Advanced Database Systems

Spring 2026

Lecture #02:

SQL

SQL History

Developed @ IBM Research in the 1970s

- **System R** project
- Originally “SEQUEL”: Structured English Query Language

Commercialised/popularised in the 1980s

- Adopted by Oracle in the late 1970s
- IBM released DB2 in 1983

ANSI standard in 1986. ISO in 1987

- Structured Query Language
- Current standard is **SQL:2023**

SQL's Persistence

52 years old!

1974 – Developed at IBM by Donald D. Chamberlin and Raymond F. Boyce

Questioned repeatedly

90's: Object-Oriented DBMS (OQL, etc.)

2000's: XML (Xquery, Xpath, XSLT)

2010's: NoSQL & MapReduce

SQL keeps re-emerging as the standard

Even Hadoop, Spark etc. mostly used via SQL

May not be perfect, but it is useful

SQL Pros and Cons

Declarative!

- Say what you want, not how to get it

Implemented widely

- With varying levels of efficiency, completeness
- Most DBMSs support at least **SQL-92**

Constrained

- Not targeted at Turing-complete tasks

Feature-rich

- Many years of added features
- Extensible: callouts to other languages, data sources

Outline

Relational Terminology

Single-table Queries

Aggregations + Group By

Joins

Nested Queries

Relational Terminology

Database: Set of named relations

Relation (Table):

- **Schema:** description ("metadata") `Student(sid: int, name: text, dept: text)`
- **Instance:** collection of data satisfying the schema

sid	name	dept
12344	Jones	CS
12355	Smith	Physics
12366	Gold	CS

Tuple (record, row)

Attribute (field, column)

Relational Tables

Schema is fixed

Unique attribute names, attribute types are **atomic**

`Student(sid: int, name: text, dept: text)`

Instances can change often

In SQL, an instance is a **multiset** (bag) of tuples

name	dept	age
Jones	CS	18
Smith	Physics	21
Jones	CS	18

SQL Language

Three sublanguages

DDL	Data Definition Language	*Define and modify schema*
DML	Data Manipulation Language	*Write queries intuitively*
DCL	Data Control Language	*Control access to data*

RDBMS responsible for efficient evaluation

- Choose and run algorithms for declarative queries
- Choice of algorithm must **not** affect query answer

Example Database

Student(<u>sid</u>, name, dept, age)

sid	name	dept	age
12344	Jones	CS	18
12355	Smith	Physics	23
12366	Gold	CS	21

Enrolled(<u>sid</u>, <u>cid</u>, grade)

sid	cid	grade
12344	INF-10080	65
12355	INF-11199	72
12355	INF-11122	61
12366	INF-10080	80
12344	INF-11199	53

Course(<u>cid</u>, name, year)

cid	name	year
INF-11199	Advanced Database Systems	2020
INF-10080	Introduction to Databases	2020
INF-11122	Foundations of Databases	2019
INF-11007	Data Mining and Exploration	2019

Basic Single-Table Queries

```
SELECT [DISTINCT] <column expression list>
  FROM <single table>
[WHERE <predicate>]
```

```
SELECT *
  FROM Student
 WHERE age = 18
```

Get all 18-year-old students

Simplest version is straightforward

- Produce all tuples in the table that match the predicate
- Output the expressions in the **SELECT** list
- Expression can be a column reference, or an arithmetic expression over column refs
- **DISTINCT** removes duplicate rows before output

```
SELECT DISTINCT cid
  FROM Enrolled
 WHERE grade > 95
```

Get IDs of courses with grades > 95

ORDER BY

`ORDER BY <column*> [ASC|DESC]`

Sort the output tuples by the values in one or more of their columns

```
SELECT sid, grade FROM Enrolled
 WHERE cid = 'INF-11199'
 ORDER BY grade
```

sid	grade
12344	53
12399	72
12355	72
12311	76

Ascending order by default, but can be overridden

- Can mix and match, lexicographically

```
SELECT sid, grade FROM Enrolled
 WHERE cid = 'INF-11199'
 ORDER BY grade DESC, sid ASC
```

sid	grade
12311	76
12355	72
12399	72
12344	53

LIMIT

LIMIT <count> [offset]

