COS 597D:
Principles of
Database and Information Systems

SQL:
Overview and highlights

Based on slides for Database Management Systems by R. Ramakrishnan and J. Gehrke

The SQL Query Language

- **S**tructured **Q**uery **L**anguage
- Developed by IBM (system R) in the 1970s
- Need for a standard since it is used by many vendors
  - ANSI (American National Standards Institute)
  - ISO (International Organization for Standardization)
- Standards:
  - SQL-86
  - SQL-92 (major revision)
  - SQL-99 (major extensions)
  - SQL 2003 (XML ↔ SQL)
  - SQL 2008
  - SQL 2011
  - continue enhancements

Creating Relations in SQL

Observe:
•type (domain) of each attribute specified

•type enforced by DBMS whenever tuples are added or modified.

- CREATE TABLE Movie (
```
 name CHAR(30),
 producer CHAR(30),
 rel_date CHAR(8),
 rating CHAR,
 PRIMARY KEY (name, producer, rel_date) )
```
- CREATE TABLE Employee (
```
SS# CHAR(9),
 name CHAR(30),
 addr CHAR(50),
 startYr INT,
 PRIMARY KEY (SS#))
```
- CREATE TABLE Assignment (
```
position CHAR(20),
 SS# CHAR(9),
 manager SS# CHAR(9),
 PRIMARY KEY (position),
 FOREIGN KEY(SS# REFERENCES Employee),
 FOREIGN KEY (managerSS# REFERENCES Employee) )
```

Referential Integrity in SQL

- SQL-92 on support all 4 options on deletes and updates.
  - Default is NO ACTION (*delete/update is rejected*)
  - CASCADE (also delete all tuples that refer to deleted tuple)
  - SET NULL / SET DEFAULT (sets foreign key value of referencing tuple)

```
CREATE TABLE Acct
        (bname CHAR(20) DEFAULT 'main',
         acctn CHAR(20),
         bal REAL,
         PRIMARY KEY ( acctn),
         FOREIGN KEY (bname) REFERENCES Branch
                    ON DELETE SET DEFAULT )
```

➢BUT individual implementations may NOT support

Primary and Candidate Keys in SQL

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

- at most one book with a given title and edition – date, publisher and isbn are determined
- Used carelessly, can prevent the storage of database instances that arise in practice! Title and ed suffice? UNIQUE (title, ed, pub)?

```
CREATE TABLE Book
  (isbn CHAR(10)
   title CHAR(100),
   ed INTEGER,
   pub CHAR(30),
   date INTEGER,
   PRIMARY KEY (isbn),
   UNIQUE (title, ed ))
```

Basic SQL Query

```
SELECT   [DISTINCT] select-list
FROM     from-list
WHERE    qualification
```

- *from-list* A list of relation names (possibly with a *range-variable* after each name).
- *select-list* A list of attributes of relations in *from-list*
- *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of <, >,=, ≤, ≥, ≠) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *from-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *select-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example Instances

- We will use these instances of the Acct and Branch relations in our examples.

instance of Branch

bname	bcity	assets
pu	Pton	10
nyu	nyc	20
time sq	nyc	30

instance of Acct

bname	acctn	bal
pu	33	356
nyu	45	500

Example of Conceptual Evaluation

```
SELECT  acctn
FROM    Branch, Acct
WHERE   Branch.bname=Acct.bname AND assets<20
```

bname	bcity	assets	bname	acctn	bal
pu	**Pton**	**10**	**pu**	**33**	**356**
pu	Pton	**10**	nyu	45	500
nyu	nyc	20	pu	33	356
nyu	nyc	20	**nyu**	45	500
time sq	nyc	30	pu	33	356
time sq	nyc	30	nyu	45	500

Expressions and Strings

```
SELECT  name, age=2011-yrofbirth
FROM    Alumni
WHERE   dept LIKE 'C%S'
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find pairs (Alumnus(a) name and age defined by year of birth) for alums whose dept. begins with "C" and ends with "S".*
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters.

Range Variables

- Refer to tuples from a relation
- Really needed only if the same relation appears twice in the FROM clause. :

