SQL: Data Manipulation Language

Anne Denton

Department of Computer Science
North Dakota State University

Outline

Table of Contents

Basic Form of SELECT

- Even the most basic form of SQL select does more than the relational algebra selection

  SELECT <attribute list>
  FROM <table list>
  WHERE <condition>

- <attribute list> is a comma-separated list of attributes, or * to indicate that all attributes are to be listed
- FROM <table list>
  - When only one table is listed, the query corresponds to a selection or projection
- <condition> is the selection criterion applied to the set of rows

SELECT from more than one table

- Listing a comma-separated list of multiple tables allows joining them, provided you also specify the equality of join attribute in the WHERE–clause
- If only a comma-separated list is given, without WHERE–clause, the result will be Cartesian product which is not normally useful
- Instead of a FROM <table list> you can provide a join clause, see later

Single-table SELECT

- A single-table select such as

```
SELECT student_fname, student_lname
FROM Student
WHERE sid = 4711 OR sid = 815;
```

- Selects from rows (relational algebra σ)
- Projects to columns (relational algebra π)

Duplicate elimination

- Duplicates NOT eliminated
  - Otherwise problems with aggregate functions
  - Duplicate elimination computationally expensive (requires sorting)
- To eliminate duplicates use

```
SELECT DISTINCT major_dept
FROM Student;
```

Question 1

A `SELECT DISTINCT` query is

1. Faster than a regular select query
2. Has the same complexity as doing the select plus sorting the result
3. Slower than a selection and sorting because every record in the output has to be compared with every other record

Consider tables from earlier SQL notes

Consider tables from early SQL notes

Student

sid	student_fname	student_lname	major_dept
42	John	Doe	2740
4711	Jane	Smith	2740
815	Jack	Box	null

Department

dept_no	dept_name	building
2740	CSci	QBB
2755	Physics	South. Eng.

Inner join

- Selection from multiple tables, typically requires equality of join attribute

```
SELECT *
FROM Student, Department
WHERE major_dept = dept_no;
```

- Logic is based on Cartesian product
- Omitting `WHERE` clause results in Cartesian product which is not usually helpful
- Implementation may not actually use Cartesian product
  - Hashing is often used
  - When few items are selected, data structures that enable fast access to key (like indexes) may be used
  - The query optimizer determines how query is evaluated

Question 2 (Multiple answers can be correct)

SELECT * FROM Student, Department

1. Gives, as a result, the inner join of both tables
2. Gives, as a result, the Cartesian product of both tables
3. Is not particularly useful in practice unless a WHERE clause is added

Question 3 (Multiple answers can be correct)

Whether a join query is evaluated by actually computing a Cartesian product as intermediate step

1. Depends on the specific query and the data
2. Depends on how the user writes the query
3. Depends on the query optimizer

Inner join

- Why is this an inner join?

```
SELECT *
FROM Student, Department
WHERE major_dept = dept_no;
```

  - Not an outer join: Only records are included that exist in both tables, and for which the join attribute is not null and is equal in both tables
  - Not a natural join: The join attribute is listed twice, and you have to explicitly select the columns you want to avoid that

42	John	Doe	2740	2740	CSci	QBB
4711	Jane	Smith	2740	2740	CSci	QBB

Newer notation for inner join

- Newer notation for inner join:

```
SELECT *
FROM Student INNER JOIN Department
ON major_dept = dept_no;
```

- Advantages over conventional notation
  - Generalizes to outer join in a straightforward way
  - Earlier outer join notations were highly vendor dependent
  - Makes clear which equality condition relates to join

Question 4 (Multiple answers can be correct)

```
SELECT * FROM Student INNER JOIN Department ON
major_dept = dept_no;
```

instead of

```
SELECT * FROM Student, Department WHERE
major_dept = dept_no;
```

1. Is better for joining tables because it avoids executing the Cartesian product
2. Makes queries more easily readable
3. Is a worse approach for joining tables

Natural join in SQL?

- Similar notation to inner join, but no `ON` clause. Join attributes are assumed to have the same name in both tables
- Eliminates duplicate column automatically
- Not recommended by practitioners, so we will not discuss it further

Table of Contents

Left outer join

- Left outer join includes records from first table that have no match in second table

```
SELECT *
FROM Student LEFT OUTER JOIN Department ON
major_dept = dept_no;
```

42	John	Doe	2740	2740	CSci	QBB
4711	Jane	Smith	2740	2740	CSci	QBB
815	Jack	Box	<null>	<null>	<null>	<null>

Right outer join

- Right outer join included records from second table that have no match in first

```
SELECT *
FROM Student RIGHT OUTER JOIN Department ON
major_dept = dept_no;
```

