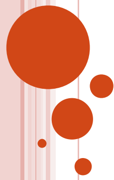
Administrative

- Exam
  - Oct 19, 2010
  - 2:30pm in 3005 (usual room)
- No class on Oct 21
- What topic to do for review?
- November 18th away for conference



Database Systems I

Week 5
SQL Queries

Introduction

- We now introduce SQL, the standard query language for relational DBS.
- As RA, an SQL query takes one or two input tables and returns one output table.
- Any RA query can also be formulated in SQL, but in a more user-friendly manner.
- In addition, SQL contains certain features of great practical importance that go beyond the expressiveness of RA, e.g. sorting and aggregation functions.

Example Instances

R1

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

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

- We will use these instances of the Sailors and Reserves tables in our examples.

Basic SQL Query

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

- *relation-list*: list of relation names (possibly with a *tuple-variable* after each name).
- *target-list*: 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

- *Semantics* of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the *Cartesian product* of *relation-list*.
  - *Selection* of the tuples satisfying *qualifications*.
  - *Projection* onto the attributes that are in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is not an efficient way to process a query!
  - An optimizer will find more efficient strategies to compute *the same answers*.
- It is often helpful to write an SQL query in the same order (FROM, WHERE, SELECT).

Example Conceptual Evaluation

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

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Projection

- Expressed through the SELECT clause.
- Can specify any subset of the set of all attributes.

```
SELECT sname, age
FROM Sailors;
```

- "*" selects all attributes.

```
SELECT *
FROM Sailors;
```

Can rename attributes.

```
SELECT sname as Name, age as Age
FROM Sailors;
```


Projection

- For numeric attribute values, can also apply arithmetic operators +, * etc.
- Can create derived attributes and name them, using AS or "=":

```
SELECT age AS age0, age1=age-5, 2*S.age AS age2
FROM Sailors;
```

PROJECTION

- The result of a projection can contain duplicates (why?).
- To eliminate duplicates from the output, specify DISTINCT in the SELECT clause.

```
SELECT  DISTINCT age
FROM  Sailors;
```


SELECTION

- Expressed through the WHERE clause.
- Selection conditions can compare constants and attributes of relations mentioned in the FROM clause.
- Comparison operators: =, <>, <, >, <=, >=
- For numeric attribute values, can also apply arithmetic operators +, * etc.
- Simple conditions can be combined using the logical operators AND, OR and NOT.
- Default precedence: NOT, AND, OR.
- Use parentheses to change precedence.

SELECTION

```
SELECT  *
FROM    Sailors
WHERE  sname = 'Watson';

SELECT  *
FROM    Sailors
WHERE  rating >= age;

SELECT  *
FROM    Sailors
WHERE  (rating = 5 OR rating = 6) AND age <= 20;
```

SFU SIMON FRASER UNIVERSITY

String Comparisons

- LIKE is used for approximate conditions (pattern matching) on string-valued attributes:
  string LIKE pattern
  Satisfied if pattern contained in string attribute.
- NOT LIKE satisfied if pattern not contained in string attribute.
- '_' in pattern stands for any one character
- '%' stands for 0 or more arbitrary characters.

```
SELECT *
FROM Sailors S
WHERE S.sname LIKE 'B_%B';
```


SFU SIMON FRASER UNIVERSITY

Null Values

- Special attribute value *NULL* can be interpreted as
  - Value unknown (e.g., a rating has not yet been assigned),
  - Value inapplicable (e.g., no spouse's name),
  - Value withheld (e.g., the phone number).
- The presence of *NULL* complicates many issues:
  - Special operators needed to check if value is null.
  - Is rating>8 true or false when rating is equal to null?
  - What about AND, OR and NOT connectives?
  - Meaning of constructs must be defined carefully.
    - E.g., how to deal with tuples that evaluate neither to TRUE nor to FALSE in a selection?

SFU SIMON FRASER UNIVERSITY

Null Values

- *NULL* is not a constant that can be explicitly used as an argument of some expression.
- *NULL* values need to be taken into account when evaluating conditions in the *WHERE* clause.
- Rules for *NULL* values:
  - An arithmetic operator with (at least) one *NULL* argument always returns *NULL* .
  - The comparison of a *NULL* value to any second value returns a result of UNKNOWN.
- A selection returns only those tuples that make the condition in the *WHERE* clause TRUE, those with UNKNOWN or FALSE result do not qualify.

