CompSci 516
Data Intensive Computing Systems

Lecture 2
SQL

Instructor: Sudeepa Roy

Announcement

- If you are enrolled to the class, but have not received the email from Piazza, please send me an email
- Homework 1 – Part I will be released tonight
- Homework 1 – Part II will be released on Thursday
- due on Thursday – Feb 4 – 11:59 pm

Today's topic

- SQL in a nutshell
- Reading material
  - [RG] Chapters 3 and 5
  - Additional reading for practice

    [GUW] Chapter 6

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Relational Database: Definitions

- *Relational database:* a set of relations
- *Relation:* made up of 2 parts:
  - *Schema* : specifies name of relation, plus name and type of each column.
    - E.G. Students(*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real).
  - *Instance* : a *table,* has rows and columns
    - each row/tuple follows the schema and domain constraints
    - #Rows = *cardinality*, #fields = *degree / arity.*
- Can think of a relation as a *set* of rows or *tuples,* i.e., all rows are distinct
  - however, it is true for the relational model, not for standard DBMS that allow duplicate rows. Why?

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

Cardinality = 3, degree = 5,
all rows distinct

The SQL Query Language

- Developed by IBM (system R) in the 1970s
- Need for a standard since it is used by many vendors
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision)
  - SQL-99 (major extensions, current standard)

Purposes of SQL

- Data Manipulation Language (DML)
  - Querying: SELECT-FROM-WHERE
  - Modifying: INSERT/DELETE/UPDATE
- Data Definition Language (DDL)
  - CREATE/ALTER/DROP

The SQL Query Language

- To find all 18 year old students, we can write:

all attributes

```
SELECT  *
FROM  Students S
WHERE  S.age=18
```

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

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

```
SELECT  S.name, S.login
```

Querying Multiple Relations

- What does the following query compute?

```
SELECT  S.name, E.cid
FROM  Students S, Enrolled E
WHERE  S.sid=E.sid AND E.grade="A"
```

Given the following instances of Enrolled and Students:

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

Enrolled

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

we get: ??

Querying Multiple Relations

- What does the following query compute?

```
SELECT  S.name, E.cid
FROM  Students S, Enrolled E
WHERE  S.sid=E.sid AND E.grade="A"
```

Given the following instances of Enrolled and Students:

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

Enrolled

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

we get:

S.name	E.cid
Smith	Topology112

Creating Relations in SQL

- Creates the Students relation
  - the type (domain) of each field is specified
  - enforced by the DBMS whenever tuples are added or modified

```
CREATE TABLE Students
    (sid CHAR(20),
     name CHAR(20),
     login CHAR(10),
     age INTEGER,
     gpa REAL)
```

- As another example, the Enrolled table holds information about courses that students take

```
CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(20),
     grade CHAR(2))
```

Destroying and Altering Relations

```
DROP TABLE Students
```

- Destroys the relation Students
  - The schema information *and* the tuples are deleted.

```
ALTER TABLE Students
    ADD COLUMN firstYear: integer
```

- The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a *null* value in the new field.

Adding and Deleting Tuples

- Can insert a single tuple using:

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

- Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

Integrity Constraints (ICs)

- IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

Primary Key Constraints

- A set of fields is a *key* for a relation if :
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key.
  - Part 2 false? A *superkey*
  - If there's >1 key for a relation, one of the keys is chosen (by DBA) to be the *primary key*
  - E.g., *sid* is a key for Students
  - The set {*sid, gpa*} is a superkey.
  - *Is there any possible benefit to refer to a tuple using primary key (than any key)?*

Primary and Candidate Keys in SQL

- Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.
  - "For a given student and course, there is a single grade."

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

Primary and Candidate Keys in SQL

- Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.
  - "For a given student and course, there is a single grade."

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

Primary and Candidate Keys in SQL

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

- "For a given student and course, there is a single grade."
- vs.
- "Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade."

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

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

Primary and Candidate Keys in SQL

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

- "For a given student and course, there is a single grade."
- vs.
- "Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade."

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

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

Primary and Candidate Keys in SQL

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

- "For a given student and course, there is a single grade."
- vs.
- "Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade."

- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

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

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

Foreign Keys, Referential Integrity

- *Foreign key* : Set of fields in one relation that is used to `refer' to a tuple in another relation
  - Must correspond to primary key of the second relation
  - Like a `logical pointer'
- E.g. *sid* is a foreign key referring to Students:
  - Enrolled(*sid*: string, *cid*: string, *grade*: string)
  - If all foreign key constraints are enforced, *referential integrity* is achieved, i.e., no dangling references.

Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to enroll for courses.

```
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

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

- Consider Students and Enrolled
  - *sid* in Enrolled is a foreign key that references Students.
- What should be done if an Enrolled tuple with a non-existent student id is inserted?
  - *Reject it!*
- What should be done if a Students tuple is deleted?
  - Three semantics allowed by SQL
  - Also delete all Enrolled tuples that refer to it (cascade delete)
  - Disallow deletion of a Students tuple that is referred to.
  - Set sid in Enrolled tuples that refer to it to a *default sid*.
    - In SQL, also: Set sid in Enrolled tuples that refer to it to a special value *null*, denoting *`unknown'* or *`inapplicable'*
- Similar if primary key of Students tuple is updated.

Referential Integrity in SQL

- SQL/92 and SQL:1999 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 Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid,cid),
   FOREIGN KEY (sid)
    REFERENCES Students
     ON DELETE CASCADE
     ON UPDATE SET DEFAULT )
```

Where do ICs Come From?

- ICs are based upon the semantics of the real-world enterprise that is being described in the database relations
- Can we infer ICs from an instance?
  - We can check a database instance to see if an IC is violated, but 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.

Example Instances

- We will use these instances of the Sailors and Reserves relations in our examples
- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

Sailor

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Reserves

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

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 relations in *relation-list*
- *qualification* Comparisons
  - (Attr *op* const) or (Attr1 *op* Attr2)
  - where *op* is one of $<, >, =, \leq, \geq, \neq$ 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

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

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of <*relation-list*>
  - Discard resulting tuples if they fail <*qualifications*>.
  - Delete attributes that are not in <*target-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 of Conceptual Evaluation

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND R.bid=103
```

Sailor

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Step 1: Form **cross product** of Sailor and Reserves

sid	sname	rating	age	sid	bid	day
22	dustin	7	45	22	101	10/10/96
22	dustin	7	45	58	103	11/12/96
31	lubber	8	55	22	101	10/10/96
31	lubber	8	55	58	103	11/12/96
58	rusty	10	35	22	101	10/10/96
58	rusty	10	35	58	103	11/12/96

Reserves

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

Example of Conceptual Evaluation

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND R.bid=103
```

Sailor

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Step 2: Discard tuples that do not satisfy <qualification>

sid	sname	rating	age	sid	bid	day
~~22~~	~~dustin~~	~~7~~	~~45~~	~~22~~	~~101~~	~~10/10/96~~
~~22~~	~~dustin~~	~~7~~	~~45~~	~~58~~	~~103~~	~~11/12/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~22~~	~~101~~	~~10/10/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~58~~	~~103~~	~~11/12/96~~
~~58~~	~~rusty~~	~~10~~	~~35~~	~~22~~	~~101~~	~~10/10/96~~
58	rusty	10	35	58	103	11/12/96

Reserves

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

Example of Conceptual Evaluation

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND R.bid=103
```

Sailor

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Reserves

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

Step 3: Select the specified attribute(s)

sid	sname	rating	age	sid	bid	day
~~22~~	~~dustin~~	~~7~~	~~45~~	~~22~~	~~101~~	~~10/10/96~~
~~22~~	~~dustin~~	~~7~~	~~45~~	~~58~~	~~103~~	~~11/12/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~22~~	~~101~~	~~10/10/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~58~~	~~103~~	~~11/12/96~~
~~58~~	~~rusty~~	~~10~~	~~35~~	~~22~~	~~101~~	~~10/10/96~~
58	rusty	10	35	58	103	11/12/96

A Note on "Range Variables"

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103
```

OR

```
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103
```

It is good style, however, to use range variables always!

Find sailors who've reserved at least one boat

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

Sailor

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Reserves

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

Find sailors who've reserved at least one boat

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

Sailor

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Reserves

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

- Would adding DISTINCT to this query make a difference?