42	John	Doe	2740	2740	CSci	QBB
4711	Jane	Smith	2740	2740	CSci	QBB
<null>	<null>	<null>	<null>	2755	Physics	South Eng.

Question 5 (Multiple answers can be correct)

The terms "LEFT" and "RIGHT" for OUTER JOINS refer to

1. The order in which attributes will be listed in the output
2. Whether tables are stored as left or right tables
3. The order in which tables are listed in the query, with the first one considered left and the second one right
4. Whether unmatched rows from the first (LEFT) and second (RIGHT) table respectively are being added

Full outer join

- Full outer join includes both

```
SELECT *
FROM Student FULL OUTER JOIN Department ON
major_dept = dept_no;
```

42	John	Doe	2740	2740	CSci	QBB
4711	Jane	Smith	2740	2740	CSci	QBB
815	Jack	Box	<null>	<null>	<null>	<null>
<null>	<null>	<null>	<null>	2755	Physics	South Eng.

Question 6

The two columns that represent the join attribute in an outer join

1. Always have the same value and if one is NULL the other is too
2. Always have the same value, but one can be NULL even while the other is not
3. Can have mismatching values

Prefixing

- Consider:

```
CREATE TABLE StudentEmail( sid INT PRIMARY KEY, sEmail
VARCHAR(50),
FOREIGN KEY (sid) REFERENCES Student);

INSERT INTO StudentEmail VALUES (42,

'John.Doe@someuni.edu');
```

- Attributes have same name in two tables
  - Distinguish them by prefixing them with table name
- `SELECT * FROM Student LEFT OUTER JOIN StudentEmail ON Student.sid=StudentEmail.sid;`

Question 7 (Multiple answers can be correct)

When the join attribute has the same name in both tables

1. The "ON att1 = att2" portion of the OUTER JOIN can be omitted
2. The attribute name has to be prefixed with the respective table names

Aliasing

- Now consider aliasing, e.g. in when a table is to be joined with itself:

```
CREATE TABLE Course( cID INT PRIMARY KEY,
cName VARCHAR(50), prerequisite INT,
FOREIGN KEY (prerequisite) REFERENCES Course(cID));
INSERT INTO Course VALUES (213, 'Modern Software
Development',NULL);
INSERT INTO Course VALUES (366, 'Database Systems',213);
```

- The keyword `AS` is used for defining the alias:

```
SELECT a.cID, a.cName, b.cID, b.cName FROM Course AS a,
Course AS b WHERE a.cID = b.prerequisite;
```

Notes on aliasing

- A table referencing itself can happen for hierarchies, as above
- One table can also reference the same table twice, e.g. for networks like webpage1 linking to webpage2
- Aliasing is also sometimes used to avoid typing long table names
- Note that the aliased name comes in effect already in the `SELECT` clause although it is only defined in the `FROM` clause

Table of Contents

Wildcards

- Using wildcards, i.e., substring comparisons
  - Underscore (_) stands for a single character
  - Percent (%) stands for arbitrary number of characters
  - You must use `LIKE` instead of equal sign! (unlike transparent treatment of wildcards Linux)

```
SELECT * FROM Student WHERE student_fname LIKE
'Ja%';
```

Checking for NULL

- Syntax to check for NULL
- You must use `IS` instead of equal sign!
  ```
  SELECT * FROM Student WHERE major_dept IS NULL;
  ```
- Note that two NULL values may not be null for the same reason!
- Similar to Java: `if (x == NaN)` is false even when `x` is `NaN`

Question 8 (Multiple answers can be correct)

```
SELECT * FROM Student WHERE major_dept = NULL;
```

1. Returns those records in the student table where the major_dept is NULL
2. Does not return anything even if there are records for which the major_dept is NULL
3. Uses incorrect syntax for checking for NULL

Mathematical and string operations

- Apply mathematical functions on number types (often useful)
- Concatenate strings
  `SELECT 'Department '|| dept_name FROM Department;`
  (becoming less commonly used, since string processing is typically done outside database)

ORDER BY

- You can order the output

```
SELECT * FROM Student
ORDER BY student_lname;
```

- Very useful, because sorting in databases is typically faster than outside
- The logic of relations makes no guarantee for row ordering
- If ORDER BY queries are expected often, consider adding an index on the respective attribute

Table of Contents

Example problem

- Consider the following Enrollment table:

```
CREATE TABLE Enrollment
( sID INT,
cID INT,
grade DECIMAL(4,1),
PRIMARY KEY(sid,cid),
FOREIGN KEY (sid) REFERENCES Student,
FOREIGN KEY (cid) REFERENCES Course);
INSERT INTO Enrollment VALUES (42,366,90.0);
INSERT INTO Enrollment VALUES (4711,366,95.0);
INSERT INTO Enrollment VALUES (42,213,98.0);
INSERT INTO Enrollment VALUES (815,213,80.0);
```

- Which students take 366 and 213?