Limit the # of tuples returned in the output

```
SELECT sid, grade FROM Enrolled
 WHERE cid = 'INF-11199'
 ORDER BY grade LIMIT 3
```

sid	grade
12344	53
12399	72
12355	72

Typically used with **ORDER BY**

Otherwise the output is <u>**non-deterministic**</u>, depends on the algo for query processing

Can set an offset to skip first records

```
SELECT sid, grade FROM Enrolled
 WHERE cid = 'INF-11199'
 ORDER BY grade LIMIT 3 OFFSET 1
```

sid	grade
12399	72
12355	72
12311	76

AGGREGATES

Functions that return a summary (aggregate) of some arithmetic expression from a bag of tuples

Get the average age of CS students

```
SELECT AVG(age) AS avg_age
  FROM Student WHERE dept = 'CS'
```

avg_age
20.5

Get the average age and # of CS students

```
SELECT AVG(age) AS avg_age,
       COUNT(sid) AS cnt
  FROM Student WHERE dept = 'CS'
```

avg_age	cnt
20.5	153

Aggregate functions can only be used in the **SELECT** list

Other aggregates: **SUM**, **COUNT**, **MIN**, **MAX**

GROUP BY

Get the average age per department

```
SELECT dept, AVG(age) AS avg_age
  FROM Student
 GROUP BY dept
```

dept	avg_age
CS	20.5
Physics	21.1
Maths	19.8

Partition table into groups with the same **GROUP BY** column values

- Can group by a list of columns

Produce an aggregate result per group

- Cardinality of output = # of distinct group values

Can put grouping columns in the **SELECT** output list

Group By

Non-aggregated values in **SELECT** output clause must appear in **GROUP BY** clause

```
SELECT dept, name, AVG(age)
  FROM Student
 GROUP BY dept
```

✗

```
SELECT dept, name, AVG(age)
  FROM Student
 GROUP BY dept, name
```

✓

FILTER GROUPS

Get the average age per department

```
SELECT dept, AVG(age) AS avg_age
  FROM Student
 GROUP BY dept
```

dept	avg_age
CS	20.5
Physics	21.1
Maths	19.8

Get departments with average student age above 21

```
SELECT dept, AVG(age) AS avg_age
  FROM Student
 WHERE avg_age > 21
 GROUP BY dept
```

dept	avg_age
Physics	21.1

Having

Get departments with average student age above 21

```
SELECT dept, AVG(age) AS avg_age
  FROM Student
 GROUP BY dept
HAVING AVG(age) > 21
```

HAVING filters results **after** grouping and aggregation

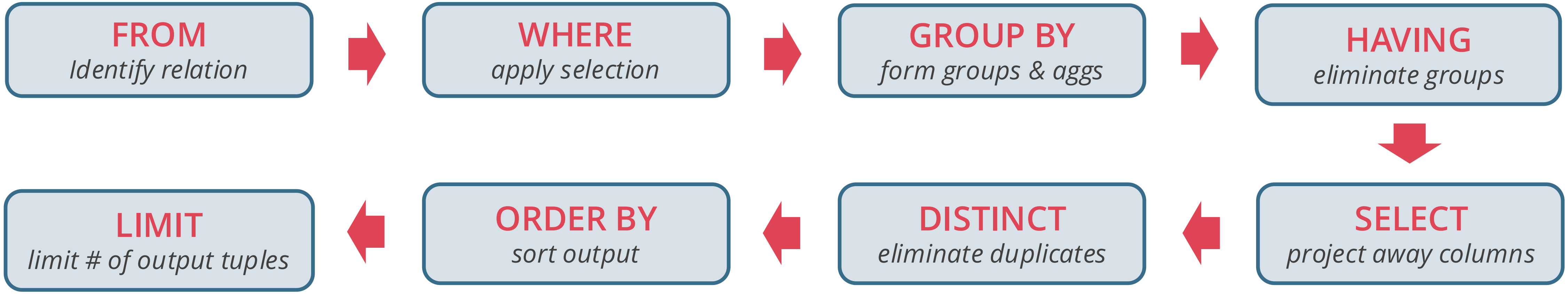
- Hence can contain anything that could go in the SELECT list
- I.e., GROUP BY columns or aggregates (e.g., `COUNT(*) > 5`)

