Lecture 03: Advanced SQL

15-445/645 Database Systems (Fall 2017)
Carnegie Mellon University
Prof. Andy Pavlo

Relational Languages

vs. imperative. procedural.
↳ "How to compute it"

- User only needs to specify what they want (Declarative language i.e. SQL)

  ↳ "What data we want to extract"

Because query is declarative.

- DBMS decides how to compute the answer
- **Query optimizer** figures out the best plan to get the answer → Usually Commercial db is highly optimized compared to open-sourced.
- Data manipulation language (DML): Inserts, updates, deletes etc
- Data definition language (DDL): How the database looks (i.e. schema)
- SQL is based on **bags (has duplicates) not sets (no duplicates)**

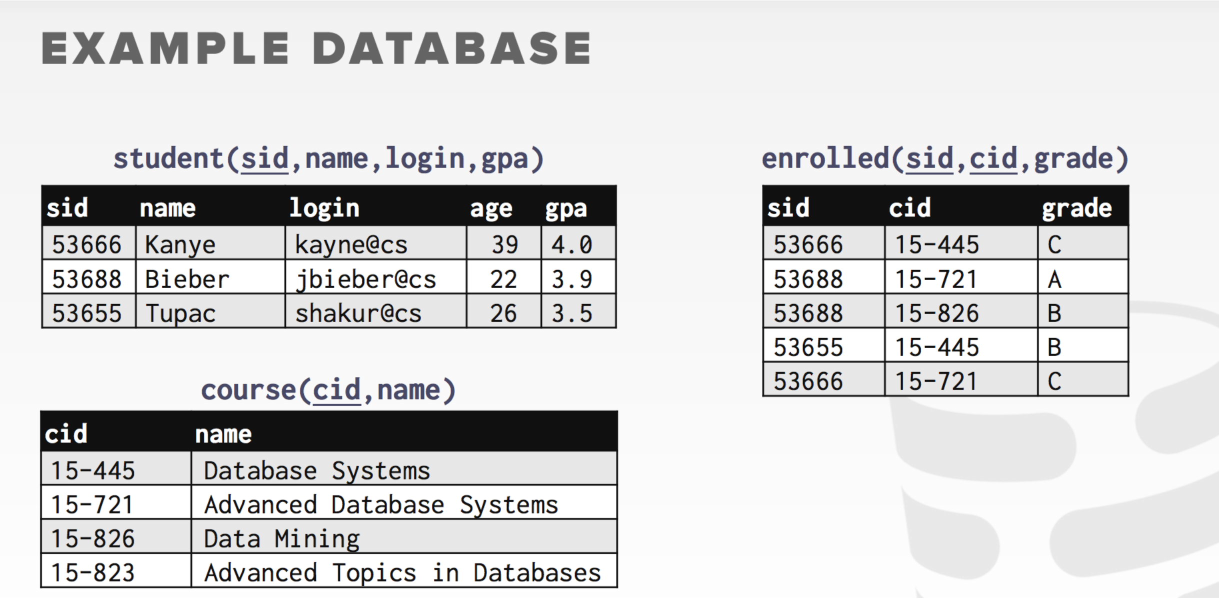
  ← Unordered collection which allows duplicates.
  Because removing duplicates are a very expensive operation.

	list	set	bag
order	O	X	X
duplicate	O	X	O

History

- Edgar Codd published major paper on relational models
- SQL : Structured Query Language
- Originally "SEQUEL" from IBM
- IBM was the biggest party in Databases, so SQL became the standard
- SQL-92 is the basic standard that needs to be supported
- Each vendor follows the standard to a certain degree

each vendor try to update the standard to support their own features.



EXAMPLE DATABASE

student(sid,name,login,gpa)

sid	name	login	age	gpa
53666	Kanye	kayne@cs	39	4.0
53688	Bieber	jbieber@cs	22	3.9
53655	Tupac	shakur@cs	26	3.5

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53655	15-445	B
53666	15-721	C

course(cid,name)

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-823	Advanced Topics in Databases

Example database used for lecture

Aggregates

```
AVG, MIN, MAX, SUM, COUNT
```

- Takes a bag of tuples => does computation => produces result
- Can only be used in SELECT output list
- "Get # of students with a "@cs" login (all these queries are equivalent) Same op, different ways.

Query optimizer try to make query as simple as possible.

```
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'
```

- Supports multiple aggregates

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

- Supports distinct: "COUNT(DISTINCT login)"

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

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

Cannot aggregate it. multiple e.cid → ?

```
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 be group byd

```
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;
```

cannot be put into WHERE.

String Operations

- Strings are **case sensitive and single quotes only** with some exceptions (it depends on implementations)
  - MySQL: Case insensitive and Single/double quotes
  - SQLite: Single/double quotes SQL: single quote only → better practice than double quotes.
- **LIKE** is used for string matching
  - ”%” matches any substrings (including substring)
  - ”_” matches any one character
- ”||” used to concatenate two or more strings together

Output redirection

- For when you want to store query results into another table and run followup queries

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

SQL-92

- Insert tuples from query into another table ← You can't run followup queries.
  - Inner SELECT must generate same columns as target table → CourseIds : one attribute which is int.

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

Date, Time operations

“Worst part.” Supports or Syntax varies widely among implementations.

Output control

- ORDER BY used to order tuples based on column ← can also use attributes that are not SELECTed

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

- Multiple ORDER BY's can be used to break ties

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

→ sort by grade first. use sid to break ties.

- LIMIT used to limit number of result tuples

```
LIMIT <count> [offset]
```

- Offset can be used to return a range

Nested Queries

- Often difficult to optimize, but very often the most intuitive way to read & write SQL.
- Inner queries can appear (almost) anywhere in query

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

inner query.

- Get names of students in 445

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

  - sid has different scope depending on query

- **ALL**: Must satisfy expression for all rows in subquery
- **ANY**: Must satisfy expression for atleast one row in subquery
- **IN**: Equivalent to =ANY()
- **EXISTS**: Atleast one row is returned

ex) WHERE sid >= ANY (SELECT sid ...)
= WHERE sid IN (SELECT MAX(sid)...)

- **Scope of outer query is included in inner query (i.e. inner query can access attributes from outer query)**
  - Not the other way around

Window Functions

- Performs calculation across set of tuples
- Allows you to group calculation into windows

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

ROW NUMBER is nondeterministic. it may change.

- Placing ORDER BY within OVER() makes result deterministic ordering of results even if database changes internally

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

PARTITION BY cid

- ~~RANK is done after you order, ROW_NUMBER before you order~~

Common Table Expressions (CTEs)

- Alternative to windows or nested queries
- Can create a temporary table for just one query

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

- You can bind output columns to names before the AS keyboard

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

- Allows for recursive CTE ↝ Useful to walk the graph or walk the tree. (Impossible without recursion: # of steps has to be predetermined if you can't use recursion)
  - Base case + UNION ALL + recursive use of CTE

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

Counter
0
1
⋮
9.

▷ SQL is type-safe. if it isn't doable, it's found in syntax checks. No need for runtime checks.

// But this also depends on implementations.

Conclusion

- SQL is not a dead language
- Strive to compute answers in one SQL query