SQL

UVic C SC 370

Dr. Daniel M. German
Department of Computer Science
University of Victoria

June 3, 2004 Version: 1.1.2

Overview

- A review of SQL
  - Basic Select statements
  - UNION, INTERSECT, EXCEPT
  - Nested queries
  - Aggregate operations
  - GROUP BY and HAVING
  - NULL
  - Constraints

Basic form of a SQL Query

- SQL query:

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

- Every query must have a `SELECT` clause
- The `FROM` specifies a cross product of tables
- The optional `WHERE` clause specifies selection conditions on the tables mentioned in the `FROM`
- This query corresponds to a relational algebra expression involving selection, projections and cross-products.

Example

-
  ```
  SELECT S.sname, S.age
  FROM Sailors S
  ```

-

sname	*age*
Dustin	45
Brutus	33
Lubber	55.5
Andy	25.5
Rusty	35
Horatio	35
Zorba	16
Horatio	35
Art	25.5
Bob	63.5

Example, without DISTINCT

- This could include several copies of the same row

```
SELECT S.sname, S.age
FROM Sailors S
```

- This result is known as a **multiset**

sname	*age*
Dustin	45
Brutus	33
Lubber	55.5
Andy	25.5
Rusty	35
Horatio	35
Zorba	16
Horatio	35
Art	25.5
Bob	63.5

Multiset

- A **Multiset** is similar to a set but there could be multiple copies of each element
- Two multisets could have the same elements and still be different because the number of copies of each element, e.g. $\{a, b, b\}$ and $\{b, a, b\}$ are the same, but $\{a, a, b\}$ is not.

Another Example

- (Q11) Find all sailors with a rating above 7

```
SELECT S.sid, S.sname, S.rating, S.age
FROM Sailors AS S
WHERE S.rating > 7
```

- Notice the use of AS to as an alternative for an alias

sid	*sname*	*rating*	*age*
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
71	Zorba	10	16
74	Horatio	9	35

Another Example... using *

- * shorthand for “all columns” in the order in which they are defined in the table schema
- Poor programming style. Query changes if the schema changes

```
SELECT *
FROM Sailors AS S
WHERE S.rating > 7
```

sid	*sname*	*rating*	*age*
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
71	Zorba	10	16
74	Horatio	9	35

SELECT in detail

- `SELECT` does projection
- `WHERE` does selection
- The **from-list** in the `FROM` clause is list of tables
- The **select-list** is a list of expressions involving columns of those tables (from-list)
- The **qualification** in the `WHERE` is a boolean combination of conditions of the form **expression `op` expression** where `op` is one of: $<, <=, =, <>, >=, >$
- The `DISTINCT` is optional

But, what is the meaning of a query?

- A query does not tells us how to compute it
- The result of a query is a **relation**, which is a **multiset** of rows
- A conceptual evaluation strategy (easy to understand, but not necessarily what the database uses–in fact, it is quite inefficient)

  1. Compute the cross product of the tables in the **from-list**
  2. Delete the rows in the cross-product that fail the **qualification** conditions
  3. Delete all columns that do not appear in the **select-list**
  4. If **DISTINCT** is specified, eliminate duplicate rows

Example of Query Evaluation

- *Q1 Find the names of sailors who have reserved boat number 103*

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

- Assume these instances:

sid	*bid*	*day*
22	101	1998-10-10
58	103	1998-11-12

sid	*sname*	*rating*	*age*
22	Dustin	7	45
31	Lubber	8	55.5
58	Rusty	10	35

Query Evaluation...

- The first step is to compute the cross product:

sid	*sname*	*rating*	*age*	*sid*	*bid*	*day*
22	Dustin	7	45	22	101	1998-10-10
22	Dustin	7	45	58	103	1998-11-12
31	Lubber	8	55.5	22	101	1998-10-10
31	Lubber	8	55.5	58	103	1998-11-12
58	Rusty	10	35	22	101	1998-10-10
58	Rusty	10	35	58	103	1998-11-12

- Now we apply qualification:

  `S.sid = R.sid AND R.bid = 103`

sid	*sname*	*rating*	*age*	*sid*	*bid*	*day*
58	Rusty	10	35	58	103	1998-11-12

Query Evaluation...