HAVING can only be used in aggregate queries

- It's an optional clause

Conceptual SQL Evaluation

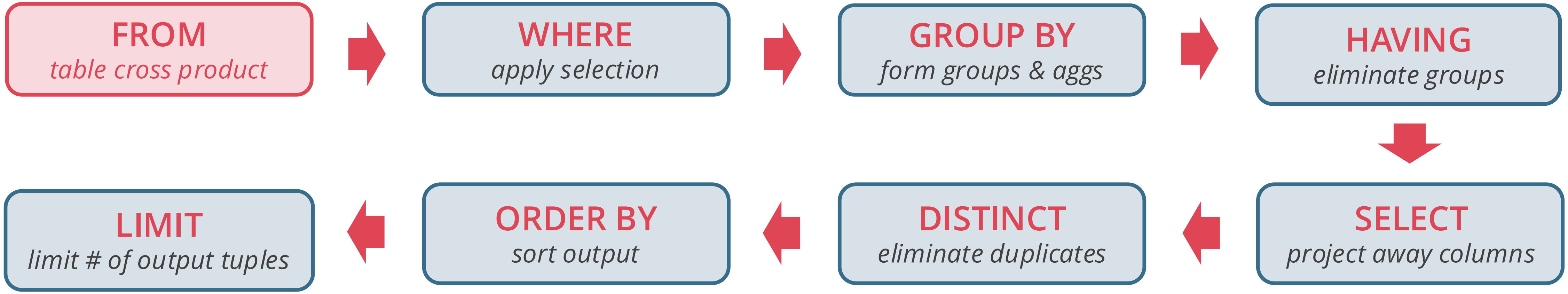
```
SELECT [DISTINCT] <column expression list>
  FROM <single table>
[WHERE <predicate>]
[GROUP BY <column list> [HAVING <predicate>]]
[ORDER BY <column list>] [LIMIT <count>]
```



Does not imply the query will actually be evaluated this way!

MULTIPLE-TABLE QUERIES

```
SELECT [DISTINCT] <column expression list>
  FROM <table1 [AS t1], ..., tableN [AS tn]>
[WHERE <predicate>]
[GROUP BY <column list> [HAVING <predicate>]]
[ORDER BY <column list>] [LIMIT <count>]
```



This evaluation strategy is almost always inefficient!

Join Query

Get the names and grades of students in INF-11199

```
SELECT S.name, E.grade
  FROM Student AS S, Enrolled AS E
 WHERE S.sid = E.sid
   AND E.cid = 'INF-11199'
```

name	grade
Smith	72
Jones	53

Student(sid, name, dept, age)

sid	name	dept	age
12344	Jones	CS	18
12355	Smith	Physics	23
12366	Gold	CS	21

Enrolled(sid, cid, grade)

sid	cid	grade
12344	INF-10080	65
12355	INF-11199	72
12355	INF-11122	61
12366	INF-10080	80
12344	INF-11199	53

Declarative computation

- Let the DBMS figure out how to compute this query
- Possible options:
  1) Cross product → filter on **sid** & **cid** → projection
  2) Filter on **cid** → cross product → filter on **sid** → projection
  3) Something else?

Join Query – Another Syntax

Get the names and grades of students in INF-11199

```
SELECT S.name, E.grade
  FROM Student AS S, Enrolled AS E
 WHERE S.sid = E.sid
   AND E.cid = 'INF-11199'
```

All 3 queries are equivalent

```
SELECT S.name, E.grade
  FROM Student S INNER JOIN Enrolled E
    ON S.sid = E.sid
 WHERE E.cid = 'INF-11199'
```

Inner join what we've learned so far

INNER is optional here

```
SELECT S.name, E.grade
  FROM Student S NATURAL JOIN Enrolled E
 WHERE E.cid = 'INF-11199'
```

