
VARCHAR(30)	DATE	FLOAT	INT

Column Types → VARCHAR(30) DATE FLOAT INT

Example: Primary Key

Unique For Each Row

id	name	birth	gpa	grad
14	Anderson	1987-10-22	3.9	2009
38	Jones	1990-4-16	2.4	2012
77	Hernandez	1989-8-12	3.1	2011
104	Chen	1990-2-4	3.2	2011
INT	VARCHAR(30)	DATE	FLOAT	INT

Basic Table Operations

```
CREATE TABLE students (
    id INT AUTO_INCREMENT,
    name VARCHAR(30),
    birth DATE,
    gpa FLOAT,
    grad INT,
    PRIMARY KEY(id));

INSERT INTO students(name, birth, gpa, grad)
      VALUES ('Anderson', '1987-10-22', 3.9, 2009);
INSERT INTO students(name, birth, gpa, grad)
      VALUES ('Jones', '1990-4-16', 2.4, 2012);

DELETE FROM students WHERE name='Anderson';

DROP TABLE students;
```

Query: Display Entire Table

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-4-16	2.4	2012
3	Hernandez	1989-8-12	3.1	2011
4	Chen	1990-2-4	3.2	2011

```
SELECT * FROM students;
```

```
+----+-----------+------------+------+------+
| id | name      | birth      | gpa  | grad |
+----+-----------+------------+------+------+
|  1 | Anderson  | 1987-10-22 |  3.9 | 2009 |
|  2 | Jones     | 1990-04-16 |  2.4 | 2012 |
|  3 | Hernandez | 1989-08-12 |  3.1 | 2011 |
|  4 | Chen      | 1990-02-04 |  3.2 | 2011 |
+----+-----------+------------+------+------+
```

Query: Select Columns `name` and `gpa`

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-4-16	2.4	2012
3	Hernandez	1989-8-12	3.1	2011
4	Chen	1990-2-4	3.2	2011

```
SELECT name, gpa
FROM students;
```

```
+-----------+------+
| name      | gpa  |
+-----------+------+
| Anderson  |  3.9 |
| Jones     |  2.4 |
| Hernandez |  3.1 |
| Chen      |  3.2 |
+-----------+------+
```

Query: Filter Rows

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-4-16	2.4	2012
3	Hernandez	1989-8-12	3.1	2011
4	Chen	1990-2-4	3.2	2011

```
SELECT name, gpa FROM students
WHERE gpa > 3.0;
```

```
+-----------+------+
| name      | gpa  |
+-----------+------+
| Anderson  |  3.9 |
| Hernandez |  3.1 |
| Chen      |  3.2 |
+-----------+------+
```

Query: Sort Output

- **The ORDER BY keyword is used to sort the result-set by one or more columns**

```
SELECT column_name, column_name
FROM table_name
ORDER BY column_name ASC|DESC, column_name ASC|DESC;
```


Query: Sort Output by `gpa`

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-4-16	2.4	2012
3	Hernandez	1989-8-12	3.1	2011
4	Chen	1990-2-4	3.2	2011

```
SELECT gpa, name, grad FROM students
WHERE gpa > 3.0
ORDER BY gpa DESC;

+------+-----------+------+
| gpa  | name      | grad |
+------+-----------+------+
|  3.9 | Anderson  | 2009 |
|  3.2 | Chen      | 2011 |
|  3.1 | Hernandez | 2011 |
+------+-----------+------+
```

Update Value(s)

- **The UPDATE statement is used to update existing records in a table.**

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```


Update Value(s)

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-4-16	2.4	2012
3	Hernandez	1989-8-12	3.1	2011
4	Chen	1990-2-4	3.2	2011

```
UPDATE students
SET gpa = 2.6, grad = 2013
WHERE id = 2
```

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-4-16	2.6	2013
3	Hernandez	1989-8-12	3.1	2011
4	Chen	1990-2-4	3.2	2011

Foreign Key

students

id	name	birth	gpa	grad	advisor_id
1	Anderson	1987-10-22	3.9	2009	2
2	Jones	1990-4-16	2.4	2012	1
3	Hernandez	1989-8-12	3.1	2011	1
4	Chen	1990-2-4	3.2	2011	1

advisors

id	name	title
1	Fujimura	assocprof
2	Bolosky	prof

```
SELECT s.name, s.gpa
FROM students s, advisors p
WHERE s.advisor_id = p.id AND p.name = 'Fujimura';
```

s.id	s.name	s.birth	s.gpa	s.grad	s.advisor_id	p.id	p.name	p.title
1	Anderson	1987-10-22	3.9	2009	2	1	Fujimura	assocprof
1	Anderson	1987-10-22	3.9	2009	2	2	Bolosky	prof
2	Jones	1990-4-16	2.4	2012	1	1	Fujimura	assocprof
2	Jones	1990-4-16	2.4	2012	1	2	Bolosky	prof
3	Hernandez	1989-8-12	3.1	2011	1	1	Fujimura	assocprof
3	Hernandez	1989-8-12	3.1	2011	1	2	Bolosky	prof
4	Chen	1990-2-4	3.2	2011	1	1	Fujimura	assocprof
4	Chen	1990-2-4	3.2	2011	1	2	Bolosky	prof

students

id	name	birth	gpa	grad	advisor_id
1	Anderson	1987-10-22	3.9	2009	2
2	Jones	1990-4-16	2.4	2012	1
3	Hernandez	1989-8-12	3.1	2011	1
4	Chen	1990-2-4	3.2	2011	1

advisors

id	name	title
1	Fujimura	assocprof
2	Bolosky	prof

```
SELECT s.name, s.gpa
FROM students s, advisors p
WHERE s.advisor_id = p.id AND p.name = 'Fujimura';
```

s.id	s.name	s.birth	s.gpa	s.grad	s.advisor_id	p.id	p.name	p.title
1	Anderson	1987-10-22	3.9	2009	2	1	Fujimura	assocprof
1	Anderson	1987-10-22	3.9	2009	2	2	Bolosky	prof
2	Jones	1990-4-16	2.4	2012	1	1	Fujimura	assocprof
2	Jones	1990-4-16	2.4	2012	1	2	Bolosky	prof
3	Hernandez	1989-8-12	3.1	2011	1	1	Fujimura	assocprof
3	Hernandez	1989-8-12	3.1	2011	1	2	Bolosky	prof
4	Chen	1990-2-4	3.2	2011	1	1	Fujimura	assocprof
4	Chen	1990-2-4	3.2	2011	1	2	Bolosky	prof