Find sailors who've reserved at least one boat

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

Sailor

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Reserves

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

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Joins

- Condition/Theta-Join
- Equi-Join
- Natural-Join
- (Left/Right/Full) Outer-Join

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

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

Condition/Theta Join

```
SELECT *
FROM Sailors S, Reserves R
WHERE S.sid=R.sid and age >= 40
```

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Form cross product, discard rows that do not satisfy the condition

sid	sname	rating	age	sid	bid	day
22	dustin	7	45	22	101	10/10/96
~~22~~	~~dustin~~	~~7~~	~~45~~	~~58~~	~~103~~	~~11/12/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~22~~	~~101~~	~~10/10/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~58~~	~~103~~	~~11/12/96~~
~~58~~	~~rusty~~	~~10~~	~~35~~	~~22~~	~~101~~	~~10/10/96~~
~~58~~	~~rusty~~	~~10~~	~~35~~	~~58~~	~~103~~	~~11/12/96~~

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

Equi Join

```
SELECT *
FROM Sailors S, Reserves R
WHERE S.sid=R.sid and age = 45
```

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

A special case of theta join
Join condition only has equality predicate =

sid	sname	rating	age	sid	bid	day
22	dustin	7	45	22	101	10/10/96
~~22~~	~~dustin~~	~~7~~	~~45~~	~~58~~	~~103~~	~~11/12/96~~
31	lubber	8	55	22	101	10/10/96
~~31~~	~~lubber~~	~~8~~	~~55~~	~~58~~	~~103~~	~~11/12/96~~
58	rusty	10	35	22	101	10/10/96
~~58~~	~~rusty~~	~~10~~	~~35~~	~~58~~	~~103~~	~~11/12/96~~

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

Natural Join

```
SELECT *
FROM Sailors S NATURAL JOIN Reserves R
```

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

A special case of equi join
Equality condition on ALL common predicates
Duplicate columns are eliminated

sid	sname	rating	age	bid	day
22	dustin	7	45	101	10/10/96
~~22~~	~~dustin~~	~~7~~	~~45~~	~~103~~	~~11/12/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~101~~	~~10/10/96~~
~~31~~	~~lubber~~	~~8~~	~~55~~	~~103~~	~~11/12/96~~
~~58~~	~~rusty~~	~~10~~	~~35~~	~~101~~	~~10/10/96~~
58	rusty	10	35	103	11/12/96

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

Outer Join

```
SELECT  S.sid, R. bid
FROM  Sailors S LEFT OUTER JOIN Reserves R
ON  S.sid=R.sid
```

sid	sname	rating	age
22	dustin	7	45
31	lubber	8	55
58	rusty	10	35

Preserves all tuples from the left table whether or not there is a match
Similarly RIGHT/FULL outer join

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

sid	bid
22	101
31	null
58	103

Expressions and Strings

```
SELECT  S.age,    age1=S.age-5,    2*S.age AS age2
FROM  Sailors S
WHERE  S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching
- *Find triples (of ages of sailors and two fields defined by expressions) for sailors*
  - *whose names begin and end with B and contain at least three characters*
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters
  - You will need these often

Find sid's of sailors who've reserved a red or a green boat

Sailors (sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)

- Assume a Boats relation (see book)

Find sid's of sailors who've reserved a red or a green boat

Sailors (sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)

- Assume a Boats relation (see book)
- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples
  - can themselves be the result of SQL queries
- If we replace OR by AND in the first version, what do we get?
- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid  AND R.bid=B.bid
  AND (B.color='red' OR B.color='green')
```

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid  AND R.bid=B.bid
            AND B.color='red'
UNION
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid  AND R.bid=B.bid
            AND B.color='green'
```

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

Sailors (sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)

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

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don't support it

```
SELECT  S.sid
FROM  Sailors S, Boats B1, Reserves R1,
          Boats B2, Reserves R2
WHERE  S.sid=R1.sid  AND R1.bid=B1.bid
  AND  S.sid=R2.sid  AND R2.bid=B2.bid
  AND (B1.color='red' AND B2.color='green')
```

Key field!

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid  AND R.bid=B.bid
            AND B.color='red'
INTERSECT
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid  AND R.bid=B.bid
            AND B.color='green'
```

