SQL

The Query Language

R & G - Chapter 5

Query Execution

Declarative Query (SQL)
Query Optimization and Execution
(Relational) Operators
File and Access Methods
Buffer Management
Disk Space Management

← We start from here

SQL: THE query language

- Developed @IBM Research in the 1970s
  - System R project
  - Vs. Berkeley's Quel language (Ingres project)
- Commercialized/Popularized in the 1980s
  - IBM beaten to market by a startup called Oracle
- Questioned repeatedly
  - 90's: OO-DBMS (OQL, etc.)
  - 2000's: XML (XQuery, Xpath, XSLT)
  - 2010's: NoSQL & MapReduce
- SQL keeps re-emerging as the standard
  - Even Hadoop, Spark etc. see lots of SQL
  - May not be perfect, but it is useful

SQL Pros and Cons

- Declarative!
  - Say **what** you want, not **how** to get it
- Implemented widely
  - With varying levels of efficiency, completeness
- Constrained
  - Core SQL is not a Turing-complete language
  - Extensions make it Turing complete
- General-purpose and feature-rich
  - many years of added features
  - extensible: callouts to other languages, data sources

Relational Terminology

- **Database**: Set of **Relations**
- **Relation** (**Table**):
  - **Schema** (description)
  - **Instance** (data satisfying the schema)
- **Attribute** (**Column**)
- **Tuple** (**Record**, **Row**)

- Also: schema of database is set of schemas of its relations

Relational Tables

- *Schema* is fixed:
  - attribute names, atomic types
  - `students(name text, gpa float, dept text)`
- *Instance* can change
  - a *multiset* of "rows" ("tuples")
  - ```
    {('Bob Snob', 3.3, 'CS'),
     ('Bob Snob', 3.3, 'CS'),
     ('Mary Contrary', 3.8, 'CS')}
    ```
    ```

SQL Language

- Two sublanguages:
  - DDL – Data Definition Language
    - Define and modify schema
  - DML – Data Manipulation Language
    - Queries can be written intuitively.

- RDBMS responsible for efficient evaluation.
  - Choose and run algorithms for declarative queries
    - Choice of algorithm must not affect query answer.

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(sid),
  FOREIGN KEY (bid) REFERENCES
   Boats(bid));
```

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

The SQL DML

Sailors

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

- Find all 27-year-old sailors:

```
SELECT *
FROM Sailors AS S
WHERE S.age = 27;
```

- To find just names and ratings, replace the first line:

```
SELECT S.sname, S.rating
FROM Sailors AS S
WHERE S.age = 27;
```

SQL: DDL

DDL – Create Table

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

- Data Types (mySQL) include:
  character(n) – fixed-length character string
  character varying(n) – variable-length character string
  binary(n), text(n), blob, mediumblob, mediumtext,

  smallint, integer, bigint, numeric, real, double precision

  date, time, timestamp, …
  serial - unique ID for indexing and cross reference
  =>

- http://dev.mysql.com/doc/refman/5.7/en/data-types.html

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.

Constraints

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.

Primary Keys

- A set of fields is a superkey if:
  - No two distinct tuples can have same values in all these 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.
- For example:
  - 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))
```

Not good either!

```
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."

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

- For example, 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(sid));
```

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.

Foreign keys actions

```
CREATE TABLE Enrolled
(sid CHAR(20), cid CHAR(20), grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid) REFERENCES Students(sid)
      ON DELETE NO ACTION  );
```

vs

```
FOREIGN KEY (sid) REFERENCES Students(sid)
      ON DELETE CASCADE);
```

vs

```
FOREIGN KEY (sid) REFERENCES Students(sid)
      ON DELETE SET NULL);
```

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 )
```

Other DDL Statements

- Alter Table
  - use to add/remove columns, constraints, rename things …
- Drop Table
  - Compare to “Delete * From Table” next
- 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!

SQL: Modification Commands

Deletion:

DELETE FROM <relation>
[WHERE <predicate>]

Example:

account(bname, acct_no, balance)

```
1.  DELETE FROM account
    -- deletes all tuples in account
```

```
2.  DELETE FROM account
    WHERE bname IN (SELECT bname
                    FROM   branch
                    WHERE bcity = 'Bkln')
   -- deletes all accounts from Brooklyn branch
```

DELETE

- Delete the record of all accounts with balances below the average at the bank.

```
DELETE FROM account
WHERE balance < (SELECT AVG(balance)
                 FROM account)
```

  - Problem: as we delete tuples from *deposit,* the average balance changes
  - Solution used in SQL:
  - 1. First, compute **avg** balance and find all tuples to delete
  - 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

SQL: Modification Commands

Insertion: INSERT INTO <relation> values (.., .., ...)

or INSERT INTO <relation>(att1, .., attn)
values(..., ..., ...)

or INSERT INTO <relation> <query expression>

account(bname, acct_no, balance)

