CS150: Database & Datamining

Lecture 4: SQL Part III

Xuming He

Spring 2019

Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.

Today's Lecture

1. Advanced SQL-izing
    - Views
    - Access Control
    - General Constraints
2. Two real-world examples
    - Social network
    - Statistical estimation

1. Advanced SQL-izing

Example Database

Sailors

sid	sname	rating	age
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

Boats

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

Reserves

sid	bid	day
1	102	9/12/2015
2	102	9/13/2015

The SQL DDL

```
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);
```

sid	sname	rating	age
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

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

sid	**bid**	**day**
1	102	9/12
2	102	9/13

Warm up SQL

Find sailors who've reserved all boats.

(relational division: no "counterexample boats")

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS ( 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 ))
```

Sailors S such that ...

there is no boat B that...

...is missing a Reserves tuple showing S reserved B

ARGMAX?

- The sailor with the highest rating
  - Give a try

```
SELECT MAX(S.rating)
  FROM Sailors S; -- OK

SELECT S.*, MAX(S.rating)
FROM   Sailors S; -- Not OK
```

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;
```

What you will learn about in this section

1. Views
2. Access Control & Constraints
3. General Constraints

Views

Views: Named Queries

```
CREATE VIEW  view_name
AS select_statement
```

Makes development simpler
Often used for security
Not “materialized”

```
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;
```

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

Redcount

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

Subqueries in FROM

Like a “view on the fly”!

```
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)

Another "view on the fly" syntax:

```
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
```

Access Control & Constraints

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

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

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

FOREIGN Key

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."

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 )
```

2. Two real-world examples

What you will learn about in this section

1. Social network graph

2. Statistical estimation

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

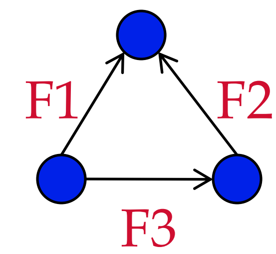
$$C_i = 2|\{e_{jk}\}| / k_i(k_i\text{-}1)$$

- 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

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



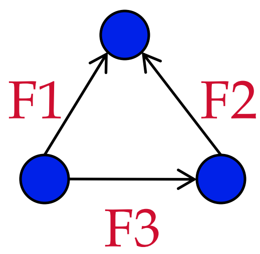

```
CREATE VIEW NEIGHBOR_CNT AS
SELECT [redacted]
  FROM
 GROUP

CREATE VIEW TRIANGLES AS
SELECT [redacted]

  FROM
 WHERE
   AND
   AND
  ;
```

In SQL

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



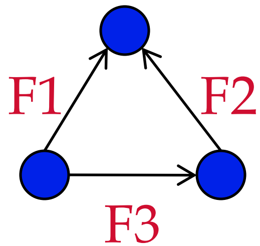

```
CREATE VIEW NEIGHBOR_CNT AS
SELECT fromID AS nodeID, count(*) AS friend_cnt
  FROM BothFriends
 GROUP BY nodeID;

CREATE VIEW TRIANGLES AS
SELECT

  FROM
 WHERE
   AND
   AND
  ;
```

In SQL

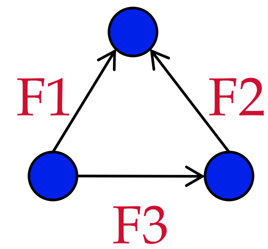
$$C_i = 2|\{e_{jk}\}| / k_i(k_i-1)$$




```
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



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

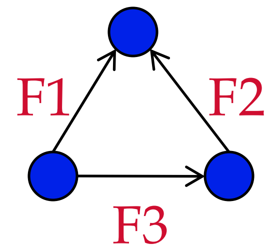
```
CREATE VIEW NEIGHBOR_EDGE_CNT AS
SELECT [redacted]
 GROUP [redacted]

CREATE VIEW CC_PER_NODE AS
SELECT [redacted]

  FROM [redacted]
 WHERE [redacted]

SELECT AVG(cc) FROM CC_PER_NODE;
```

In SQL



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

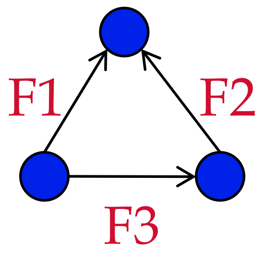
```
CREATE VIEW NEIGHBOR_EDGE_CNT AS
SELECT root, COUNT(*) as cnt FROM TRIANGLES
 GROUP BY root;

CREATE VIEW CC_PER_NODE AS
SELECT [redacted]

  FROM
 WHERE

SELECT AVG(cc) FROM CC_PER_NODE;
```

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
```

Median in SQL

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

[redacted]

```
 =
```

[redacted]

Median in SQL

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

Median in SQL (odd cardinality)

```
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 (odd cardinality)

```
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 -- ceiling(N/2)
AND
 SUM(CASE WHEN y.c >= x.c THEN 1 ELSE 0 END)
   >= (COUNT(*)/2)+1 -- floor(N/2) +1
```

Summary

- Advanced SQL
- Real-world examples of SQL queries