Nested Queries

Find 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)
```

Sailors (sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)

- A very powerful feature of SQL:
  - a WHERE/FROM/HAVING clause can itself contain an SQL query
- 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*
- *Learn about the "WITH" clause yourself! A very useful clause to define subqueries (like views) before you use them!*

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
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple

Nested Queries with Correlation

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

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

- 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

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* IN >,<,=,≥,≤,≠
- Find sailors whose rating is greater than that of some sailor called Horatio
  - similarly ALL

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

"Division" in SQL

Division corresponds to "ALL", we will learn about it when we do Relational Algebra and Calculus

Find sailors who've reserved all boats.

```
SELECT  S.sname                                    (1)
FROM  Sailors S
WHERE  NOT EXISTS
        ((SELECT  B.bid
          FROM  Boats B)
         EXCEPT
          (SELECT  R.bid
           FROM  Reserves R
           WHERE  R.sid=S.sid))
```

- Option 1:
- Let's do it the hard way, without EXCEPT:

```
SELECT  S.sname                    Sailors S such that ...
FROM  Sailors S
WHERE  NOT EXISTS  (SELECT  B.bid      there is no boat B without ...
                    FROM  Boats B
                    WHERE  NOT EXISTS  (SELECT  R.bid
   a Reserves tuple showing S reserved B  FROM  Reserves R
                                          WHERE  R.bid=B.bid
                                                 AND R.sid=S.sid))
```

(2)

Aggregate Operators

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

single column

Check yourself:
What do these queries compute?

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

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

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

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

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

Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples
  - Sometimes, we want to apply them to each of several *groups* of tuples
- Consider: *Find the age of the youngest sailor for each rating level*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

For $i = 1, 2, \ldots, 10$:

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

Conceptual Evaluation

An example in the following slides

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. Expressions in *group-qualification* must have a *single value per group*!
   - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*
6. One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

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

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

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

Step 1: Form the cross product: FROM clause
(some attributes are omitted for simplicity)

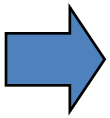
rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

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

Step 2: Apply WHERE clause

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



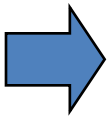
rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
~~10~~	~~16.0~~
9	35.0
3	25.5
3	63.5
3	25.5

Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

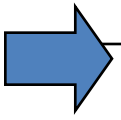
```
SELECT  S.rating,  MIN (S.age) AS minage
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

Step 3: Apply GROUP BY according to the listed attributes

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

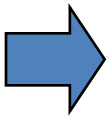
Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

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

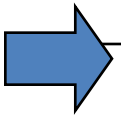
Step 4: Apply HAVING clause

The *group-qualification* is applied to eliminate some groups

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

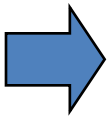
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1

Step 5: Apply SELECT clause

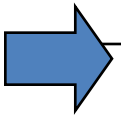
Apply the aggregate operator

At the end, one tuple per group

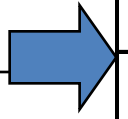
rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

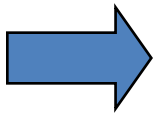


rating	minage
3	25.5
7	35.0
8	25.5

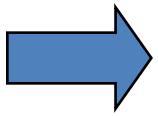
Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors and with every sailor under 60.

HAVING COUNT (*) > 1 AND EVERY (S.age <=60)

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0



rating	minage
7	35.0
8	25.5

What is the result of changing EVERY to ANY?

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
- The presence of *null* complicates many issues. E.g.:
  1. Special operators needed to check if value IS/IS NOT NULL
     - **DO NOT USE '=' WITH NULL!**
  2. Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?
     - We need a 3-valued logic: true, false and *unknown*
     - Consider them as 1, 0, 0.5 and treat (OR as MAX) and (AND as MIN) – similar to Boolean true (1) and false (0) 2-valued logic
  3. Meaning of constructs must be defined carefully
     - e.g., WHERE clause eliminates rows that don't evaluate to true

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, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.

General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- 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)))
```

Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)

Summary

- Today: SQL in a nutshell
  - A huge number of constructs and possibilities
  - You need to learn and practice it on your own