Examples:

INSERT INTO account VALUES ('Perry' , A-768, 1200)

or INSERT INTO account(bname, acct_no, balance)
VALUES ('Perry' , A-768, 1200)

```
INSERT INTO account
      SELECT   bname, lno, 200
      FROM     loan
      WHERE    bname = 'Kenmore'
```

gives free $200 savings account for each loan holder at Kenmore

SQL: Modification Commands

Update: UPDATE <relation>
SET <attribute> = <expression>
WHERE <predicate>

Ex.

```
UPDATE  account
SET       balance = balance * 1.06
WHERE   balance > 10000

UPDATE account
SET       balance = balance * 1.05
WHERE   balance <= 10000
```

Alternative:

```
UPDATE account
SET      balance =
         (CASE
    WHEN balance <= 10000 THEN balance*1.05
             ELSE  balance*1.06
            END)
```

Single Relation Queries

SQL DML 1: Basic Single-Table Queries

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

Basic Single-Table Queries

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

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
  - Expression can be a column reference, or an arithmetic expression over column refs
  ```

Basic Single-Table Queries

- ```
  SELECT          S.name, S.gpa
    FROM students S
   WHERE S.dept = 'CS'
  ```

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
  - Expression can be a column reference, or an arithmetic expression over column refs
  ```

Basic Single-Table Queries

- ```
  SELECT DISTINCT S.name, S.gpa
    FROM students S
   WHERE S.dept = 'CS';
  ```

- DISTINCT flag specifies removal of duplicates before output
  ```

ORDER BY

- ```
  SELECT DISTINCT S.name, S.gpa, S.age*2 as a2
    FROM students S
   WHERE S.dept = 'CS'
  ORDER BY S.gpa, S.name, a2;
  ```

- ORDER BY clause specifies output to be sorted
  - **Lexicographic** ordering
- Obviously must refer to columns in the output
  - Note the AS clause for naming output columns
  ```

ORDER BY

- ```sql
  SELECT DISTINCT S.name, S.gpa
    FROM students S
   WHERE S.dept = 'CS'
  ORDER BY S.gpa DESC, S.name ASC;
  ```

- Ascending order by default, but can be overridden
  - DESC flag for descending, ASC for ascending
  - Can mix and match, lexicographically
  ```

Aggregates

- ```
  SELECT AVG(S.gpa)
    FROM students S
   WHERE S.dept = 'CS'
  ```

- Before producing output, compute a summary (a.k.a. an aggregate) of some arithmetic expression
- Produces 1 row of output
  - with one column in this case
- Other aggregates: SUM, COUNT, MAX, MIN
- Note: can use DISTINCT inside the agg function
  - SELECT COUNT(DISTINCT S.name) FROM Students S
  - vs. SELECT DISTINCT COUNT (S.name) FROM Students S;
  ```

DELETE

- Delete the record of all accounts with balances below the average at the bank.

```
DELETE FROM account
WHERE balance < (SELECT AVG(balance)
                 FROM account)
```

  – Problem: as we delete tuples from *deposit,* the average balance changes

Solution used in SQL:

- 1. First, compute **avg** balance and find all tuples to delete
- 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

GROUP BY

- ```
  SELECT [DISTINCT] AVG(S.gpa), S.dept
    FROM students S
  [WHERE <predicate>]
  GROUP BY S.dept
   [HAVING <predicate>]
   [ORDER BY <column list>] ;
  ```

- Partition table into groups with same GROUP BY column values
  - Can group by a list of columns
- Produce an aggregate result per group
  - Cardinality of output = # of distinct group values
- Note: can put grouping columns in SELECT list
  - For aggregate queries, SELECT list can contain aggs and GROUP BY columns only!
  - What would it mean if we said SELECT S.name, AVG(S.gpa) above??
  ```

HAVING

```
SELECT [DISTINCT] AVG(S.gpa), S.dept
  FROM students S
[WHERE <predicate>]
 GROUP BY S.dept
 HAVING COUNT(*) > 5
 [ORDER BY <column list>] ;
```

- The HAVING predicate is applied after grouping and aggregation
  - Hence can contain anything that could go in the SELECT list
  - That is, aggs or GROUP BY columns
- HAVING can only be used in aggregate queries
- It’s an optional clause

Putting it all together

- ```
  SELECT S.dept, AVG(S.gpa), COUNT(*)
    FROM students S
   WHERE S.gender = 'F'
   GROUP BY S.dept
   HAVING COUNT(*) > 2
   ORDER BY S.dept ;
  ```
  ```

Conceptual SQL Evaluation


```
SELECT S.dept, AVG(S.gpa),
       COUNT(*)
  FROM students S
 WHERE S.gender = 'F'
 GROUP BY S.dept
 HAVING COUNT(*) > 2
 ORDER BY S.dept ;
```