students

id	name	birth	gpa	grad	advisor_id
1	Anderson	1987-10-22	3.9	2009	2
2	Jones	1990-4-16	2.4	2012	1
3	Hernandez	1989-8-12	3.1	2011	1
4	Chen	1990-2-4	3.2	2011	1

advisors

id	name	title
1	Fujimura	assocprof
2	Bolosky	prof

```
SELECT s.name, s.gpa
    FROM students s, advisors p
    WHERE s.advisor_id = p.id AND p.name = 'Fujimura';
```

```
+-----------+------+
| name      | gpa  |
+-----------+------+
| Jones     |  2.4 |
| Hernandez |  3.1 |
| Chen      |  3.2 |
+-----------+------+
```

students

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-4-16	2.4	2012
3	Hernandez	1989-8-12	3.1	2011
4	Chen	1990-2-4	3.2	2011

courses

id	number	name	quarter
1	CS142	Web stuff	Winter 2009
2	ART101	Finger painting	Fall 2008
3	ART101	Finger painting	Winter 2009
4	PE204	Mud wrestling	Winter 2009

courses_students

course_id	student_id
1	1
3	1
4	1
1	2
2	2
1	3
2	4
4	4

```
SELECT s.name, c.quarter
FROM students s, courses c, courses_students cs
WHERE c.id = cs.course_id AND s.id = cs.student_id
AND c.number = 'ART101';
```

```
+----------+-------------+
| name     | quarter     |
+----------+-------------+
| Jones    | Fall 2008   |
| Chen     | Fall 2008   |
| Anderson | Winter 2009 |
+----------+-------------+
```

Back to Our Running Example …

```
CREATE TABLE Sailors (
   sid INTEGER,
   sname CHAR(20),
   rating INTEGER,
   age REAL,
   PRIMARY KEY sid);

CREATE TABLE Boats (
   bid INTEGER,
   bname CHAR (20),
   color CHAR(10)
   PRIMARY KEY bid);

 CREATE TABLE Reserves (
   sid INTEGER,
   bid INTEGER,
   day DATE,
  PRIMARY KEY (sid, bid, day),
  FOREIGN KEY sid REFERENCES Sailors,
  FOREIGN KEY bid REFERENCES Boats);
```

<u>sid</u>	sname	rating	age
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

<u>bid</u>	bname	color
101	Nina	red
102	Pinta	blue
103	Santa Maria	red

<u>sid</u>	<u>bid</u>	<u>day</u>
1	102	9/12
2	102	9/13

Back to Our Running Example …

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
95	103	11/12/96

Sailors

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

Boats

<u>bid</u>	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Nested Queries: IN

- **Powerful feature of SQL: WHERE clause can itself contain an SQL query!**
  - Actually, so can FROM and HAVING clauses.

Names of sailors who've reserved boat #103:

```
SELECT   S.sname
FROM   Sailors S
WHERE   S.sid IN (SELECT R.sid
                  FROM   Reserves R
                  WHERE   R.bid=103)
```

- **To find sailors who've *not* reserved #103, use NOT IN.**
- **To understand semantics of nested queries:**
  - think of a *nested loops* evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

Nested Queries: NOT EXIST

Names of sailors who've <u>never</u> reserved boat #103

```
SELECT  S.sname
FROM  Sailors S
WHERE   NOT EXISTS  (SELECT *
                    FROM  Reserves R
                    WHERE  R.bid=103 AND S.sid=R.sid)
```

Sailor *S* reserves boat 103

Nested Queries with Correlation

Find names of sailors who've reserved boat #103:

```
SELECT  S.sname
FROM  Sailors S
WHERE EXISTS (SELECT  *
             FROM  Reserves R
             WHERE R.bid=103 AND S.sid=R.sid)
```

- **EXISTS is another set comparison operator, like IN.**
- **Can also specify NOT EXISTS**
- **If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103.**
  - UNIQUE checks for duplicate tuples in a subquery;
- **Subquery must be recomputed for each Sailors tuple.**
  - Think of subquery as a function call that runs a query!

Nested Queries with Correlation (2)

Names of sailors who've reserved boat 103 at most once.

```
SELECT  S.sname
FROM  Sailors S
WHERE  UNIQUE (SELECT R.bid
               FROM  Reserves R
               WHERE  R.bid=103 AND S.sid=R.sid)
```

- **UNIQUE returns true if no row appears more than once.**
  (Note: returns true if answer is empty)
- **Can we replace "SELECT *R.bid*" *by* "SELECT * "?**
  No, A sailor may reserve boat 103 on different days; and UNIQUE would return true

Nested Queries: NOT IN

Names of sailors who've ***not*** *reserved boat #103:*

```
SELECT   S.sname
FROM     Sailors S
WHERE    S.sid NOT IN
    (SELECT   R.sid
     FROM     Reserves R
    WHERE   R.bid=103)
```

Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and a green boat:

```
SELECT   R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid
      AND B.color='red'
      AND R.sid IN (SELECT R2.sid
                    FROM   Boats B2, Reserves R2
                    WHERE   R2.bid=B2.bid
                    AND   B2.color='green')
```

- **Similarly, EXCEPT queries re-written using NOT IN.**
- **How would you change this to find *names* (not *sid*'s) of Sailors who've reserved both red and green boats?**

More SQL Statements

```
ALTER TABLE table_name
ADD column_name datatype
```

```
ALTER TABLE table_name
DROP COLUMN column_name
```

```
SELECT column_name AS column_alias
FROM table_name
```

```
SELECT column_name
FROM table_name  AS table_alias
```

```
SELECT column_name(s)
FROM table_name
WHERE column_name
BETWEEN value1 AND value2
```

```
SELECT column_name(s)
FROM table_name
WHERE condition
AND|OR condition
```