```
SELECT  acctn
FROM    Branch, Acct
WHERE   Branch.bname=Acct.bname
        AND assets<20
```

OR

```
SELECT  R.acctn
FROM    Branch S, Acct R
WHERE   S.bname=R.bname
        AND assets<20
```

OR

```
SELECT  R.acctn
FROM    Branch as S, Acct as R
WHERE   S.bname=R.bname
        AND assets<20
```

```
CREATE TABLE Acct
   (bname  CHAR(20),
    acctn  CHAR(20),
    bal  REAL,
    PRIMARY KEY ( acctn),
    FOREIGN KEY (bname REFERENCES Branch )

CREATE TABLE Branch
    (bname  CHAR(20),
     bcity  CHAR(30),
     assets  REAL,
     PRIMARY KEY (bname)  )

CREATE TABLE Cust
     (name  CHAR(20),
      street  CHAR(30),
      city  CHAR(30),
      PRIMARY KEY (name)  )

CREATE TABLE Owner
  (name  CHAR(20),
   acctn  CHAR(20),
   FOREIGN KEY (name REFERENCES Cust )
   FOREIGN KEY (acctn REFERENCES Acct ) )
```

Nested Queries

Find names of all branches with accts of cust. who live in Rome

```
SELECT  A.bname
FROM  Acct A
WHERE  A.acctn IN  (SELECT  D.acctn
                    FROM  Owner D, Cust C
             WHERE  D.name = C.name AND C.city='Rome')
```

A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)

What get if use NOT IN?

To understand semantics of nested queries, think of a *nested loops* evaluation: *For each Acct tuple, check the qualification by computing the subquery.*

Nested Queries

Result = *names of all branches with accts of cust. who live in Rome:*

```
SELECT  A.bname
FROM  Acct A
WHERE  A.acctn IN  (SELECT  D.acctn
                    FROM  Owner D, Cust C
             WHERE  D.name = C.name AND C.city='Rome')
```

Princeton branch with two accts –

acct A has two owners, neither living in Rome

acct B has one owner living in Rome and one not living in Rome

Is Princeton branch in Result?

Is Princeton branch in (SELECT bname FROM Acct) - Result ?

Is Princeton branch in result of replacing IN with NOT IN above?

Nested Queries with Correlation

Find acct no.s whose owners own at least one acct with a balance over 1000

```
SELECT  D.acctn
FROM  Owner D
WHERE  EXISTS  (SELECT  *
                FROM  Owner E, Acct R
                WHERE R.bal>1000 AND R.acctn=E.acctn
                         AND E.name=D.name)
```

- EXISTS set comparison operator, like IN, tests not empty set
- UNIQUE set operator checks for duplicate tuples
  - If UNIQUE used, and * replaced by *E.name*, finds acct no.s whose owners own no more than one acct with a balance over 1000.
- Why, in general, subquery must be re-computed for each Branch tuple.

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL, *op* from >,<,=,≥,≤,≠
- *Find names of branches with assets at least as large as the assets of some NYC branch:*

```
SELECT  B.bname
FROM  Branch B
WHERE  B.assets ≥ ANY  (SELECT  Q.assets
                        FROM  Branch Q
                        WHERE Q.bcity='NYC')
```

Includes NYC branches?

note: key word SOME is interchangable with ANY - ANY easily confused with ALL

Division in SQL

Find tournament winners who have won all tournaments.

