CS143: Query and Update in SQL

Book Chapters
(4th) Chapter 4.1-6, 4.8-10, 3.3.4
(5th) Chapter 3.1-8, 3.10-11
(6th) Chapter 3.1-9, 4.1, 4.3

Things to Learn
• DML for SQL

SQL
• Structured Query Language
• The standard language for all commercial RDBMS
• SQL has many aspects
  – DDL: schema definition, constraints, index, …
  – DML: query, update, …
  – triggers, transaction, authorization, …
• In this lecture, we cover the DML aspect of SQL
  – How to query and modify existing databases
• SQL and DBMS
  – SQL is high-level description of user’s query
    * No concrete procedure for query execution is given
  – The beauty and success of DBMS
    * The system understands the query and find the best way possible to execute it automatically

Example to Use in the Class
• School information
  – Student(sid, name, age, GPA, address, …)
  – Class(dept, cnum, sec, unit, title, instructor, …)
  – Enroll(sid, dept, cnum, sec)
Basic SELECT statement

- **Query 1:** Find the titles and instructors of all CS courses

**Semantics**
- Interpret and write FROM $\rightarrow$ WHERE $\rightarrow$ SELECT
  - FROM: the list of tables to look up
  - WHERE: conditions to meet
  - SELECT: the attributes to return
- *Conceptual* execution (table cursor diagram)

```
			Check condition
        Return attr
```

**General SQL statement**

- **SELECT** $A_1, \ldots, A_n$
- FROM $R_1, \ldots, R_m$
- WHERE $C$
- $\equiv \pi_{A_1, \ldots, A_n}(\sigma_C(R_1 \times \cdots \times R_m))$
- **SELECT** $*$: all attributes
- **SELECT** is “projection” not “selection”: can be confusing
- SQL does not remove duplicates: Major difference between SQL and relational algebra
  - More examples will follow

**SQL join**

- **Query 2:** Find the names and GPAs of all students taking CS classes

  - Conceptually WHERE $R$, $S$
  (Table join diagram)
For every pair of tuples from R and S, we check condition and produce output

Notes:

- S, E: tuple variable
  - renaming operator
  - We can consider that S and E are variables that bind to every pair of tuples
- Attributes can also be renamed
  - GPA (AS) grade
- DISTINCT: remove duplicates in the results

WHERE conditions

- **Query 3**: All student names and GPAs who live on Wilshire

- `%`: any length (0–∞) string
  - one character
  - `%'Wilshire%'`: Any string containing Wilshire

Q: What does `'__%` mean?

- Other useful string functions: `UPPER()`, `LOWER()`, `CONCAT()`, ...
Set operators

- \( \cap \): INTERSECT, \( \cup \): UNION, \( - \): EXCEPT
- Can be applied to the result of SELECT statements or to relations
- **Query 4:** All names of students and instructors

**Important points to note**

- Set operators should have the same schema for operands
  * In practice, it is okay to have just compatible types
- Set operators follow set semantics and remove duplicates
  * Set semantics is well understood for set operations. Not many people know bag semantics.
  * Efficiency
- To keep duplicates, use \texttt{UNION ALL, INTERSECT ALL, EXCEPT ALL}

**Query 5:** Find ids of all students who are not taking any CS courses.

- MySQL supports only \texttt{UNION}, not \texttt{INTERSECT} or \texttt{EXCEPT}. 
Subqueries

- SELECT statement may appear in WHERE clause
  - Treated the same as regular relations
  - If the result is one-attribute one-tuple relation, the result can be used like a 'value'

Scalar-value subqueries

- **Query 6:** Find the student ids who live at the same addr as the student with id 301

- **Q:** Can we rewrite it without subquery?

- **Notes:**
  - There is a whole theory about whether/how to rewrite a subquery to non-subquery SQL
  - The basic result is we can rewrite subqueries as long as we do not have negation.
  - With negation, we need EXCEPT
  - One of the reasons why relational model has been so successful
    * Because it is easy to understand and model, we can design and prove elegant theorems.
    * Many efficient and provable algorithms.

Set membership (IN, NOT IN)

- **Query 7:** Find all student names who take CS classes.
  
  Idea: Find the set of sids that take CS classes first. Then check whether any student’s id belong to that set or not.

  - **IN** is a set membership operator
    * \((a \text{ IN } R)\) is TRUE if \(a\) appears in \(R\)
Q: Can we write the same query without subqueries?

Q: Are the above two queries equivalent?

Q: Why we care about duplicates so much?

- **Query 8**: Find the names of students who take no CS classes

Q: Can we rewrite it without subqueries?