Solution using set theoretic operations

- Which students take 366 and 213?

```
SELECT sid FROM Enrollment WHERE cID = 366
INTERSECT
SELECT sid FROM Enrollment WHERE cid = 213;
```

Choices of set-theoretic operations

- Alternatives

Set Theory	SQL
$\cap$ (Intersection)	`INTERSECT`
$\cup$ (Union)	`UNION`
$-$ (Set Difference)	`EXCEPT`

- Example of set difference
  - Which students have taken 366 but not 213?

```
SELECT sid FROM Enrollment WHERE cid = 366
EXCEPT
SELECT sid FROM Enrollment WHERE cid = 213;
```

Question 9 (Multiple answers can be correct)

Set theoretic operations

1. Can always be replaced with queries that use a combination of AND, OR, and NOT in the WHERE clause
2. Allow implementing conditions that cross different rows or even columns

Table of Contents

Aggregate functions

- Aggregate functions allow summing, averaging, etc. the records of a table:

```
SELECT AVG(grade)
FROM Enrollment
WHERE cID = 366;
```

- Available functions are:

`SUM` (sum of the values)
`MAX` (largest of the values)
`MIN` (smallest of the values)
`AVG` (average)
`COUNT` (number of values)

`MEDIAN` (statistical median) slower since it requires sorting, and partial results from separate tables cannot be reused

Question 10 (Multiple answers can be correct)

For which of the following aggregate functions can the result be computed by aggregating partial aggregates

1. SUM
2. MAX
3. MEDIAN

GROUP BY

- Aggregate functions are particularly useful when applied to multiple groups:

```
SELECT AVG(grade)
FROM Enrollment
GROUP BY cID;
```

- Use of the `GROUP BY` clause can be a little frustrating, because any expression that is not included in the aggregate function has to be listed in the `GROUP BY` clause, for example:

```
SELECT Student.sid, student_fname, student_lname,
AVG(grade)
FROM Student INNER JOIN Enrollment ON Student.sid =
Enrollment.sid
GROUP BY Student.sid, student_fname, student_lname;
```

Question 11 (Multiple answers can be correct)

Consider an Enrollment table with student and section identifiers, a Student table, and a Section table

1. Average grade per student can be computed from the enrollment table using a GROUP BY clause
2. Average grade per course can be computed from the enrollment table using a GROUP BY clause
3. Calculating the average grade per course requires a join with the course table as well as a GROUP BY clause

Table of Contents

Reuse of results

- SQL gains much of its power through reusing query results in various places:
- Remember that the result of any `SELECT` query is again a table that can be used as input to others
- A table with just one column can furthermore be used in places where sets are expected

Background: Comparison with explicit set

- Compare with explicit sets (important later for nested queries!)
  ```
  SELECT * FROM Enrollment WHERE cid IN (213, 313);
  ```
- Equivalent to
  ```
  SELECT * FROM Enrollment WHERE cid = 213 OR cid = 313;
  ```

Example of a nested query

- Example: Find all those courses in which there are students who have not declared a major

```
SELECT *
FROM Course
WHERE cid IN  (SELECT cid
               FROM Student INNER JOIN Enrollment
               ON Student.sid = Enrollment.sid
               WHERE major_dept IS NULL);
```

Advantages of unnested queries

- Nested queries may appear complicated, but they are actually particularly useful for human understanding
- Query optimizers work better on queries that are not nested
- Could the earlier query have been done without nesting?

```
SELECT Course.cid, Course.cName,
Course.prerequisite
FROM Course INNER JOIN Enrollment ON
Course.cid = Enrollment.cid
INNER JOIN Student ON Enrollment.sid =
Student.sid
WHERE Student.major_dept IS NULL;
```

Question 12 (Multiple answers can be correct)

Nesting of queries

1. Is important for solving complex queries
2. Allows replacing any table or set in a query with another query
3. Usually results in faster execution than if the result were computed without nesting

Operators for Nested Queries

- `= ANY` is equivalent to `IN`
- `>, >=, <=, <, <> ANY` evaluate to true if at least one of the elements of the multiset fulfills the condition
- `>, >=, <=, <, <> ALL` evaluate to true if all of the elements of the multiset fulfill the condition
- `EXISTS` evaluates to true if there is any element in the multiset
- `NOT EXISTS` evaluates to true if there are no elements in the multiset
- `UNIQUE` evaluates to true if there are not duplicate elements in the multiset (true set)

Question 13 (Multiple answers can be correct)

The `IN` in the following query `SELECT * FROM Enrollment WHERE cid IN (213, 313);` could be replaced with

1. `= ANY`
2. `= ALL`
3. `EXISTS`
4. None of the other commands

Summary

In general a query in SQL has the following form:

```
SELECT <attribute and function list>
FROM <table list>
[WHERE <condition>]
[GROUP BY <grouping attribute(s)>]
[HAVING <group condition>]
[ORDER BY <attribute list>]
```

The clauses in [] are optional. (We didn't discuss the HAVING clause).