```
SELECT  R.wname
FROM Winners R
WHERE  NOT EXISTS
         ((SELECT  S.tourn
           FROM  Winners S)
          EXCEPT
          (SELECT  T.tourn
           FROM  Winners T
           WHERE  T.wname=R.wname))
```

```
CREATE TABLE
Winners
  (wname  CHAR((30),
   tourn  CHAR(30),
   year   INTEGER)
```

Division in SQL – simple template

Schemas

- WholeRelation: (r_1, r_2, …, r_m, q_1, q_2, …q_n)
- DivisorRelation: (q_1, q_2, …q_n)
- WholeRelation ÷ DivisorRelation: (r_1, r_2, …, r_m)

```
SELECT    R.r1, R.r2, …, R.rm
FROM      WholeRelation R
WHERE  NOT EXISTS
        ( (SELECT  *
           FROM DivisorRelation Q
          )
          EXCEPT
          (SELECT T.q1, T.q2, …T.qn
           FROM  WholeRelation T
             WHERE R.r1= T.r1 ∧ R.r2 = T.r2 ∧ … ∧ R.rm = T.rm )  )
```

Division in SQL – general template

```
SELECT
FROM
WHERE  NOT EXISTS
       ((SELECT
            FROM
             WHERE       )
        EXCEPT
        (SELECT
             FROM
             WHERE       )
```

can do projections and other predicates within nested selects

Aggregate Operators

COUNT (*)
COUNT ([DISTINCT] A)
SUM ([DISTINCT] A)
AVG ([DISTINCT] A)
MAX (A)
MIN (A)

single column

Significant extension of relational algebra.

Example: *Find name and city of the poorest branch*

- The first query is illegal!

```
SELECT S.bname, MIN (S.assets)
FROM Branch S
```

- Is it poorest *branch* or poorest *branches*?

```
SELECT S.bname, S.assets
FROM Branch S
WHERE S.assets =
        (SELECT MIN (T.assets)
         FROM Branch T)
```

GROUP BY and HAVING

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

Find the maximum assets of all branches in a city for each city containing at least one branch.

```
SELECT B.bcity, MAX(B.assets)
FROM Branch B
GROUP BY B.bcity
```

- for each city - one name - aggregate assets

Queries With GROUP BY and HAVING

SELECT	[DISTINCT] *select-list*
FROM	*from-list*
WHERE	*qualification*
GROUP BY	*grouping-list*
HAVING	*group-qualification*

- The *select-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

22

Conceptual Evaluation

- Compute cross-product of *from-list*
- Discard tuples that fail *qualification* (WHERE)
- Delete `*unnecessary'* attributes
- Partition remaining tuples into groups by the value of attributes in *grouping-list*.
- Apply *group-qualification* to eliminate some groups. Expressions in *group-qualification* must have a *single value per group*! (HAVING)
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- Generate one answer tuple per qualifying group.

23

What attributes are unnecessary?

↓

What attributes are necessary:

Exactly those mentioned in

SELECT, GROUP BY or HAVING clauses

Find the maximum assets of all branches in a city for each city containing at least two branches.

```
SELECT  B.bcity,  MAX(B.assets)
FROM  Branch B
GROUP BY  B.bcity
HAVING COUNT(*) >1
```

empty WHERE

bname	bcity	assets
pu	Pton	10
pmc	Pton	8
nyu	nyc	20
time sq	nyc	30
upenn	phili	50

bcity	assets
Pton	10
Pton	8
nyc	20
nyc	30

bcity	
Pton	10
nyc	30

2nd column of result is unnamed.
(Use AS to name it.)

Joins in SQL

- SQL has both inner joins and *outer* join
- Use in "FROM … " portion of query
- Inner join variations
  - NATURAL INNER JOIN
  - Generalized versions
- Outer join includes tuples that don't match
  - fill in with nulls
  - 3 varieties: left, right, full

.

Outer Joins

- *Left outer join of S and R:*
  - take inner join of S and R (with whatever qualification)
  - add tuples of S that are not matched in inner join, filling in attributes coming from R with "null"
- *Right outer join:*
  - as for left, but fill in tuple of R
- *Full outer join:*
  - both left and right

Example

Given Tables:

sid	residence
77	GC
35	Lawrence
21	Butler

sid	dept
77	ELE
21	COS
42	MOL

NATURAL INNER JOIN:

77	GC	ELE
21	Butler	COS

NATURAL LEFT OUTER JOIN add:

35	Lawrence	null

NATURAL RIGHT OUTER JOIN add:

42	null	MOL

NATURAL FULL OUTER JOIN add ***both***

Example Query

jobs:
(<u>position</u>, division, SS#, managerSS#)

study:
(<u>SS#</u>, academic_dept., adviser)