Other SQL Statements

- **AUTO INCREMENT Field**
- **SELECT INTO**
  - Selects data from one table and inserts it into a new table
- **LIMIT**
  - Specify the number of records to return
- **CREATE VIEW**
  - Create a virtual table based on the result-set of an SQL statement
- **TRUNCATE TABLE**
  - Delete all table contents

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

```
SELECT column_name(s)
INTO newtable [IN
externaldb]
FROM table1;
```

```
SELECT column_name(s)
FROM table_name
LIMIT number;
```


SQL Operators

- **BETWEEN**
- **NOT BETWEEN**
- **IN**
- **UNION** [**DISTINCT** | **ALL**]
- **EXCEPT**

```
SELECT * FROM Products
WHERE (Price BETWEEN 10 AND 20)
AND NOT CategoryID IN (1,2,3);
```


More Examples…

Find sid's of sailors who have not reserved a boat

```
SELECT S.sid
FROM   Sailors S

EXCEPT

SELECT S.sid
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid
```

More on Set-Comparison Operators

- **We've already seen IN, NOT IN, EXISTS and UNIQUE.**
- **Can also use NOT EXISTS and NOT UNIQUE.**
- **Also available:** ***op*** **ANY,** ***op*** **ALL,** where *op*: >, <, =, ≠, ≥, ≤

Find sailors whose rating is greater than that of some sailor called Horatio

```
SELECT   *
FROM   Sailors S
WHERE  S.rating > ANY  (SELECT  S2.rating
                        FROM  Sailors S2
                        WHERE S2.sname='Horatio')
```

>ANY means greater than at least one value
>ALL means greater than every value

ARGMAX?

- **The sailor with the highest rating**
  - what about ties for highest?!

```
SELECT  *
FROM    Sailors S
WHERE   S.rating >= ALL
    (SELECT   S2.rating
     FROM   Sailors S2)
```

```
SELECT  *
FROM    Sailors S
WHERE   S.rating =
(SELECT   MAX(S2.rating)
     FROM   Sailors S2)
```

```
SELECT  *
FROM    Sailors S
ORDER BY rating DESC
LIMIT 1;
```

Division Operations in SQL (1)

*Find names of sailors who've reserved **all** boat:*

```
SELECT  S.sname
FROM    Sailors S
WHERE   NOT EXIST  ((SELECT  B.bid
                     FROM   Boats B) EXCEPT
                      (SELECT R.bid
                       FROM   Reserves R
                       WHERE   R.sid = S.sid ))
```

The sailor reserved all boats

Boats not reserved by the sailor

All boats

All boats reserved by the sailor

Division Operations in SQL (2)

*Find names of sailors who've reserved **all** boat:*

```
SELECT  S.sname
FROM    Sailors S
WHERE  NOT EXIST  ((SELECT   B.bid
                    FROM     Boats B
                    WHERE  NOT EXISTS (SELECT R.bid
                                       FROM   Reserves R
                                       WHERE   R.bid = B.bid
                                         AND R.sid = S.sid))
```

Sailor *S*

such that there is no boat *B*

without a reservation

showing

Sailor *S* reserved boat *B*

Sailor S such that …
there is no boat B without …
a Reserves tuple showing S reserved B.

Joins

```
SELECT (column_list)
FROM  table_name
 [INNER | {LEFT |RIGHT | FULL } OUTER] JOIN table_name
   ON qualification_list
WHERE …
```

Explicit join semantics needed unless it is an INNER join (INNER is default)

Inner Join

Only the rows that match the search conditions are returned.

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

Returns only those sailors who have reserved boats

SQL-92 also allows:

```
SELECT s.sid, s.name, r.bid
FROM Sailors s NATURAL JOIN Reserves r
```

"NATURAL" means equi-join for each pair of attributes with the same name (may need to rename with "AS")

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

sid	bid	day
22	101	10/10/96
95	103	11/12/96

s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103

Outer Joins

No "sid = 31"

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1 ⟕ *R1*

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
31	lubber	8	55.55	null	null
58	rusty	10	35.0	103	11/12/96

No match in *R1*

Left Outer Join

SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

sid	bid	day
22	101	10/10/96
95	103	11/12/96

Left Outer Join returns all matched rows, plus all unmatched rows from the table on the left of the join clause
(use nulls in fields of non-matching tuples)

SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid

s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103
31	Lubber	*null*

Returns all sailors & information on whether they have reserved boats

Right Outer Join

SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid

sid	bid	day
22	101	10/10/96
95	103	11/12/96

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Right Outer Join returns all matched rows, plus all unmatched rows from the table on the right of the join clause

SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid

r.sid	b.bid	b.name
22	101	Interlake
null	102	Interlake
95	103	Clipper
null	104	Marine

Returns all boats & information on which ones are reserved.

Full Outer Join

Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

```
SELECT r.sid, b.bid, b.name
FROM Sailors s FULL OUTER JOIN Boats b
ON s.sname = b.bname
```

```
SELECT s.sid, s.sname, b.bid, b.name
FROM Sailors s FULL OUTER JOIN Boats b
ON s.sname = b.bname
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

bid	bname	color
101	Interlake	blue
105	Lubber	purple

sid	sname	bid	bname
22	Dustin	*null*	*null*
31	Lubber	105	Lubber
95	Bob	*null*	*null*
null	*null*	101	Interlake

Recall: Aggregate Functions

Significant extension of relational algebra

COUNT (*)	The number of rows in the relation
COUNT ([DISTINCT] A)	The number of (unique) values in the A column
SUM ([DISTINCT] A)	The sum of all (unique) values in the A column
AVG ([DISTINCT] A)	The average of all (unique) values in the A column
MAX (A)	The maximum value in the A column
MIN (A)	The minimum value in the A column

Aggregate Operators

```
SELECT COUNT (*)
FROM   Sailors S
```

Count the number of sailors

```
SELECT S.sname
...ing)
```

Find the names of sailors with the highest rating

```
SELECT  AVG (S.age)
FROM    Sailors S
WHERE   S.rating=10
```

Find the average ag...

