Lecture #02: Intermediate SQL

15-445/645 Database Systems (Fall 2020)
https://15445.courses.cs.cmu.edu/fall2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Relational Languages

Edgar Codd published a major paper on relational models in the early 1970s. Originally, he only defined the mathematical notation for how a DBMS could execute queries on a relational model DBMS.

The user only needs to specify the result that they want using a declarative language (i.e., SQL). The DBMS is responsible for determining the most efficient plan to produce that answer.

Relational algebra is based on **sets** (unordered, no duplicates). SQL is based on **bags** (unordered, allows duplicates).

2 SQL History

Declarative query lanaguage for relational databases. It was originally developed in the 1970s as part of the IBM **System R** project. IBM originally called it "SEQUEL" (Structured English Query Language). The name changed in the 1980s to just "SQL" (Structured Query Language).

The language is comprised of different classes of commands:

1. **Data Manipulation Language (DML):** `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.
2. **Data Definition Language (DDL):** Schema definitions for tables, indexes, views, and other objects.
3. **Data Control Language (DCL):** Security, access controls.

SQL is not a dead language. It is being updated with new features every couple of years. SQL-92 is the minimum that a DBMS has to support to claim they support SQL. Each vendor follows the standard to a certain degree but there are many proprietary extensions.

3 Aggregates

An aggregation function takes in a bag of tuples as its input and then produces a single scalar value as its output. Aggregate functions can only be used in `SELECT` output list.

Example: Get # of students with a '@cs' login. The following three queries are equivalent:

```
SELECT COUNT(*) FROM student WHERE login LIKE '%@cs';
```

```
SELECT COUNT(login) FROM student WHERE login LIKE '%@cs';
```

```
SELECT COUNT(1) FROM student WHERE login LIKE '%@cs';
```

Can use multiple aggregates within a single `SELECT` statement:

```
CREATE TABLE student (
   sid   INT PRIMARY KEY,
   name  VARCHAR(16),
   login VARCHAR(32) UNIQUE,
   age   SMALLINT,
   gpa   FLOAT
);

CREATE TABLE course (
   cid  VARCHAR(32) PRIMARY KEY,
   name VARCHAR(32) NOT NULL
);

CREATE TABLE enrolled (
   sid   INT REFERENCES student (sid),
   cid   VARCHAR(32) REFERENCES course (cid),
   grade CHAR(1)
);
```

Figure 1: Example database used for lecture

```
SELECT AVG(gpa), COUNT(sid)
  FROM student WHERE login LIKE '%@cs';
```

Some aggregate functions support the DISTINCT keyword:

```
SELECT COUNT(DISTINCT login)
  FROM student WHERE login LIKE '%@cs';
```

Output of other columns outside of an aggregate is undefined (e.cid is undefined below)

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid;
```

Thus, other columns outside aggregate must be aggregated or used in a GROUP BY command:

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid;
```

HAVING: Filters output results after aggregation. Like a WHERE clause for a GROUP BY

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
HAVING avg_gpa > 3.9;
```

4 String Operations

The SQL standard says that strings are **case sensitive** and **single-quotes only**. There are functions to manipulate strings that can be used in any part of a query.

Pattern Matching: The `LIKE` keyword is used for string matching in predicates.

- "%" matches any substrings (including empty).
- "_" matches any one character.

Concatenation: Two vertical bars ("||") will concatenate two or more strings together into a single string.

5 Output Redirection

Instead of having the result a query returned to the client (e.g., terminal), you can tell the DBMS to store the results into another table. You can then access this data in subsequent queries.

- **New Table:** Store the output of the query into a new (permanent) table.

  ```
  SELECT DISTINCT cid INTO CourseIds FROM enrolled;
  ```

- **Existing Table:** Store the output of the query into a table that already exists in the database. The target table must have the same number of columns with the same types as the target table, but the names of the columns in the output query do not have to match.