Truth Value Unknown

- Three-valued logic: TRUE, UNKNOWN, FALSE.
- Can think of TRUE = 1, UNKNOWN = ½, FALSE = 0
  - AND of two truth values: their minimum.
  - OR of two truth values: their maximum.
  - NOT of a truth value: 1 – the truth value.
- Examples:

  TRUE AND UNKNOWN = UNKNOWN
  FALSE AND UNKNOWN = FALSE
  FALSE OR UNKNOWN = UNKNOWN
  NOT UNKNOWN = UNKNOWN

Truth Value Unknown

```
SELECT *
FROM   Sailors
WHERE  rating < 5  OR  rating >= 5;
```

Does not return all sailors, but only those with non-NULL rating.

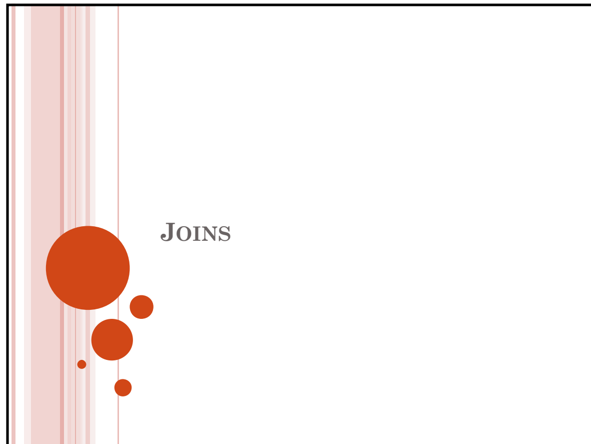
Ordering the Output

- Can order the output of a query with respect to any attribute or list of attributes.
- Add *ORDER BY* clause to the query:

```
SELECT *
FROM   Sailors S
WHERE age < 20
ORDER BY rating;

SELECT *
FROM   Sailors S
WHERE age < 20
ORDER BY rating, age;
```

- By default, ascending order. Use DESC to specify descending order.



Cartesian Product

- Expressed in FROM clause.
- Forms the *Cartesian product* of all relations listed in the FROM clause, in the given order.

```
SELECT *
FROM    Sailors, Reserves;
```

- So far, not very meaningful.

Join

- Expressed in FROM clause and WHERE clause.
- Forms the subset of the *Cartesian product* of all relations listed in the FROM clause that satisfies the WHERE condition:

```
SELECT *
FROM    Sailors, Reserves
WHERE Sailors.sid = Reserves.sid;
```

- In case of ambiguity, prefix attribute names with relation name, using the dot-notation.

JOIN

- Since joins are so common operations, SQL provides JOIN as a shorthand.

```
SELECT *
FROM Sailors JOIN Reserves ON
        Sailors.sid = Reserves.sid;
```

- NATURAL JOIN produces the natural join of the two input tables, i.e. an equi-join on all attributes common to the input tables.

```
SELECT *
FROM Sailors NATURAL  JOIN Reserves;
```


JOIN

- Typically, there are some *dangling* tuples in one of the input tables that have no matching tuple in the other table.
  - Dangling tuples are not contained in the output.
- Outer joins are join variants that do not loose any information from the input tables.

LEFT OUTER JOIN

- includes all dangling tuples from the **left** input table
- *NULL* values filled in for all attributes of the right input table

```
SELECT *
FROM   employee  LEFT OUTER JOIN department
          ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
John	NULL	NULL	NULL
Steinberg	33	Engineering	33

Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
John	NULL

Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

SFU SIMON FRASER UNIVERSITY

Right Outer Join

- includes all dangling tuples from the **right** input table
- *NULL* values filled in for all attributes of the right input table

```
SELECT *
FROM   employee RIGHT OUTER JOIN department
       ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Steinberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	35

Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
John	NULL

Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

- What's the difference between LEFT and RIGHT joins?
- Can one replace the other?

SFU SIMON FRASER UNIVERSITY

Full Outer Join

- includes all dangling tuples from **both** input tables
- *NULL* values filled in for all attributes of any dangling tuples

```
SELECT *
FROM   employee
       FULL OUTER JOIN department
         ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
John	NULL	NULL	NULL
Steinberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	35

Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
John	NULL

Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

SFU SIMON FRASER UNIVERSITY

SQL JOINS

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffatt, 2008

Tuple Variables

- Tuple variable is an alias referencing a tuple from the relation over which it has been defined.
- Again, use dot-notation.
- Needed only if the same relation name appears twice in the query.