```
SELECT  COUNT (DISTINCT S.rating)
FROM    Sailors S
WHERE   S.sname='Bob'
```

Count the number of distinct ratings of sailors called "Bob"

```
SELECT  AVG (DISTINCT S.age)
FROM    Sailors S
WHERE   S.rating=10
```

...d the average of the ...inct ages of sailors ...ith a rating of 10

Aggregate Operators

```
SELECT COUNT (*)
FROM   Sailors S
```

```
SELECT S.sname
FROM   Sailors S
WHERE  S.rating= (SELECT MAX(S2.rating)
                  FROM  Sailors S2)
```

```
SELECT  AVG (S.age)
FROM    Sailors S
WHERE   S.rating=10
```

```
SELECT  AVG (DISTINCT S.age)
FROM    Sailors S
WHERE   S.rating=10
```

```
SELECT  COUNT (DISTINCT S.rating)
FROM    Sailors S
WHERE   S.sname='Bob'
```

More Aggregate Functions…

```
Avg()          Lcase()
Count()        Mid()
First()        Len()
Last()         Round()
Max()          Now()
Min()          Format()
SQL
Sum()
Group By
Having
Ucase()
```

```
SELECT column_name,
aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING
aggregate_function(column_name)
operator value
```


Restriction on SELECT Lists With Aggregation

- **If any aggregation is used, then each element of the SELECT list must be either:**
  1. Aggregated, or
  2. An attribute on the GROUP BY list.

Illegal Query Example

- **You might think you could find the bar that sells Bud the cheapest by:**

```
SELECT bar, MIN(price)
FROM Sells
WHERE beer = 'Bud';
```

- **But this query is illegal in SQL.**
- **Why?**

Find name and age of the oldest sailor(s)

Only aggregate operations allowed

```
SELECT S.sname, MAX (S.age)
FROM  Sailors S
```

Find name and age of the oldest sailor(s)

- **The first query is incorrect!**
- **Third query equivalent to second query**
  - allowed in SQL/92 standard, but not supported in some systems.

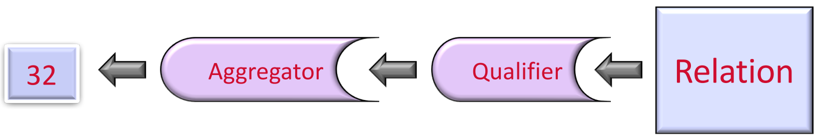
```
SELECT  S.sname, MAX (S.age)
FROM  Sailors S
```

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  S.age =
          (SELECT  MAX (S2.age)
           FROM  Sailors S2)
```

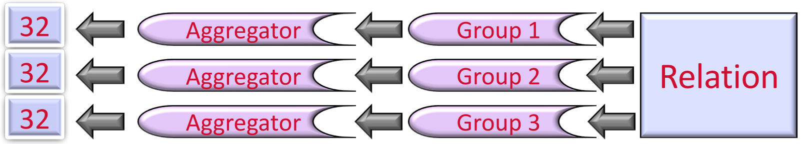
```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  (SELECT  MAX (S2.age)
           FROM  Sailors S2)
           = S.age
```

GROUP BY and HAVING (1)

- **So far, we've applied aggregate operators to all (qualifying) tuples.**



- **Sometimes, we want to apply them to each of several *groups* of tuples.**



GROUP BY and HAVING (2)

<u>**Consider:**</u> ***Find the age of the youngest sailor for each rating level.*** ***/* Min(age) for multiple groups***

- If we know that rating values go from 1 to 10, we can write 10 queries that look like this:

For i = 1, 2, ... , 10:

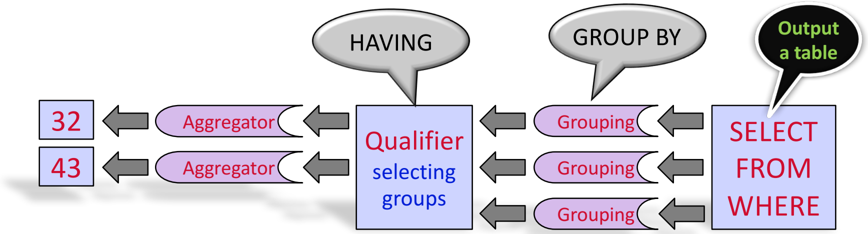
```
SELECT  MIN (S.age)
FROM  Sailors S
WHERE  S.rating = i
```

- In general, we don't know how many rating levels exist, and what the rating values for these levels are !

Queries With GROUP BY and HAVING

```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

MIN(Attribute)



Queries With GROUP BY

- **To generate values for a column based on groups of rows, use aggregate functions in SELECT statements with the GROUP BY clause**

```
SELECT      [DISTINCT] target-list
FROM        relation-list
[WHERE      qualification]
GROUP BY    grouping-list
```

The *target-list* contains

(i) list of column names &

(ii) terms with aggregate operations (e.g., MIN (*S.age*)).

- column name list (i) **can contain only attributes from the *grouping-list*.**

Group By Examples

For each rating, find the average age of the sailors

```
SELECT S.rating, AVG (S.age)
FROM Sailors S
GROUP BY S.rating
```

For each rating find the age of the youngest sailor with age ≥ 18

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
```

Conceptual Evaluation

```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

1. **The cross-product of *relation-list* is computed**
2. **Tuples that fail *qualification* are discarded**
3. **`*Unnecessary'* fields are deleted**
4. **The remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.**
5. **The *group-qualification* is then applied to eliminate some groups**
6. **One answer tuple is generated per qualifying group**

Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 <u>such</u> sailors

```
SELECT    S.rating, MIN (S.age)
FROM      Sailors S
WHERE     S.age >= 18
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

Input relation

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

Answer

rating	age
7	35.0

1	33.0
7	45.0
7	35.0
8	55.5
10	35.0

Only one group satisfies HAVING

Only S.rating and S.age are mentioned in SELECT

"GROUP BY and HAVING" Examples

Find the age of the youngest sailor with age ≥ 18

```
SELECT  MIN (S.age)
FROM  Sailors S
WHERE  S.age >= 18
```

```
SELECT  S.rating, MIN (S.age)
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
```

*Find the age of the youngest sailor with age ≥ 18, **for each rating***

*Find the age of the youngest sailor with age ≥ 18, for each rating **with at least 2 <u>such</u> sailors***

```
SELECT  S.rating, MIN (S.age)
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

For each red boat, find the number of reservations for this boat

3) Count the number of reservations for each red-boat group