NATURAL means equi-join for pairs of attributes with the same name

Join Variants

```
SELECT <column list>
FROM <table>
  [INNER | NATURAL | { LEFT | RIGHT | FULL } OUTER] JOIN
  ON <qualification list>
WHERE ...
```

The different types of **outer** joins determine what we do with rows that don’t match the join condition

Left Outer Join

Student

sid	name	dept	age
121	Jones	CS	18
122	Smith	Physics	19
123	Gold	CS	21

Enrolled

sid	cid	grade
121	INF-10080	65
123	INF-11199	72
121	INF-11122	61
201	INF-11199	53

```
SELECT S.name, E.grade
  FROM Student S LEFT OUTER JOIN Enrolled E
    ON S.sid = E.sid
```

name	grade
Jones	65
Jones	61
Gold	72
Smith	NULL

Return all matched rows & **preserve all unmatched rows from the table on the left** of the join clause

Use NULLs in fields of non-matching tuples

Right Outer Join

Student

sid	name	dept	age
121	Jones	CS	18
122	Smith	Physics	19
123	Gold	CS	21

Enrolled

sid	cid	grade
121	INF-10080	65
123	INF-11199	72
121	INF-11122	61
201	INF-11199	53

```
SELECT S.name, E.grade
  FROM Student S RIGHT OUTER JOIN Enrolled E
    ON S.sid = E.sid
```

name	grade
Jones	65
Jones	61
Gold	72
NULL	53

Return all matched rows & **preserve all unmatched rows from the table on the right** of the join clause

Full Outer Join

Student

sid	name	dept	age
121	Jones	CS	18
122	Smith	Physics	19
123	Gold	CS	21

Enrolled

sid	cid	grade
121	INF-10080	65
123	INF-11199	72
121	INF-11122	61
201	INF-11199	53

```
SELECT S.name, E.grade
  FROM Student S FULL OUTER JOIN Enrolled E
    ON S.sid = E.sid
```

name	grade
Jones	65
Jones	61
Gold	72
Smith	NULL
NULL	53

Return all matched & unmatched rows from the tables on both sides of the join clause

NESTED QUERIES

Queries containing other queries

They are often difficult to optimise

Inner queries can appear (almost) anywhere in query

Get the names of students enrolled in any course

Outer Query →

```
SELECT S.name FROM Student S
WHERE S.sid IN
  ( SELECT E.sid FROM Enrolled E )
```

← Inner Query

NESTED QUERIES

Get the names of students in INF-11199

```
SELECT S.name FROM Student S
 WHERE S.sid IN (
   SELECT E.sid FROM Enrolled E
    WHERE E.cid = 'INF-11199'
 )
```

"S.sid in the set of students that take INF-11199"

This is a bit odd, but it is equivalent:

```
SELECT S.name FROM Student S
 WHERE EXISTS (
   SELECT E.sid FROM Enrolled E
    WHERE E.cid = 'INF-11199'
      AND S.sid = E.sid )
```

Nested query with correlation on **sid**

Correlated subquery is recomputed for each Student tuple

More on Set-Comparison Operators

Seen so far: IN, EXISTS

Can also have: NOT IN, NOT EXISTS, *op* ALL, *op* ANY

where *op* is a standard comparison operator (=, <>, !=, >, >=, <, <=)

ALL → Must satisfy expression for all rows in subquery

ANY → Must satisfy expression for at least one row in subquery

IN → Equivalent to '= ANY( )'

NOT IN → Equivalent to '!= ALL( )'

EXISTS → At least one row is returned

Get the names of students in INF-11199

```
SELECT S.name FROM Student S
 WHERE S.sid = ANY (
   SELECT E.sid FROM Enrolled E
    WHERE E.cid = 'INF-11199'
 )
```

SUMMARY

This was a crash course on SQL

- Many aspects not covered though, only essential

SQL is a declarative language

- Somebody must translate SQL to algorithms... but how?

The data structures and algorithms that make SQL possible also power:

- NoSQL, data mining, scalable ML analytics,...
- A toolbox for scalable computing!
- That fun begins next week