Lecture #02: Modern SQL

15-445/645 Database Systems (Spring 2026)
https://15445.courses.cs.cmu.edu/fall2026/
Carnegie Mellon University
Andy Pavlo

1 SQL History

SQL is a declarative query language for relational databases. As oposed to imperative languages, in a declarative language the programmer/user only declares what needs to be done as oposed to how the operations should be done (e.g. Join these two tables). SQL 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).

Despite SQL being an old language, it is still being actively updated with new features every couple of years. Some of the major updates released with each new edition of the SQL standard are shown below:

- **SQL:1999** Regular Expressions, Triggers, Object-Oriented features
- **SQL:2003** XML, Windows, Sequences, Auto-Gen IDs
- **SQL:2008** Truncation, Fancy Sorting
- **SQL:2011** Temporal DBs, Pipelined DML
- **SQL:2016** JSON, Polymorphic tables
- **SQL:2023** Property Graph Queries, Multi-Dimensional Arrays

The minimum language syntax a system needs to support in order to claim that it supports SQL is SQL-92. Each vendor follows the standard to a certain degree and there are many proprietary extensions.

2 Relational Languages

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 and access controls.
4. It also includes view definition, integrity and referential constraints, and transactions.

Relational algebra (which is the algebra that SQL is based on) uses **sets** (unordered collections which do not allow duplicates). However, SQL is based on **bags** (unordered collections which allow duplicates) to avoid the extra work of removing duplicates by default. Duplicates can still be removed via features like the `DISTINCT` keyword.

3 Example Database

Here is the schema of a database we will use in our examples:

```
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

4 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 (almost) only be used in a `SELECT` output list.

- `AVG(COL)`: The average of the values in `COL`
- `MIN(COL)`: The minimum value in `COL`
- `MAX(COL)`: The maximum value in `COL`
- `SUM(COL)`: The sum of the values in `COL`
- `COUNT(COL)`: The number of tuples in the relation

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';
```

Some aggregate functions (e.g. `COUNT`, `SUM`, `AVG`) support the `DISTINCT` keyword:

Example: *Get # of unique students and their average GPA with a '@cs' login.*

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

A single SELECT statement can contain multiple aggregates:

Example: *Get # of students and their average GPA with a '@cs' login.*

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

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

Example: *Get the average GPA of students in each course.*

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

Most real-world database systems will error in this case, but some systems such as SQLite will allow it by picking an arbitrary value. The SQL:2023 standard now supports the ANY_VALUE aggregation function which does the same thing.

Example: *Get the average GPA of students in each course.*

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

Non-aggregated values in SELECT output clause must appear in the GROUP BY clause. This will partition the tuples based off of the value and calculate the aggregates for each subset. In this case there will be a canonical value for each group.

Example: *Get the average GPA of students in each course.*

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

Grouping sets can be used to specify multiple groupings in a single query rather than using UNION on multiple queries. This results in the DBMS needing to only scan through the data once rather than multiple times.

Example: *Get the count of students by each course and grade, the count of students by course, and the total student count.*

```
SELECT c.name AS c_name, e.grade,
       COUNT(*) AS num_students
  FROM enrolled AS e
  JOIN course AS c ON e.cid = c.cid
  GROUP BY GROUPING SETS (
   (c.name, e.grade),
   (c.name),
   (),
);
```

The FILTER clause filters results *before* aggregation computation (i.e. filter out rows going into the aggregate function). This allows computing multiple aggregates on the same raw data but with different conditions. It is helpful for pivoting rows to columns, among other things.

Example: *Get the avg course grade of students enrolled in 15-445 and 15-721.*

```
SELECT
  AVG(s.gpa) FILTER(WHERE e.cid = '15-445') AS intro_db_avg_gpa,
  AVG(s.gpa) FILTER(WHERE e.cid = '15-721') AS adv_db_avg_gpa
FROM enrolled AS e JOIN student AS s
ON e.sid = s.sid
```

The HAVING clause filters output results based on aggregation computation (i.e. filters out groups as oposed to filtering rows which is what the WHERE clause does). This makes HAVING behave like a WHERE clause for a GROUP BY.

Example: *Get the set of courses in which the average student GPA is greater than 3.9.*

```
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;
```

Note: this example uses a legacy implicit join syntax (here the DBMS can deduce that a join is required to handle the WHERE clause). You should always write out explicit joins in your queries.