```
SELECT    B.bid,  COUNT (*) AS scount
FROM      Boats B, Reserves R
WHERE     R.bid=B.bid AND B.color='red'
GROUP BY  B.bid
```

1) Find all reservations for red boats

2) Group the reservations for red boats according to bid

For each red boat, find the number of reservations for this boat

Grouping over a join of two relations

B.color is not in the grouping-list

```
SELECT    B.bid,  COUNT (*) AS scount
FROM      Boats B, Reserves R
WHERE     R.bid=B.bid
GROUP BY  B.bid
HAVING    B.color='red'
```

Illegal !

<u>Note</u>: HAVING clause is to select groups !

Queries With GROUP BY and HAVING

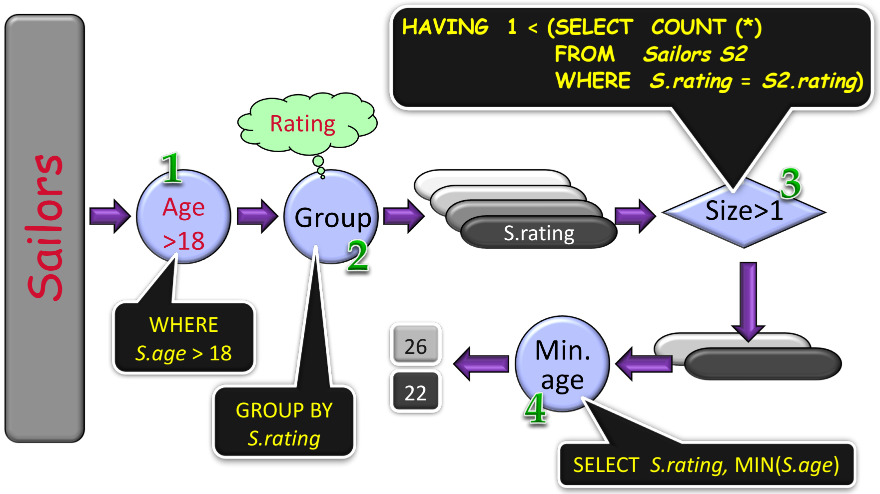
```
SELECT     [DISTINCT]  target-list
FROM       relation-list
WHERE      qualification
GROUP BY   grouping-list
HAVING     group-qualification
```

- **Use the HAVING clause with the GROUP BY clause to restrict which group-rows are returned in the result set**

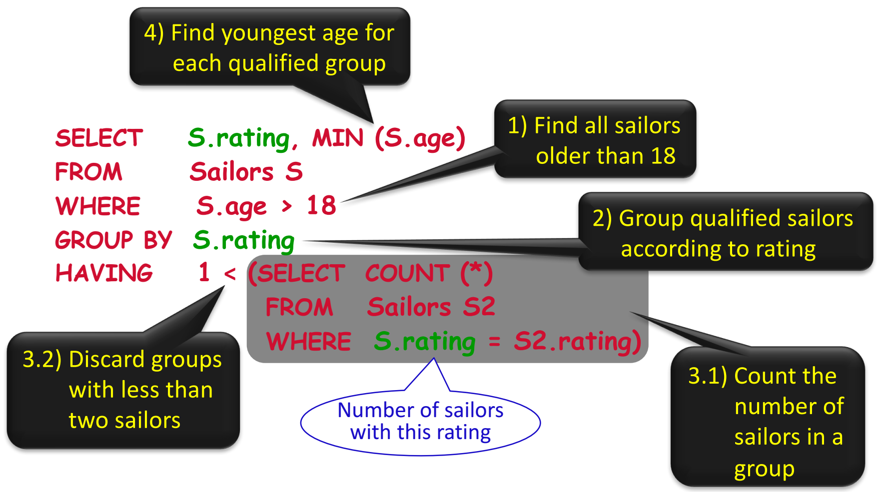
Conceptual Evaluation

- Form groups as before.
- The *group-qualification* is then applied to eliminate some groups.
  - Expressions in *group-qualification* must have a *single value per group*!
  - That is, attributes in *group-qualification* must be arguments of an aggregate op or must also appear in the *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find the age of the youngest sailor older than 18, for each rating with at least 2 sailors



Find the age of the youngest sailor older than 18, for each rating with at least 2 sailors



Find the age of the youngest sailor older than 18, for each rating level that has at least 2 sailors

```
SELECT     S.rating, MIN (S.age)
FROM       Sailors S
WHERE      S.age > 18
GROUP BY   S.rating
HAVING     1 < (SELECT  COUNT (*)
                FROM    Sailors S2
                WHERE   S.rating = S2.rating)
```

Replacing this by "HAVING COUNT(*) > 1"

- **Shows HAVING clause can also contain a subquery.**
- **We can use S.rating inside the nested subquery because it has a single value for the current group of sailors.**
- **What if HAVING clause is replaced by "HAVING COUNT(*) >1"**
  - Find the age of the youngest sailor older than 18, for each rating level that has at least two **such** sailors. /* see next page

Find the age of the youngest sailor older than 18, for each rating level that has at least 2 sailors

```
SELECT     S.rating, MIN (S.age)
FROM       Sailors S
WHERE      S.age > 18
GROUP BY   S.rating
HAVING     1 < (SELECT  COUNT (*)
                FROM    Sailors S2
                WHERE   S.rating = S2.rating)
```

Counting including sailors younger than 18

"age" is not mentioned in this subquery

At least 2 sailors

```
SELECT     S.rating, MIN (S.age)
FROM       Sailors S
WHERE      S.age > 18
GROUP BY   S.rating
HAVING     COUNT (*) > 1
```

Counting include only adult sailors

At least 2 such sailors, i.e., older than 18

Find those ratings for which the average age is the minimum over all ratings

Aggregate operations cannot be nested

```
SELECT S.rating
FROM  Sailors S
WHERE  S.age =  (SELECT  MIN (AVG (S2.age))
                 FROM Sailors S2)
```

Find those ratings for which the average age is the minimum over all ratings

Correct solution (in SQL/92):

Output this rating group and its average age

Find average age for each rating group. This table has two columns