```
SELECT DISTINCT M.academic_dept., J.division
FROM  study M NATURAL LEFT OUTER JOIN jobs J
```

What does this produce?

General form SQL Query

- Now seen all major components

Structure of Query:

SELECT *select-list*
FROM *from-list*
WHERE *qualification*
GROUP BY *grouping-list*
HAVING *group-qualification*

UNION or INTERSECT or EXCEPT

SELECT *select-list*
FROM *from-list*
WHERE *qualification*
GROUP BY *grouping-list*
HAVING *group-qualification*

… *continuing general query form*

•Three set operations
•Only these combine separate SELECT statements.
•All other SELECTs nested.

Scope of range variable within SELECT… FROM… and nested subqueries in it

30

Null Values

- represent *unknown* value or *inapplicable* attribute
- can test attribute value IS NULL or IS NOT NULL
- need a 3-valued logic (true, false and *unknown*) to deal with null values in predicates.
  - comparisons with *null* evaluate to *unknown*
  - Boolean operations on *unknown* depend on truth table
  - can test IS UNKNOWN and IS NOT UNKNOWN
- meaning of constructs must be defined carefully
  - Example: WHERE clause eliminates rows that don't evaluate to true
  - aggregations, except COUNT(*), ignore *nulls*

Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to 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, candidate key constraints, foreign key constraints, general constraints.

General Constraints

- Useful when more general ICs than keys are involved.

```
CREATE TABLE GasStation
    ( name CHAR(30),
      street CHAR(40),
      city CHAR(30),
      st CHAR(2),
      type CHAR(4),
      PRIMARY KEY (name, street, city, st),
      CHECK ( type='full' OR type='self' ),
      CHECK (st <>'nj' OR type='full') )
```

More General Constraints

- Can use queries to express constraint.
- Constraints can be named.
- Constraints can use other tables

⇒ Must check if other table modified

```
CREATE TABLE FroshSemEnroll
    ( sid CHAR(10),
      sem_title CHAR(40),
      PRIMARY KEY (sid, sem_title),
      FOREIGN KEY (sid) REFERENCES Students
      CONSTRAINT froshonly
      CHECK (2017 =
                ( SELECT S.classyear
                  FROM Students S
                  WHERE S.sid=sid) ) )
```

Constraints Over Multiple Relations

Number of bank branches in a city is less than 3 or the population of the city is greater than 100,000

- Cannot impose as CHECK on each table. If either table is empty, the CHECK is satisfied
- Is conceptually wrong to associate with individual tables
- ASSERTION is the right solution; not associated with either table.

Number of bank branches in a city is less than 3 or the population of the city is greater than 100,000

```
CREATE ASSERTION branchLimit
CHECK
( NOT EXISTS ( (SELECT C.name, C.state
                FROM Cities C
                WHERE C.pop <=100000 )
               INTERSECT
               ( SELECT D.name, D.state
                 FROM Cities D
                 WHERE 3 <=
                      (SELECT COUNT (*)
                       FROM Branches B
                       WHERE B.bcity=D.name ) ) ) )
```

Summary

- SQL an important factor in the early acceptance of the relational model
  - more natural than earlier, procedural query languages.
- Significantly more expressive power than fundamental relational model
  - Blend of relational algebra and calculus - plus extensions
- Many alternative ways to write a query
  - optimizer should look for most efficient evaluation plan
  - when efficiency counts, users need to be aware of how queries are optimized and evaluated for best results
- SQL allows specification of rich integrity constraints
  - But often DB system does not support