- Finally, we do projection:

sname
Rusty

Expressions and Strings in the SELECT

- Each item in the **select-list** can be an expression of the form **expression AS column_name** where **expression** is any arithmetic or string expression over columns and constants
- **column_name** becomes the name of the result column
- It can also contain aggregates (discussed later)
- Some DBMS allow the use of UD (user defined) and library functions
- Example:

```
SELECT S.sname, S.rating+1 AS rating
FROM Sailors S, Reserves R1, Reserves R2
WHERE S.sid = R1.sid AND S.sid = R2.sid AND
      R1.day = R2.day AND R1.bid <> R2.bid
```

Collating Sequences

- Character and string operations are done by using an ordering called **collating sequence**
- This allows for multi-byte and foreign languages support
- Also, some DBMS use a case-sensitive default collating sequence (mysql, MS SQL server, **the textbook** for instance)

Pattern Matching

- SQL provides very rudimentary pattern matching:
- `LIKE` operator
  - %: Wild card, match zero or more arbitrary characters
  - _: Match exactly one arbitrary character
- `'_AB%'` matches any string that has at least 3 chars, A as second char, and B as third one.
- Example: Q18: Find the ages of sailors whose name begins and ends with B and has at least 3 characters

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

- Notice the use of the % at the end of the string. It matches the trailing spaces

UNION

- Computes the **union** between two `SELECT` statements
- *Q5: Find the names of sailors who have reserved a red or a green boat (or both)*

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

- Using `UNION`:

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

INTERSECT

- Computes the **intersection** between two `SELECT` statements
- *Q6: Find the names of sailors who have reserved both a red and a green boat*

```
SELECT DISTINCT s.sname
FROM Sailors S, Reserves R1, Reserves R2, Boats B1, Boats B2
WHERE S.sid = R1.sid  AND S.sid = R2.sid
    AND R1.bid = B1.bid AND R2.bid = B2.bid
    AND B1.color = ’red’ AND B2.color = ’green’
```

- Using `INTERSECT`:

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

- This has a bug, can you spot it? See textbook (page 143 for discussion of the error)

EXCEPT

- Computes the **set difference** between two `SELECT` statements
- *Q19: Find the sids of all sailors who have reserved red boats but not green boats.*

```
SELECT s.sid
FROM Sailors S, Reserves R,  Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ’red’
EXCEPT
SELECT s.sid
FROM Sailors S, Reserves R,  Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ’green’
```

- Or the simpler query:

```
SELECT r.sid
FROM Reserves R,  Boats B
WHERE  R.bid = B.bid AND B.color = ’red’
EXCEPT
SELECT R.sid
FROM Reserves R,  Boats B
WHERE R.bid = B.bid AND B.color = ’green’
```

Nested Queries

- In SQL you can embed queries (**subqueries**) inside queries
- Subqueries can include conditions that refer to a relation that needs to be computed
- Subqueries usually appear in the `WHERE` clause, but can also appear in the `FROM` (or `HAVING`)
- *Q1: Find the names of the sailors who have reserved boat 103*

```
SELECT s.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid = 103)
```

- *Q1: Find the names of the sailors who have* **NOT** *reserved boat 103*
- Replace `IN` with `NOT IN`

Conceptual Evaluation Strategy

- Extend step 2 by recomputing the subquery before testing the **qualification** condition
- If the subquery has another subquery, we apply the same idea recursively

Multiple Nested Queries

- *Q2. Find the names of sailors who have reserved a red boat*

```
SELECT s.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid IN (SELECT B.Bid
                                FROM Boats B
                                WHERE B.color = ’red’))
```