```
SELECT  Temp.rating, Temp.avgage
FROM    (SELECT  S.rating, AVG (S.age) AS avgage
         FROM  Sailors S
         GROUP BY  S.rating) AS Temp
WHERE   Temp.avgage = (SELECT  MIN (Temp.avgage)
                       FROM    Temp)
```

Average age for some rating group

Minimum over all ratings

Null Values

- **Field values in a tuple are sometimes**
  - *unknown* (e.g., a rating has not been assigned), or
  - *inapplicable* (e.g., no spouse's name).
- **SQL provides a special value *null* for such situations.**

Null Values

The presence of *null* complicates many issues:

- **Special operators needed, e.g., IS NULL to test if a value is *null*.**
- **Is *rating>8* true or false when *rating* is equal to *null*? *null***
- **What about AND, OR and NOT ? Need a 3-valued logic (*true, false,* and *unknown*), e.g., (*unknown* OR *false*) = *unknown*.**
- **Meaning of constructs must be defined carefully, e.g., WHERE clause eliminates rows that don't evaluate to true.**
  - Null + 5 = null; but SUM (null, 5) = 5. (nulls can cause some unexpected behavior)
- **New operators (in particular, *outer joins*) possible/needed.**

Views

```
CREATE VIEW view_name
AS select_statement
```

Makes development simpler
Often used for security
Not instantiated - makes updates tricky

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
   FROM Boats B, Reserves R
   WHERE  R.bid=B.bid AND  B.color='red'
    GROUP BY  B.bid
```


```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
   FROM Boats B, Reserves R
   WHERE  R.bid=B.bid AND  B.color='red'
    GROUP BY  B.bid
```


b.bid	scount
102	1

Reds

Alternative: Division using Group By

Find names of sailors who've reserved all boats.

- **Can you do this using Group By and Having?**

```
SELECT  S.name
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
 GROUP BY S.name, S.sid
 HAVING
        COUNT(DISTINCT R.bid) =
              ( Select COUNT (*) FROM Boats)
```

Note: must have both sid and name in the GROUP BY clause. Why?

Sailors who have reserved all boats

```
SELECT  S.name
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
GROUP BY S.name, S.sid
HAVING COUNT(DISTINCT R.bid) =
        ( Select COUNT (*) FROM Boats)
```

sname	sid	bid
Frodo	1	102
Bilbo	2	101
Bilbo	2	102
Frodo	1	102
Bilbo	2	103

sname	sid	bid
Frodo	1	102,102
Bilbo	2	101, 102, 103

sname	sid	count
Frodo	1	1
Bilbo	2	3

count
3

Sailors

sid	sname	rating	age
1	Frodo	7	22
2	Bilbo	2	39
3	Sam	8	27

Boats

bid	bname	color
101	Nina	red
102	Pinta	blue
103	Santa Maria	red

Reserves

sid	bid	day
1	102	9/12
2	102	9/12
2	101	9/14
1	102	9/10
2	103	9/13

Two more important topics

- **Constraints**
- **SQL embedded in other languages**

Integrity Constraints

- **IC conditions that every <u>legal</u> instance of a relation must satisfy.**
  - Inserts/deletes/updates that violate ICs are disallowed.
  - Can ensure application semantics (e.g., sid is a key),
  - …or prevent inconsistencies (e.g., sname has to be a string, age must be < 200)
- **Types of IC's: Domain constraints, primary key constraints, foreign key constraints, general constraints.**
  - Domain constraints: Field values must be of right type. Always enforced.
  - Primary key and foreign key constraints: coming right up.

Where do ICs Come From?

- **Semantics of the real world!**
- **Note:**
  - We can check IC violation in a DB instance
  - We can NEVER infer that an IC is true by looking at an instance.
    - An IC is a statement about all possible instances!
  - From example, we know name is not a key, but the assertion that sid is a key is given to us.
- **Key and foreign key ICs are the most common**
- **More general ICs supported too.**

Keys

- **Keys are a way to associate tuples in different relations**
- **Keys are one form of IC**

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

FOREIGN Key

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

PRIMARY Key

Primary Keys

- **A set of fields is a superkey if:**
  - No two distinct tuples can have same values in all key fields
- **A set of fields is a key for a relation if :**
  - It is a superkey
  - No subset of the fields is a superkey
- **what if >1 key for a relation?**
  - One of the keys is chosen (by DBA) to be the primary key. Other keys are called candidate keys.
- **E.g.**
  - sid is a key for Students.
  - What about name?
  - The set {sid, gpa} is a superkey.

Primary and Candidate Keys

- **Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.**

- Keys must be used carefully!

```
CREATE TABLE Enrolled1
  (sid CHAR(20),
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid))
```

vs.

```
CREATE TABLE Enrolled2
   (sid CHAR(20),
    cid  CHAR(20),
    grade CHAR(2),
    PRIMARY KEY  (sid),
    UNIQUE (cid, grade))
```

"For a given student and course, there is a single grade."

Primary and Candidate Keys

```
CREATE TABLE Enrolled1
  (sid CHAR(20),
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid))
```

vs.

```
CREATE TABLE Enrolled2
  (sid CHAR(20),
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade))
```

```
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled1 VALUES ('1234', 'cs61C', 'A+');
```

```
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled2 VALUES ('1234', 'cs61C', 'A+');
INSERT INTO enrolled2 VALUES ('4567', 'cs186', 'A+');
```

"For a given student and course, there is a single grade."

Primary and Candidate Keys

```
CREATE TABLE Enrolled1
  (sid CHAR(20),
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid));
```

vs.

```
CREATE TABLE Enrolled2
  (sid CHAR(20),
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade));
```

```
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled1 VALUES ('1234', 'cs61C', 'A+');
```

```
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled2 VALUES ('1234', 'cs61C', 'A+');
INSERT INTO enrolled2 VALUES ('4567', 'cs186', 'A+');
```

"Students can take only one course, and no two students in a course receive the same grade."

Foreign Keys, Referential Integrity

- ***Foreign key***: **a "logical pointer"**
  - Set of fields in a tuple in one relation that `refer' to a tuple in another relation.
  - Reference to *primary key* of the other relation.
- **All foreign key constraints enforced?**
  - *referential integrity*!
  - i.e., no dangling references.