```
SELECT  S.sname
FROM    Sailors S, Reserves R1, Reserves R2
WHERE  S.sid=R1.sid AND S.sid=R2.sid
           AND R1.bid <> R2.bid
```

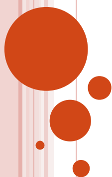
- It is good style, however, to use tuple variables always.
  - Benefits?

A Further Example

- Find sailors who've reserved at least one boat:

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

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



Set operators

Set Operations

- SQL supports the three basic set operations.
- UNION: union of two relations
- INTERSECT: intersection of two relations
- EXCEPT: set-difference of two relations
- Two input relations must have same schemas. Can use AS to make input relations compatible.

Set Operations

- Find sid's of sailors who've reserved a red or a green boat.
- If we replace OR by AND in the first version, what do we get?
  - Sailors who reserved both red and green boats?
- 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');
```


Set Operations

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

```
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');
```

- Find sid's of sailors who've reserved a red and a green boat.
  - Complex query
  - Difficult to understand

```
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');
```

Set Operations

- Find sid's of sailors who've reserved a red and a green boat.
  - Complex query
  - Difficult to understand
- Use INTERSECT

```
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 attribute!

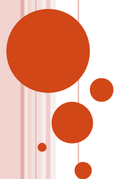
```
(SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND
R.bid=B.bid AND B.color='red')
INTERSECT
(SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND
R.bid=B.bid AND B.color='green');
```


Set Operations

Query(… OR …) → Query(…) UNION Query(…)

Query(… AND …) → Query(…) INTERSECT Query(…)

Subqueries
Breaking complex queries down



Subqueries

- A *subquery* is a query nested within another SQL query.
- Subqueries can
  - return a single constant that can be used in the WHERE clause,
  - return a relation that can be used in the WHERE clause,
  - appear in the FROM clause, followed by a tuple variable through which results can be referenced in the query.
- Subqueries can contain further subqueries etc., i.e. there is no restriction on the level of nesting.

Subqueries

- The output of a subquery returning a single constant can be compared using the normal operators =, <>, >, etc.

```
SELECT  S.age
FROM  Sailors S
WHERE  S.age >  (SELECT  S.age
                 FROM  Sailors S
                 WHERE  S.sid=22);
```

- How can we be sure that the subquery returns only one constant?
- To understand semantics of nested queries, think of a *nested loops* evaluation
  - For each Sailors tuple, check the qualification by computing the subquery.

Subqueries

- The output of a subquery R returning an entire relation can be compared using the special operators
  - EXISTS R is true if and only if R is non-empty.
  - s IN R is true if and only if tuple (constant) s is contained in R.
  - s NOT IN R is true if and only if tuple (constant) s is not contained in R.
  - s op ALL R is true if and only if constant s fulfills op with respect to every value in (unary) R.
  - s op ANY R is true if and only if constant s fulfills op with respect to at least one value in (unary) R.
- Op can be one of $>, <, =, \geq, \leq, \neq$

Subqueries

- EXISTS, ALL and ANY can be negated by putting NOT in front of the entire expression.

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

```
SELECT  R.bid
FROM  Reserves R
WHERE  R.sid  IN   (SELECT  S.sid
                    FROM  Sailors S
                    WHERE  S.name='Rusty');
```

What is the result?

```
SELECT  *
FROM  Sailors S1
WHERE  S1.age > ALL   (SELECT  S2.age
                       FROM  Sailors S2
                       WHERE  S2.name='rusty');
```


Subqueries

- In a FROM clause, we can use a parenthesized subquery instead of a table.