Table of Contents

Insert

- There are two forms of insertion statements, providing all attributes:

```
INSERT INTO Course
VALUES(372,'Comparative Programming Languages',213);
```

  If for some of the values NULL or the default value is ok, specifying a subset is possible:

```
INSERT INTO Course(cID,cName) VALUES(160,'CSci I');
```

- In the first notation attributes are entered according to the order in the definition of the table
- In the second notation the order is determined by the list of attributes given with the table name
- In the first notation, values can be explicitly set to `NULL` or `DEFAULT`

Inserting values as result of a query

- The values in the insert statement can be the result of a query
- If the values in the original table change, the new table will NOT be updated
- Use a `VIEW` or `MATERIALIZED VIEW` if you want them to be updated

Automatic Key Generation

- The most common way of automatically creating a primary key in PostGreSQL is to use the data type `SERIAL`
- Some other DBMSs require a separate definition of a `SEQUENCE`

Table of Contents

Basic deletion syntax

- Deletions are done as follows:

```
DELETE
FROM Course
WHERE cID = 366;
```

- The `WHERE` clause specifies conditions on the tuples which are updated
- Many tuples may be deleted with one `DELETE` statement, and care is advised when doing so! For example, the following will delete all records from table Course:

```
DELETE FROM Course;
```

Question 14 (Multiple answers can be correct)

`DELETE FROM Course;` for an existing table `Course`

1. Does not delete anything because no condition was given
2. Deletes every row in Course, because no condition is given
3. Will result in an error message

Impact of delete

- Deletions are done from one table at a time
- Remember that if `ON DELETE CASCADE` was chosen as referentially triggered action for a foreign key that references the primary key of the table from which you are deleting, records from other tables may be deleted automatically!

UPDATE

- Updates, as insertions and deletions are done per record:

```
UPDATE Course
SET prerequisite = NULL
WHERE cID = 366;
```

- The `WHERE` clause specifies conditions on the tuples which are updated
- The `SET` clause may use results from other relations:

```
UPDATE Course
SET prerequisite = (SELECT cID
FROM Course
WHERE cName = 'Modern Software Development')
WHERE cID = 366;
```

Impact of update

- Tuples are modified in one relation at a time
- Referentially triggered actions may apply in much the same way as for delete, provided `ON UPDATE CASCADE`, `ON UPDATE SET NULL`, or `ON UPDATE SET DEFAULT` were specified
- Many tuples may be modified with one UPDATE statement:

```
UPDATE Enrollment
SET grade = grade * 1.2;
```

Table of Contents

Views

- Allows making programming for different user views more transparent
- Important tool when normalization considerations result in many small tables
- Can be used for buffering results of aggregate functions
- Two different types of `VIEW` exist in most DBMSs
- Especially useful for tables that include derived attributes
  - For regular views the query that generated them will be executed whenever they are referenced
  - For materialized views, the view itself is stored and is updated using whatever strategy appears best to the query optimizer; typically they will be modified for updates to the base tables, rather than upon receiving a query

Example of a view

- Storing a query as view

```
CREATE VIEW Grade_report
AS SELECT s.sid, s.student_fname,
s.student_lname, c.cid, c.cname, e.grade
FROM Student s, Course c, Enrollment e
WHERE s.sid = e.sid AND c.cid = e.cid;
```

- Can be queried like a table

```
SELECT * from Grade_report;
```

- Updates have to be done to base tables

Question 15 (Multiple answers can be correct)

If the intention is that query results will be maintained consistent with the underlying base tables, even if the base tables change, they can be inserted into a

1. `VIEW`
2. `MATERIALIZED VIEW`
3. `TABLE`

Attribute names in views

- Attribute names
  - May be automatically taken over from the base relations, as in the example above
  - They may also be explicitly supplied as in the example below

```
CREATE VIEW Course_stats (cID,avgGrade)
AS SELECT cID, AVG(grade)
FROM Enrollment
GROUP BY cID;
```

- New attribute names will be used

```
SELECT * from Course_stats;
```