Foreign Keys in SQL

- **E.g. Only students listed in the Students relation should be allowed to enroll for courses.**
  - *sid* is a foreign key referring to Students:

```
CREATE TABLE Enrolled
(sid CHAR(20),cid CHAR(20),grade CHAR(2),
 PRIMARY KEY (sid,cid),
 FOREIGN KEY (sid) REFERENCES Students);
```

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B
~~11111~~	~~English102~~	~~A~~

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Enforcing Referential Integrity

- ***sid*** **in Enrolled: foreign key referencing Students.**
- **Scenarios:**
  - Insert Enrolled tuple with non-existent student id?
  - Delete a Students tuple?
    - Also delete Enrolled tuples that refer to it? (CASCADE)
    - Disallow if referred to? (NO ACTION)
    - Set sid in referring Enrolled tups to a *default* value? (SET DEFAULT)
    - Set sid in referring Enrolled tuples to *null*, denoting `*unknown*' or `*inapplicable*'. (SET NULL)
- **Similar issues arise if primary key of Students tuple is updated.**

General Constraints

- **Useful when more general ICs than keys are involved.**
- **Can use queries to express constraint.**
- **Checked on insert or update.**
- **Constraints can be named.**

```
CREATE TABLE  Sailors
        ( sid  INTEGER,
        sname  CHAR(10),
        rating  INTEGER,
        age  REAL,
        PRIMARY KEY  (sid),
        CHECK  ( rating >= 1
                AND rating <= 10 ))
```

```
CREATE TABLE  Reserves
        ( sname  CHAR(10),
        bid  ~~INTEGER,~~
        day  DATE,
        PRIMARY KEY  (bid,day),
        CONSTRAINT  noInterlakeRes
        CHECK  ('Interlake' <>
                ( SELECT  B.bname
                FROM  Boats B
                WHERE  B.bid=bid)))
```


Constraints Over Multiple Relations

```
CREATE TABLE  Sailors
        ( sid  INTEGER,
        sname  CHAR(10),
        rating  INTEGER,
        age  REAL,
        PRIMARY KEY  (sid),
        CHECK
        ( (SELECT COUNT (S.sid) FROM Sailors S)
        + (SELECT COUNT (B.bid) FROM
                Boats B) < 100 )
```

Number of boats plus number of sailors is < 100

Constraints Over Multiple Relations

- Awkward and wrong!
  - Only checks sailors!
- ASSERTION is the right solution; not associated with either table.
  - Unfortunately, not supported in many DBMS.
  - *Triggers* are another solution.

```
CREATE TABLE  Sailors
        ( sid  INTEGER,
        sname  CHAR(10),
        rating  INTEGER,
        age  REAL,
        PRIMARY KEY  (sid),
        CHECK
        ( (SELECT COUNT (S.sid) FROM Sailors S)
        + (SELECT COUNT (B.bid) FROM
                Boats B) < 100 )
```

Number of boats plus number of sailors is < 100

```
CREATE ASSERTION  smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid)
 FROM Boats B) < 100 )
```

Two more important topics

- ~~Constraints~~
- **SQL embedded in other languages**

Writing Applications with SQL

- **SQL is not a general purpose programming language.**
  - + Tailored for data retrieval and manipulation
  - + Relatively easy to optimize and parallelize
  - Can't write entire apps in SQL alone
- **Options:**
  - Make the query language "Turing complete"
    - Avoids the "impedance mismatch"
    - makes "simple" relational language complex
  - Allow SQL to be embedded in regular programming languages.
  - Q: What needs to be solved to make the latter approach work?

Cursors

- **Can declare a cursor on a relation or query**
- **Can *open* a cursor**
- **Can repeatedly *fetch* a tuple (moving the cursor)**
- **Special return value when all tuples have been retrieved.**
- **ORDER BY allows control over the order tuples are returned.**
    - Fields in ORDER BY clause must also appear in SELECT clause.
- **LIMIT controls the number of rows returned (good fit w/ORDER BY)**
- **Can also modify/delete tuple pointed to by a cursor**
  - A "non-relational" way to get a handle to a particular tuple

Database APIs

- **A library with database calls (API)**
  - special objects/methods
  - passes SQL strings from language, presents result sets in a language-friendly way
  - *ODBC* a C/C++ standard started on Windows
  - *JDBC* a Java equivalent
  - Most scripting languages have similar things
    - E.g. in Ruby there's the "pg" gem for Postgres
- **ODBC/JCDB try to be DBMS-neutral**
  - at least try to hide distinctions across different DBMSs

Summary

- Relational model has well-defined query semantics
- SQL provides functionality close to basic relational model

  (some differences in duplicate handling, null values, set operators, …)
- Typically, many ways to write a query
  - DBMS figures out a fast way to execute a query, regardless of how it is written.

ADVANCED EXAMPLES

Getting Serious

- **Two “fancy” queries for different applications**
  - Clustering Coefficient for Social Network graphs
  - Medians for “robust” estimates of the central value

Serious SQL: Social Nets Example

```
-- An undirected friend graph. Store each link once
CREATE TABLE Friends(
     fromID integer,
     toID integer,
     since date,
     PRIMARY KEY (fromID, toID),
     FOREIGN KEY (fromID) REFERENCES Users,
     FOREIGN KEY (toID) REFERENCES Users,
     CHECK (fromID < toID));

-- Return both directions
CREATE VIEW BothFriends AS
   SELECT * FROM Friends
   UNION ALL
   SELECT F.toID AS fromID, F.fromID AS toID, F.since
   FROM Friends F;
```

6 degrees of friends

```
SELECT F1.fromID, F5.toID
  FROM BothFriends F1, BothFriends F2, BothFriends F3,
       BothFriends F4, BothFriends F5
 WHERE F1.toID = F2.fromID
   AND F2.toID = F3.fromID
   AND F3.toID = F4.fromID
   AND F4.toID = F5.fromID;
```

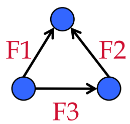
Clustering Coefficient of a Node

$$C_i = 2|\{e_{jk}\}| / k_i(k_i\text{-}1)$$

- where:
  - k_i is the number of neighbors of node I
  - e_{jk} is an edge between nodes j and k neighbors of i, ($j < k$). (A triangle!)
- I.e. Cliquishness: the fraction of your friends that are friends with each other!
- Clustering Coefficient of a graph is the average CC of all nodes.