Set comparison operator (> ALL, < SOME, ...)

- **Query 9**: Find the ids of students whose GPA is greater than all students of age below 18

  - ALL is the universal quantifier ∀
• **Query 10**: Find the names of students whose GPA is better than at least one other student of age ≥ 18

  - **SOME** is the existential quantifier \( \exists \)

**Other Set comparison operators: > ALL, <= SOME, = SOME, ..., etc.**

  - (<> ALL) \( \equiv \) (NOT IN), (= SOME) \( \equiv \) IN

**EXISTS and Correlated subqueries**

• **Query 11**: Find the names of the students who take CS courses

  - **EXISTS**: WHERE EXISTS(SELECT ... FROM ... WHERE)
    * True if SELECT .. FROM .. WHERE returns at least one tuple

**Correlated subquery interpretation:**

  * Outer query looks at one tuple at a time and binds the tuple to \( S \)
  * For each \( S \), we execute the inner query and check the condition
  * This is just interpretation. **DBMS executes it more efficiently but get the same result** (but not necessarily MySQL).

**Subqueries in FROM clause**

  - Considered as a regular relation

**Example**: SELECT name
  FROM (SELECT name, age FROM Student) \( S \)
  WHERE age > 17
  * A subquery inside FROM **MUST** be renamed
  * Student names with age > 17

• **Q**: Do subqueries make SQL more expressive than relational algebra?
Aggregates

- The operators so far check the condition “tuple-by-tuple”
- They never “summarize” multiple tuples into one.
  For example, ‘SUM’, ‘AVG’ of GPA is not possible.

- Aggregate function (aggregate diagram)

- **Query 12:** Find the average GPA

- Common aggregate functions: **SUM, AVG, COUNT, MIN, MAX** on single attribute or **COUNT(*)**.

Problems of Duplicates

- **Query 13:** The number of students taking CS classes

- **Query 14:** The average GPA of the students taking CS classes

GROUP BY clause

- Sometimes, we want to get separate statistics for each group of tuples

  **Example:**

<table>
<thead>
<tr>
<th>Age</th>
<th>AVG(GPA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>3.7</td>
</tr>
<tr>
<td>19</td>
<td>2.1</td>
</tr>
<tr>
<td>20</td>
<td>3.1</td>
</tr>
</tbody>
</table>

  But **AVG()** takes average over **all** tuples.
• **Query 15:** Find the average GPA for each age group

  **Q:** Is the following query meaningful?

  ```sql
  SELECT sid, age, AVG(GPA)
  FROM Student
  GROUP BY age
  ```

  – SELECT can have only attributes that have a single value in each group or *aggregates*

• **Query 16:** Find the number of classes each student is taking

  **Q:** What about the students who take no classes?

  **Comments:** We will learn about outer join that can address this issue later.

  **HAVING clause**

• **Query 17:** Find students who take two or more classes

  – Conditions on aggregates should appear in the HAVING clause.

  **Q:** Can we rewrite the query without HAVING clause?

  – In general, we can rewrite a query not to have a HAVING clause.
ORDER BY clause

- Sometimes we may want to display tuples in a certain order. For example, order all students by their GPA

- SELECT sid, GPA
  FROM Student
  ORDER BY GPA DESC, sid ASC
  - All students and GPAs, in the descending order of their GPAs and the ascending order of sids. Default is ASC if omitted.
  - Does not change SQL semantics. Just makes the display easier to look at and understand

General SQL SELECT statement

- SELECT attributes, aggregates
  FROM relations
  WHERE conditions
  GROUP BY attributes
  HAVING conditions on aggregates
  ORDER BY attributes, aggregates

- Evaluation order: FROM → WHERE → GROUP BY → HAVING → ORDER BY → SELECT
Data Modification in SQL (INSERT/DELETE/UPDATE)

- **Insertion**: \texttt{INSERT INTO} \textit{Relation Tuples}
  
  - \textbf{Query 18}: Insert tuple (301, CS, 201, 01) to \textit{Enroll}?

  - \textbf{Query 19}: Populate \textit{Honors} table with students of GPA > 3.7?

- **Deletion**: \texttt{DELETE FROM} \textit{R WHERE} \textit{Condition}
  
  - \textbf{Query 20}: Delete all students who are not taking classes

- **Update**: Update \textit{R}
  
  \texttt{SET A1 = V1, A2 = V2, …, An = Vn WHERE} \textit{Condition}

  - \textbf{Query 21}: Increase all CS course numbers by 100
NULL and Three-valued logic

• Arithmatic operators and comparison

Q: SELECT name 
    FROM Student 
    WHERE GPA * 100/4 > 90 
What should we do if GPA is NULL?