- `IN` tests if the first operand (a row) is in its second operand (a relation)

Correlated Nested Queries

- The inner query can depend on a value of the current row being examined
- *Q1. Find the names of sailors who have reserved boat number 103*

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

- `EXISTS` tests if a result is not empty
- In this example, for each row of *Sailors* we tests if the result of the inner query is non-empty
- The existence of S in the subquery is a called a correlation
- Note that this is a proper use of the `*` in the `SELECT` clause

UNIQUE

- `UNIQUE` returns true if no row appears twice in the answer to a subquery
- `UNIQUE` of an empty set returns **TRUE**
- Not supported by postgresql

Set-Comparison Operators

- Existential qualifiers
- `op ANY` and `op ALL`, where `op` is one of:

  $<, <=, =, <>, >=, >$
- *Q22. Find sailors whose rating is better than some sailor called Horatio*

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

Set-Comparison Operators...

- *Q24. Find the sailors with the highest rating:*

```
SELECT s.sid
FROM Sailors S
WHERE S.rating >= ALL (SELECT S2.rating
                       FROM Sailors S2)
```

Example

- *Q9. Find the names of sailors who have reserved all boats*

```
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))
```

sname
Dustin

- We compute the set of all boats, then we remove:
- For each sailor S, the set of boats reserved by S
- And for each sailor, we check that this result is empty (that is, the set of boats minus the set of boats reserved by S is empty)

Aggregate Operators

- Sometimes we need to compute some a value that depends on multiple rows
- SQL extends relational algebra with 5 aggregate operations, that can be applied to any column, say A:
  - `COUNT ([DISTINCT] A)`: Returns the number of (unique) values of the A column
  - `SUM ([DISTINCT] A)`: Returns the sum of all (unique) values of the A column
  - `AVG ([DISTINCT] A)`: Returns the average of all (unique) values of the A column
  - `MAX (A)`: Returns the maximum value of the A column
  - `MIN (A)`: Returns the maximum value of the A column

Aggregation Example

- *Q27.Find the name and age of the oldest sailor*

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = (SELECT MAX (S2.Age)
       FROM Sailors S2)
```

- In this case, the result of the subquery is a relation of one row and one column, the DBMS translates it into a value
- The following query would be illegal:

```
SELECT S.sname, MAX(S.age)
FROM Sailors S
```

- If a `SELECT` uses aggregation, it must use **only** aggregate operations (unless the query contains `GROUP BY`)

Aggregation Example

- The following query counts the number of rows in the table
- *Q28 Count the number of sailors*

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

- *Q28 Count the number of different sailor names*

```
SELECT COUNT(DISTINCT sname) as Total
FROM Sailors S
```

`GROUP BY` and `HAVING`

- Sometimes we need to aggregate subsets of the relation
- Example: Q31. *Find the age of the youngest sailor for each rating level*
- Instead of writing one query for each rating (rather tedious and error prone) we can use `GROUP BY`

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
GROUP BY S.rating
```

GROUP BY

- General format:

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

- **select-list**: columns and aggregate operations. **Every column** in the select-list should also appear in the grouping list.
- The expressions in the **group-qualification** in the `HAVING` must have a **single** value per group:
  - A column here must appear as the argument to the aggregation operator
  - or it must also appear in the **grouping-list**
- If `GROUP BY` is omitted, the table is considered a single group.

Semantics of GROUP BY, HAVING query

1. First step, create cross-product of tables
2. Apply qualification of WHERE
3. Eliminate unnecessary columns (keep those columns mentioned in the SELECT, GROUP BY and HAVING)
4. Sort the table according to the GROUP BY
5. Apply the group qualification in the having clause
6. Apply the aggregation and the SELECT and generate one row
7. Optional: If SELECT DISTINCT then remove duplicates

Semantics of GROUP BY, HAVING ...

✣ **Q32** Find the age of the youngest sailor who is eligible to vote (at least 18 years old) for each rating level with at least 2 sailors

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

Semantics of GROUP BY, HAVING ...