In SQL

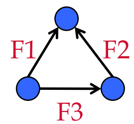
$$C_i = 2|\{e_{jk}\}| / k_i(k_i\text{-}1)$$




```
CREATE VIEW NEIGHBOR_CNT AS
SELECT fromID AS nodeID, count(*) AS friend_cnt
  FROM BothFriends
 GROUP BY nodeID;

CREATE VIEW TRIANGLES AS
SELECT F1.toID as root, F1.fromID AS friend1,
       F2.fromID AS friend2
  FROM BothFriends F1, BothFriends F2, Friends F3
 WHERE F1.toID = F2.toID     /* Both point to root */
   AND F1.fromID = F3.fromID /* Same origin as F1  */
   AND F3.toID = F2.fromID   /* points to origin of F2 */
  ;
```

In SQL



$$C_i = 2|\{e_{jk}\}| / k_i(k_i\text{-}1)$$

```
CREATE VIEW NEIGHBOR_EDGE_CNT AS
SELECT root, COUNT(*) as cnt FROM TRIANGLES
 GROUP BY root;

CREATE VIEW CC_PER_NODE AS
SELECT NE.root, 2.0*NE.cnt /
                 (N.friend_cnt*(N.friend_cnt-1)) AS CC
  FROM NEIGHBOR_EDGE_CNT NE, NEIGHBOR_CNT N
 WHERE NE.root = N.nodeID;

SELECT AVG(cc) FROM CC_PER_NODE;
```

Median

- **Given n values in sorted order, the one at position n/2**
  - Assumes an odd # of items
  - For an even #, can take the lower of the middle 2
- **A much more "robust" statistic than average**
  - Q: Suppose you want the mean to be 1,000,000. What fraction of values do you have to corrupt?
  - Q2: Suppose you want the median to be 1,000,000. Same question.
  - This is called the *breakdown point* of a statistic.
  - Important for dealing with data *outliers*
    - E.g. dirty data
    - Even with real data: "overfitting"

Median in SQL

```
SELECT c AS median FROM T
 WHERE
 (SELECT COUNT(*) from T AS T1
   WHERE T1.c < T.c)
 =
 (SELECT COUNT(*) from T AS T2
   WHERE T2.c > T.c);
```


Median in SQL

```
SELECT c AS median FROM T
 WHERE
 (SELECT COUNT(*) from T AS T1
   WHERE T1.c < T.c)
 =
 (SELECT COUNT(*) from T AS T2
   WHERE T2.c > T.c);
```


Faster Median in SQL

```
SELECT x.c as median
  FROM T x, T y
 GROUP BY x.c
HAVING
 SUM(CASE WHEN y.c <= x.c THEN 1 ELSE 0 END)
   >= (COUNT(*)+1)/2
AND
 SUM(CASE WHEN y.c >= x.c THEN 1 ELSE 0 END)
   >= (COUNT(*)/2)+1
```

Why faster?
Note: handles even # of items!

APPENDIX

DDL – Create Table

- **CREATE TABLE *table_name*
  ({ *column_name data_type*
  [DEFAULT *default_expr*] [
  column_constraint [, ...]] |
  table_constraint } [, ...])**

- **Data Types (PostgreSQL) include:**
  character(n) – fixed-length character string
  character varying(n) – variable-length character string
  smallint, integer, bigint, numeric, real, double precision
  date, time, timestamp, …
  serial - unique ID for indexing and cross reference
  …
- **PostgreSQL also allows OIDs and other "system types", arrays, inheritance, rules…**
  conformance to the SQL-1999 standard is variable.

Constraints

- **Recall that the schema defines the legal instances of the relations.**
- **Data types are a way to limit the kind of data that can be stored in a table, but they are often insufficient.**
  - e.g., prices must be positive values
  - uniqueness, referential integrity, etc.
- **Can specify constraints on individual columns or on tables.**

Column constraints

**[CONSTRAINT *constraint_name*]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
CHECK (*expression*) |
REFERENCES *reftable* [(*refcolumn*)] [ON DELETE *action*] [ON UPDATE *action*] }**

primary key = *unique + not null; also used as default target for references. (can have at most 1)*

***expression* must produce a boolean result and reference that column's value only.**

***references* is for foreign keys; *action* is one of:
NO ACTION, CASCADE, SET NULL, SET DEFAULT**

Table constraints

- **CREATE TABLE *table_name***
  ({ *column_name data_type* [DEFAULT *default_expr*] [*column_constraint* [, ...]] | *table_constraint* } [, ...])

Table Constraints:

- **[CONSTRAINT *constraint_name*]**
  { UNIQUE (*column_name* [, ...]) |
  PRIMARY KEY (*column_name* [, ...]) |
  CHECK (*expression*) |
  FOREIGN KEY (*column_name* [, ...]) REFERENCES *reftable* [(*refcolumn* [, ...])] [ON DELETE *action*] [ON UPDATE *action*] }

Other DDL Statements

- **Alter Table**
  - use to add/remove columns, constraints, rename things …
- **Drop Table**
  - Compare to "Delete * From Table"
- **Create/Drop View**
- **Create/Drop Index**
- **Grant/Revoke privileges**
  - SQL has an authorization model for saying who can read/modify/delete etc. data and who can grant and revoke privileges!

Create Table (Examples)

```
CREATE TABLE films (
    code        CHAR(5) PRIMARY KEY,
    title       VARCHAR(40),
    did         DECIMAL(3),
    date_prod   DATE,
    kind        VARCHAR(10),
CONSTRAINT production UNIQUE(date_prod)
FOREIGN KEY did REFERENCES distributors
   ON DELETE NO ACTION
);
CREATE TABLE distributors (
    did    DECIMAL(3) PRIMARY KEY,
    name   VARCHAR(40)
    CONSTRAINT con1 CHECK (did > 100 AND name <> ' ')
);
```