  ```
  INSERT INTO CourseIds (SELECT DISTINCT cid FROM enrolled);
  ```

6 Output Control

Since results SQL are unordered, you have to use the `ORDER BY` clause to impose a sort on tuples:

```
SELECT sid FROM enrolled WHERE cid = '15-721'
 ORDER BY grade DESC;
```

You can use multiple `ORDER BY` clauses to break ties or do more complex sorting:

```
SELECT sid FROM enrolled WHERE cid = '15-721'
 ORDER BY grade DESC, sid ASC;
```

You can also use any arbitrary expression in the `ORDER BY` clause:

```
SELECT sid FROM enrolled WHERE cid = '15-721'
 ORDER BY UPPER(grade) DESC, sid + 1 ASC;
```

By default, the DBMS will return all of the tuples produced by the query. You can use the `LIMIT` clause to restrict the number of result tuples:

```
SELECT sid, name FROM student WHERE login LIKE '%@cs'
 LIMIT 10;
```

Can also provide an offset to return a range in the results:

```
SELECT sid, name FROM student WHERE login LIKE '%@cs'
 LIMIT 10 OFFSET 20;
```

Unless you use an ORDER BY clause with a LIMIT, the DBMS could produce different tuples in the result on each invocation of the query because the relational model does not impose an ordering.

7 Nested Queries

Invoke queries inside of other queries to execute more complex logic within a single query. The scope of outer query is included in inner query (i.e. inner query can access attributes from outer query), but not the other way around.

Inner queries can appear in almost any part of a query:

1. SELECT Output Targets:

```
SELECT (SELECT 1) AS one FROM student;
```

2. FROM Clause:

```
SELECT name
  FROM student AS s, (SELECT sid FROM enrolled) AS e
 WHERE s.sid = e.sid;
```

3. WHERE Clause:

```
SELECT name FROM student
 WHERE sid IN ( SELECT sid FROM enrolled );
```

Example: Get the names of students that are enrolled in '15-445'.

```
SELECT name FROM student
 WHERE sid IN ( SELECT sid FROM enrolled WHERE cid = '15-445' );
```

Note that sid has different scope depending on where it appears in the query.

Nest Query Results Expressions:

- ALL: Must satisfy expression for all rows in sub-query.
- ANY: Must satisfy expression for at least one row in sub-query.
- IN: Equivalent to =ANY().
- EXISTS: At least one row is returned.

8 Window Functions

Performs "moving" calculation across set of tuples. Like an aggregation but it still returns the original tuples.

Functions: The window function can be any of the aggregation functions that we discussed above. There are also also special window functions:

1. ROW_NUMBER: The number of the current row.
2. RANK: The order position of the current row.

Grouping: The OVER clause specifies how to group together tuples when computing the window function. Use PARTITION BY to specify group.

```
SELECT cid, sid, ROW_NUMBER() OVER (PARTITION BY cid)
  FROM enrolled ORDER BY cid;
```

You can also put an ORDER BY within OVER to ensure a deterministic ordering of results even if database changes internally.

```
SELECT *, ROW_NUMBER() OVER (ORDER BY cid)
  FROM enrolled ORDER BY cid;
```

Important: The DBMS computes RANK after the window function sorting, whereas it computes ROW_NUMBER before the sorting.

9 Common Table Expressions

Common Table Expressions (CTEs) are an alternative to windows or nested queries to writing more complex queries. One can think of a CTE like a temporary table for just one query.

The WITH clause binds the output of the inner query to a temporary result with that name.
Example: Generate a CTE called cteName that contains a single tuple with a single attribute set to "1". The query at the bottom then just returns all the attributes from cteName.

```
WITH cteName AS (
    SELECT 1
)
SELECT * FROM cteName;
```

You can bind output columns to names before the AS:

```
WITH cteName (col1, col2) AS (
    SELECT 1, 2
)
SELECT col1 + col2 FROM cteName;
```

A single query can contain multiple CTE declarations:

```
WITH cte1 (col1) AS (
    SELECT 1
),
cte2 (col2) AS (
    SELECT 2
)
SELECT * FROM cte1, cte2;
```

Adding the RECURSIVE keyword after WITH allows a CTE to reference itself.
Example: Print the sequence of numbers from 1 to 10.

```
WITH RECURSIVE cteSource (counter) AS (
    (SELECT 1)
    UNION
    (SELECT counter + 1 FROM cteSource
    WHERE counter < 10)
)
SELECT * FROM cteSource;
```