1 First step, create cross-product of tables. Because only one relation is involved, then return original relation

sid	*sname*	*rating*	*age*
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	35
85	Art	3	25.5
95	Bob	3	63.5

Semantics of GROUP BY, HAVING ...

2 Apply qualification of WHERE:

```
WHERE S.age >= 18
```

sid	*sname*	*rating*	*age*
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
74	Horatio	9	35
85	Art	3	25.5
95	Bob	3	63.5

Semantics of `GROUP BY, HAVING` ...

3 Eliminate unnecessary columns (keep those columns mentioned in the `SELECT, GROUP BY` and `HAVING`): *rating*, *age*

rating	*age*
7	45
1	33
8	55.5
8	25.5
10	35
7	35
9	35
3	25.5
3	63.5

Semantics of `GROUP BY, HAVING` ...

4 Sort the table according to the `GROUP BY`

```
GROUP BY S.rating
```

rating	*age*
1	33
3	25.5
3	63.5
7	45
7	35
8	55.5
8	25.5
9	35
10	35

Semantics of `GROUP BY, HAVING` ...

5 Apply the group qualification in the having clause

```
HAVING COUNT(*) > 1
```

rating	*age*
3	25.5
3	63.5
7	45
7	35
8	55.5
8	25.5

- Note that `WHERE` happens **before** `HAVING`

Semantics of `GROUP BY, HAVING` ...

6 Apply the aggregation and the `SELECT` and generate one row:

```
SELECT S.rating, MIN (S.age)
```

- This is the result for this query

rating	*minage*
3	25.5
7	35
8	25.5

7 **Optional**: If `SELECT DISTINCT` then remove duplicates

Another example

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

minage
25.5
35

More aggregate queries

- *Q33. For each red boat, find the number of reservations for this boat*

  ```
  SELECT B.bid, COUNT(*) AS reservationcount
  FROM Boats B, Reserves R
  WHERE R.bid = B.bid AND B.color = ’red’
  GROUP BY B.bid
  ```

bid	*reservationcount*
102	3
104	2

More aggregate queries...

- This query is illegal:

  ```
  SELECT B.bid, COUNT(*) AS reservationcount
  FROM Boats B, Reserves R
  WHERE R.bid = B.bid
  GROUP BY B.bid
  HAVING B.color = ’red’
  ```

  - **only columns that appear in the `GROUP BY` can appear in the `HAVING` clause**
  - Unless they appear as arguments to an aggregate in the `HAVING` clause

`NULL` values

- Remember, `NULL` means a value is **unknown**
- What happens when we compare a value against `NULL`?
  - The result of <, <=, =, <>, >=, > is `NULL` if one operand is `NULL`
  - To test if a value is (not) null use `IS NULL` (`IS NOT NULL`)
- Arithmetic operations return `NULL` if one of their arguments is `NULL`

Boolean operations with NULL

- Boolean operations have to be extended to support an **unknown** value (a value that `IS NULL`)
- In the following table, *a* can be unknown

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

More on `NULL`

- Aggregate operations discard `NULL` values
  - In this case, the `NULL` values should be discarded first
  - If they are applied only to `NULL` values, the result is `NULL` (with the exception of `COUNT`)
- Impact on `WHERE`:
  - Any row that is NULL is also eliminated (row does not evaluate to TRUE)
  - This has impact in `EXISTS`
- Duplicates:
  - Two rows are identical if their corresponding columns are equal or are NULL
  - This definition avoids the problem of comparing `NULL` vs. `NULL` (what is the result?)

