# SQL: The Query Language (part 2 of 2)

R & G - Chapter 5

## Basic Single-Table Queries

```
SELECT [DISTINCT] <column expression list>
  FROM <single table>
[WHERE <predicate>]
[GROUP BY <column list>
 [HAVING <predicate>] ]
[ORDER BY <column list>];
```

## Null Values

- Field values are sometimes unknown or inapplicable
  - SQL provides a special value null for such situations.
- The presence of null complicates many issues. E.g.:
  - Special syntax "IS NULL" and "IS NOT NULL"
  - Assume rating = NULL. Consider predicate "rating>8".
    - True? False? (answer is always *false*)
    - What about AND, OR and NOT connectives?
    - SUM?
  - We need a 3-valued logic (true, false and unknown).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, outer joins) possible/needed.

## Joins

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

INNER is default

## Inner/Natural Joins

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s, Reserves r
WHERE s.sid = r.sid
```

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

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

all 3 are equivalent!

"NATURAL" means equi-join for each pair of attributes with the same name

---

```
SELECT s.sid, s.sname, r.bid
FROM Sailors2 s INNER JOIN Reserves2 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 |

# 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.sname, r.bid
FROM Sailors2 s LEFT OUTER JOIN Reserves2 r
ON s.sid = r.sid;

Returns all sailors & bid for boat in any of their reservations
Note: no match for s.sid? => r.bid is NULL

SELECT s.sid, s.name, r.bid
FROM Sailors2 s LEFT OUTER JOIN Reserves2 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 |
| 31 | Lubber | |

# Right Outer Join

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.bname
FROM Reserves2 r RIGHT OUTER JOIN Boats2 b
ON r.bid = b.bid;

Returns all boats & information on which are reserved

No match for b.bid? => r.sid is NULL

SELECT r.sid, b.bid, b.bname
FROM Reserves2 r RIGHT OUTER JOIN Boats2 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 |

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

# 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.bname
FROM Reserves2 r FULL OUTER JOIN Boats2 b
ON r.bid = b.bid

- Returns all boats & all reservations
- No match for r.bid?
  - b.bid is NULL AND b.bname is NULL
- No match for b.bid?
  - r.sid is NULL

SELECT r.sid, b.bid, b.name
FROM Reserves2 r FULL OUTER JOIN Boats2 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 |

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

Note: in this case it is the same as the ROJ!
*bid* is a foreign key in reserves, so all reservations must have a corresponding tuple in boats

## Views: Named Queries

CREATE VIEW *view_name*
AS *select_statement*

Makes development simpler
Often used for security
(not "materialized")

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats2 B, Reserves2 R
    WHERE  R.bid=B.bid AND   B.color='red'
    GROUP BY  B.bid
```

## Views Instead of Relations in Queries

```
CREATE VIEW Redcount
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats2 B, Reserves2 R
    WHERE  R.bid=B.bid AND   B.color='red'
    GROUP BY  B.bid
```

| bid | scount |
| --- | --- |
| 102 | 1 |

Reds

```
SELECT  bname, scount
    FROM Redcount R, Boats2 B
    WHERE  R.bid=B.bid
        AND scount < 10
```

## Subqueries in FROM

```
SELECT  bname, scount
  FROM Boats2 B,
         (SELECT B.bid,  COUNT (*)
            FROM Boats2 B, Reserves2 R
            WHERE  R.bid=B.bid AND   B.color='red'
         GROUP BY  B.bid) AS Reds(bid, scount)
  WHERE  Reds.bid=B.bid
     AND scount < 10
```

## WITH (common table expression)

```
WITH Reds(bid, scount) AS
(SELECT B.bid,  COUNT (*)
          FROM Boats2 B, Reserves2 R
          WHERE  R.bid=B.bid AND   B.color='red'
     GROUP BY  B.bid)
SELECT  bname, scount
  FROM Boats2 B, Reds
    WHERE  Reds.bid=B.bid
       AND scount < 10
```

## Discretionary Access Control

GRANT *privileges* ON *object* TO *users*
[WITH GRANT OPTION]

- Object can be a Table or a View
- Privileges can be:
  - Select
  - Insert
  - Delete
  - References (cols) – allow to create a foreign key that references the specified column(s)
  - All
- Can later be REVOKEd
- Users can be single users or groups
- See Chapter 17 for more details.

## Two more important topics

- Constraints
- SQL embedded in other languages

## Integrity Constraints

- IC conditions that every legal 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 *minimal*:
  - 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/JDBC try to be DBMS-neutral
  - at least try to hide distinctions across different DBMSs
- Object-Relational Mappings (ORMs)
  - Ruby on Rails, Django, Spring, BackboneORM, etc.
    - Automagically map database rows into PL objects
    - Magic can be great; magic can bite you.
  - This year we won't cover ORMs much – see CS169.

## 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.

BACKUP MATERIAL

## 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;
```

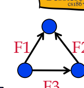
## Clustering Coefficient of a Node

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

- 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

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



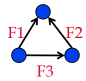

```
CREATE VIEW NEIGHBOR_CNT AS
SELECT [redacted]
  FROM
 GROUP

CREATE VIEW TRIANGLES AS
SELECT [redacted]

  FROM
 WHERE
   AND
   AND
  ;
```

## In SQL

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



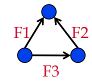

```
CREATE VIEW NEIGHBOR_CNT AS
SELECT fromID AS nodeID, count(*) AS friend_cnt
  FROM BothFriends
 GROUP BY nodeID;

CREATE VIEW TRIANGLES AS
SELECT

  FROM
 WHERE
   AND
   AND
  ;
```

## In SQL

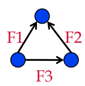
$C_i = 2|\{e_{jk}\}| / k_i(k_i-1)$




```
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




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

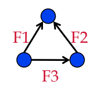
```
CREATE VIEW NEIGHBOR_EDGE_CNT AS
SELECT
 GROUP

CREATE VIEW CC_PER_NODE AS
SELECT

  FROM
 WHERE

SELECT AVG(cc) FROM CC_PER_NODE;
```

## In SQL




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

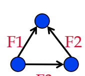
```
CREATE VIEW NEIGHBOR_EDGE_CNT AS
SELECT root, COUNT(*) as cnt FROM TRIANGLES
 GROUP BY root;

CREATE VIEW CC_PER_NODE AS
SELECT

  FROM
 WHERE

SELECT AVG(cc) FROM CC_PER_NODE;
```

## In SQL




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

```
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
```

## Median in SQL

```
SELECT c AS median FROM T
 WHERE
 =
```

## Median in SQL

```
SELECT c AS median FROM T
 WHERE
 (SELECT COUNT(*) from T AS T1
   WHERE T1.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!

## Using “Window Functions”

Window functions: an SQL idiom to compute with order.
http://www.postgresql.org/docs/9.3/static/tutorial-window.html

```
CREATE VIEW twocounters AS
(SELECT x,
        ROW_NUMBER() OVER (ORDER BY x ASC) AS RowAsc,
        ROW_NUMBER() OVER (ORDER BY x DESC) AS RowDesc
   FROM numbers
);

SELECT AVG(x)
FROM twocounters
WHERE RowAsc IN (RowDesc, RowDesc - 1, RowDesc + 1);
```

O(n logn!)
Note: handles even # of items.

## Notes for Studying

- You'll be responsible for all the constructs we mentioned, except
  - Window functions
  - Programming Language APIs
- In HW3 you may write queries using:
  - Any PostgreSQL features you like
  - Except for callouts to user-defined code (C, Java, Python, R, etc.)