The above query syntax is supported by many major database systems, but is not compliant with the SQL standard. To make the query standard compliant, we must repeat use of AVG(S.GPA) in the body of the HAVING clause.

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

5 String Operations

The SQL standard says that strings are **case sensitive** and **single-quotes only**. Real-world systems will vary in how loose they are about both points (e.g. MySQL).

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.

SIMILAR TO allows for regular expression matching but it is not supported across all systems as many have their own syntax instead.

String Functions SQL-92 defines string functions. Many database systems implement other functions in addition to those in the standard. Examples of standard string functions include SUBSTRING(S, B, E) and

`UPPER(S)`.

Concatenation: Two vertical bars ("||") will concatenate two or more strings together into a single string (but different systems might use a different symbol or implement it as a function).

6 Date and Time

Databases generally want to keep track of dates and time, so SQL supports operations to manipulate `DATE` and `TIME` attributes. These can be used as either outputs or predicates.

Specific syntax for date and time operations can vary wildly across systems.

7 Output Control

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

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

The default sort order is ascending (`ASC`). We can manually specify `DESC` to reverse the order:

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

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

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

We 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. Many systems provide their own syntax for specifying how to get a set number of the first results from the output, but a common one is the `LIMIT` clause:

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

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

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

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

SQL also allows you to store query results into a different table with the `INTO` keyword (some systems even allow redirection into a temporary table with `INTO TEMPORARY`).

8 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);
```

- **Temporary Table:** Store the output of the query into a temporary table created during the insertion. This table can then be used until the client disconnects.

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

9 Nested Queries

Nested queries invoke queries inside of other queries to execute more complex logic within a single query. Nested queries are often difficult to optimize.

The scope of the outer query is included in an inner query (i.e. the inner query can access attributes from outer query). The opposite is not true.

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 a different scope depending on where it appears in the query.

Example: *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT student.sid, name
  FROM student
  JOIN (SELECT MAX(sid) AS sid
          FROM enrolled) AS max_e
    ON student.sid = max_e.sid;
```

Nested 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.

Example: *Find all courses that have no students enrolled in it.*

```
SELECT * FROM course
  WHERE NOT EXISTS(
    SELECT * FROM enrolled
      WHERE course.cid = enrolled.cid
);
```

10 Lateral Joins

The `LATERAL` operator allows a nested query to reference attributes in other nested queries that precede it. You can think of lateral joins like a `for` loop that allows you to invoke another query for each tuple in a table.

Example: *Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order..*

Once we have gotten the `course` records, we can think of this query like below. For each course:

- Compute the number of enrolled students in this course
- Compute the average GPA of the enrolled students in this course

```
SELECT * FROM course AS c
  LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
    WHERE enrolled.cid = c.cid) AS t1,
  LATERAL (SELECT AVG(gpa) AS avg FROM student AS s
    JOIN enrolled AS e ON s.sid = e.sid
    WHERE e.cid = c.cid) AS t2;
```

11 Common Table Expressions

Common Table Expressions (CTEs) are an alternative to windows or nested queries when writing more complex queries. They provide a way to write auxiliary statements for use in a larger query. A CTE can be thought of as a temporary table that is scoped to a single query.

The `WITH` clause binds the output of the inner query to a temporary table with the same name.

Example: *Generate a CTE called* `cteName` *that contains a single tuple with a single attribute set to "1". Select all attributes from* `cteName`.

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

We 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 may 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. This enables the implementation of recursion in SQL queries. With recursive CTEs, SQL is provably Turing-complete, implying that it is as computationally expressive as more general purpose programming languages (ignoring the fact that it is a bit more cumbersome).

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;
```

12 Window Functions

A window function performs "sliding" calculation across a set of tuples that are related. Window functions are similar to aggregations, but tuples are not collapsed into a singular output tuple.

The conceptual execution for window functions can be imagined as such (*note that not all window functions will behave like this*):

1. The table is partitioned
2. Each partition is sorted (if there is an `ORDER BY` clause)
3. For each record, it creates a window spanning multiple records
4. Finally the output is computed for each window

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;
```

We 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.

Example: *Find the student with the second highest grade for each course.*

```
SELECT * FROM (
  SELECT *, RANK() OVER (PARTITION BY cid
           ORDER BY grade ASC) AS rank
    FROM enrolled) AS ranking
WHERE ranking.rank = 2;
```

Note that we order by ASC *because the grades are* A, B, C *instead of number grades.*