- Q: What should be the value for GPA * 100/4?

- Rule: Arithmatic operators with NULL input returns NULL 

- Q: What should be NULL > 90?

- Rule: Arithmatic comparison with NULL value return Unknown
    - SQL is Three-valued logic: True, False, Unknown
    - SQL returns only True tuples
    - GPA * 100/4 > 90 does not return a tuple if GPA is NULL

• Three-valued logic

- Q: GPA > 3.7 AND age > 18. What if GPA is NULL and age < 18?

- Q: GPA > 3.7 OR age > 18. What if GPA is NULL and age < 18?
Truth table

* AND: U AND T = U, U AND F = F, U AND U = U
* OR: U OR T = T, U OR F = U, U OR U = U

- NOT Unknown = Unknown. It’s not known

- SQL returns only True tuples

**Checking NULL**

- IS NULL or IS NOT NULL to check if the value is null.

**Set operators** \( (\cup, \cap, -) \)

- Q: What should be \{2.4, 3.0, NULL\} \(\cup\) \{3.6, NULL\}?

- Rule: NULL is treated like other values in set operators

**Aggregates**

- Q: | ID | GPA | SELECT AVG(GPA)  
|-----|-----|------------------|
| 1   | 3.0 | FROM Student     
| 2   | 3.6 | What should be the result?  
| 3   | 2.4 | What about COUNT(*)? COUNT(GPA)?  
| 4   | NULL|                  

- Rule: Aggregates are computed ignoring NULL value, except COUNT(*).

  * Too much information is lost otherwise.
  * COUNT(*) considers a NULL tuple as a valid tuple
  * When the input to an aggregate is empty, COUNT returns 0; all others return NULL.
SQL and bag semantics

- What is a bag (multiset)?
  - A set with duplicate elements
  - Order does not matter
  - Example: \{a, a, b, c\} = \{a, c, b, a\} \neq \{a, b, c\}

- SQL and bag semantics
  - Default SQL statements are based on bag semantics
    * We already learned the bag semantics
    * Except set operators (UNION, INTERSECT, EXCEPT), which use set semantics
  - We can enforce set semantics by using DISTINCT keyword

- Bag semantics for set operators
  - UNION ALL, INTERSECT ALL, EXCEPT ALL

  - Q: \{a, a, b\} \cup \{a, b, c\}?

  - Q: \{a, a, a, b, c\} \cap \{a, a, b\}?

  - Q: \{a, a, b, b\} − \{a, b, b, c\}?

- What rules still hold for Bag?

  - Q: Under bag semantics, \(R \cup S = S \cup R\) \(R \cap S = S \cap R\)?
    \(R \cap (S \cup T) = (R \cap S) \cup (R \cap T)\)?
    * Under bag semantics, some rules still hold, some do not
    * Consider, \(R = \{a\}, S = \{a\}, T = \{a\}\) to check the distributive rule.
OUTER join

- **Query 22**: How many classes does each student take?

  - **Q**: What about student 208, Esther? What should we print? What is the problem?

  - **Q**: Anyway to preserve dangling tuples?

- **OUTER JOIN** operator in **WHERE** clause:
  - **R LEFT OUTER JOIN S ON R.A = S.A**
    * Keep all dangling tuples from R by padding S attributes with NULL.
  - **R RIGHT OUTER JOIN S ON R.A = S.A**
    * Keep all dangling tuples from S by padding R attributes with NULL
  - **R FULL OUTER JOIN S ON R.A = S.A**
    * Keep all dangling tuples both from R and S with appropriate padding

- **Q**: How to rewrite the above query to include Esther?
Expressive power of SQL

- **Example**: All ancestors

<table>
<thead>
<tr>
<th>child</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan</td>
<td>John</td>
</tr>
<tr>
<td>John</td>
<td>James</td>
</tr>
<tr>
<td>James</td>
<td>Elaine</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- **Q**: Can we find all ancestors of Susan using SQL?

- **Example**: All reachable destination

<table>
<thead>
<tr>
<th>city 1</th>
<th>city 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- **Q**: Find all cities reachable from A?

- **Comments**: SQL92 does not support “recursion” and thus cannot compute the *transitive closure*.

- Recursion is supported in SQL3.
- WITH RECURSIVE R(A1, A2) AS ...
- e.g., WITH RECURSIVE Ancestor(child, ancestor) AS (    (SELECT * FROM Parent) UNION
  (SELECT P.child, A.ancestor FROM Parent P, Ancestor A WHERE P.parent = A.child )
)
- IBM DB2 supports it, while Oracle does not. Read Book 5.2 for detail