```
SELECT  *
FROM  Reserves R, (SELECT  S.sid
                   FROM  Sailors S
                   WHERE  S.age>60) OldSailors
WHERE R.sid = OldSailors.sid;
```

- Need to define a corresponding tuple variable to reference tuples from the subquery output.

Correlated Subqueries

```
SELECT  B.bid
FROM  Boats B
WHERE  NOT EXISTS  (SELECT  *
                    FROM  Reserves R
                    WHERE  R.bid=B.bid);
```

- The second subquery is *correlated* to the outer query, i.e. the execution of the subquery may return different results for every tuple of the outer query.
- Illustrates why, in general, subquery must be re-computed for each tuple of the outer query / tuple.
- Notice the lack of variable before NOT EXISTS.
  - What does NOT EXIST?

A Further Example

- *Find sid's of sailors who've reserved both a red and a green boat:*

Red:
```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
```
Green:
```
        AND S.sid IN  (SELECT  S2.sid
                       FROM  Sailors S2, Boats B2, Reserves R2
                       WHERE  S2.sid=R2.sid AND
                           R2.bid=B2.bid AND  B2.color='green');
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause.
  - What about INTERSECT query?

Division in SQL

- *Find sailors who've reserved all boats.*
- With EXCEPT:

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

For each sailor, check to see if there are any boats that have *not been* reserved.

Any boats not reserved by the sailor.

All reservations by all sailors.

Division in SQL

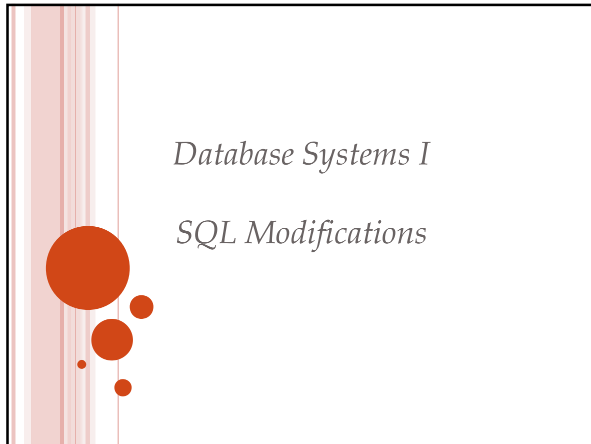
- *Find sailors who've reserved all boats.*
- Without EXCEPT:

```
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 without ...

a Reserves tuple showing S reserved B



Introduction

- SQL provides three operations that modify the instance (state) of a DB:
  - INSERT: inserts new tuple(s),
  - DELETE: deletes existing tuples(s), and
  - UPDATE: updates attribute value(s)s of existing tuple(s).
- Individual modifications may yield an inconsistent DB state, and only a sequence of modifications (*transaction*) may lead again to a consistent state.
- The DBS ensures certain properties of a transaction that guarantee the consistency of the resulting DB state.

Insertion

```
INSERT INTO  R (A1, . . ., An)
VALUES  (v1, . . ., vn);
```

- Inserts a single tuple with values vi for attributes Ai into table R.

```
INSERT INTO  Sailors (sid, sname,rating, age)
VALUES  (69,'mike', 2, 20);
```

- If values are not provided for all attributes, NULL values will be inserted.
- Short hand if all attribute values are given:

```
INSERT INTO  Sailors
VALUES  (69,'mike', 2, 20);
```

- Values need to be provided in the order of the corresponding attributes.

INSERTION

```
INSERT INTO  R (A1, . . ., An)
<subquery> ;
```

- Inserts a set of tuples (relation) with values for attributes Ai into table R, as specified by a subquery.

```
INSERT INTO  Sailors (sid)
      SELECT DISTINCT R.sid
      FROM Reserves R
      WHERE  R.sid NOT IN
            (SELECT sid
            FROM Sailors);
```

- The subquery is completely evaluated before the first tuple is inserted.

DELETION

```
DELETE FROM  R
WHERE <condition> ;
```

- Deletes the set (!) of all tuples from R which satisfy the condition of the WHERE clause.

```
DELETE FROM  Sailors       // one tuple
WHERE  sid = 69;

DELETE FROM  Sailors       // multiple tuples
WHERE  rating < 3;
```

- Cannot directly specify a tuple to be deleted as is possible for insertions.

UPDATE

```
UPDATE R
SET <new value assignments>
WHERE <condition>;
```

- Updates attributes in the specified manner for all R tuples satisfying the given condition.

```
UPDATE Sailors
SET age = age + 1;

UPDATE Sailors
SET rating = rating * 1.1, age = age + 1
WHERE  age < 30 and sid IN
      (SELECT R.sid
      FROM Reserves R);
```