Outer Joins

- **Outer Join**: A variant of the join operation that relies on `NULL` values
- Example: $Sailos \bowtie_c Reserves$
- Tuples in Sailors that do not match a row in Reserves do not appear in the result
- In an outer join, Sailors rows without a matching Reserves row appear exactly once in the result, with the columns from Reserves assigned `NULL` values

Variations of Outer Joins

- **Left outer join** of S and R shows every single S row, filling the unmatched rows with `NULL`
- **Right outer join** of S and R shows every single R row, filling the unmatched rows with `NULL`
- **Full outer join** of S and R shows every single S and R row, filling the unmatched rows with `NULL`

Example

- Natural Join

```
SELECT s.sid, R.bid
FROM Sailors S NATURAL JOIN Reserves R
```

- Left Outer Join

```
SELECT s.sid, R.bid
FROM Sailors S NATURAL LEFT OUTER JOIN Reserves R
```

- Right Outer Join

```
SELECT s.sid, R.bid
FROM Sailors S NATURAL RIGHT OUTER JOIN Reserves R
```

- Full Outer Join

```
SELECT s.sid, R.bid
FROM Sailors S NATURAL FULL OUTER JOIN Reserves R
```

Example...

- Results of previous queries

sid	*bid*
22	101
22	102
22	103
22	104
31	102
31	103
31	104
64	101
64	102
74	103

Natural Join

sid	*bid*
22	101
22	102
22	103
22	104
29	
31	102
31	103
31	104
32	
58	
64	101
64	102
71	
74	103
85	
95	

Left Outer Join

sid	*bid*
22	101
22	102
22	103
22	104
31	102
31	103
31	104
64	101
64	102
74	103

Right Outer Join

sid	*bid*
22	101
22	102
22	103
22	104
29	
31	102
31	103
31	104
32	
58	
64	101
64	102
71	
74	103
85	
95	

Full Outer Join

Complex Integrity Constraints

- We have learn how to specify constraints on the keys of a table
- But what about a constraint on the values that a given row in a table can take?
- For that we use **table constraints**
- Example: we want to restrict a rating to values between 1 and 10.

```
CREATE TABLE Sailors (sid INTEGER,
                      sname CHAR(10),
                      rating INTEGER,
                      age    REAL,
                      PRIMARY KEY (sid),
                      CHECK (rating >= 1 AND rating <= 10))
```

A more complex example

- We want to constraint that the *Interlake* boats cannot be reserved:

```
create table Reserves (
        sid     INTEGER,
        bid     INTEGER,
        day     DATE,
        PRIMARY KEY (sid, bid, day),
        FOREIGN KEY (sid) REFERENCES Sailors
           ON DELETE CASCADE,
        FOREIGN KEY (bid) REFERENCES Boats
           ON DELETE CASCADE,
        CONSTRAINT noInterlakeRes
        CHECK ('Interlake' <>  (SELECT B.Bname
                                FROM Boats B
                                WHERE B.bid = Reserves.bid)))
```

- Unfortunately postgresql does not support subqueries in the `CHECK` expression

CHECK

- The condition of the check has to be a valid expression evaluating to a boolean result.
- Every time a row is inserted o modified, the `CHECK` expression is evaluated.

You can create your own domains/types

- `CREATE DOMAIN`

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
      CHECK (VALUE >= 1 AND VALUE <= 10)
CREATE DOMAIN counterval INTEGER DEFAULT 1
      CHECK (VALUE >= 0)
```

- Not supported by postgresql
- Once defined, you use it as any other type in a `CREATE TABLE`:

```
rating ratingval,
```

- Internally, the `DOMAIN` behaves just like the underlying type used to define it, ie. we can compare *ratingval* and *counterval* variables (they are just integers)
- Ideally, we would like to get an error when we compare two different domain variables.

CREATE TYPE

- SQL:1999 introduces the notion of **distinct types**

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
      CHECK (VALUE >= 1 AND VALUE <= 10)
CREATE DOMAIN counterval INTEGER DEFAULT 1
      CHECK (VALUE >= 0)
```

- These statements create two new types: **ratingval** and **counterval**
- Not supported by postgresql
- Now we cannot combine integers with variables of these types. integer, *ratingval* and *counterval* are treated as totally different and incompatible types between each other.