LECTURE 4

SQL

Relational databases and various methods to query them

Databases

Brief Databases Overview

- A **database** is an organized collection of data.
- A **database management system (DBMS)** is a software system that **stores**, **manages**, and **facilitates access** to one or more databases.
- Why use DBMSes?
    - Our data might not be stored in a simple-to-read format such as a CSV (comma-separated values) file.
    - Think of a CSV like an Excel sheet or a sheet in Google sheets.
    - Up till now, most of the data were given to you in CSV files, but that will not always be the case in the real world.
    - If our data are stored in a DBMS, we must use languages such as Structured Query Language (SQL) to query for our data.

Advantages of DBMS over CSV (or similar)

Data Storage:

- **Reliable storage** to survive system crashes and disk failures.
- Optimize to compute on data that **does not fit in memory**.
- Special data structures to **improve performance**.

Data Management:

- Configure how data is **logically organized** and **who has access**.
- Can enforce guarantees on the data (e.g. non-negative bank account balance).
    - Can be used to **prevent data anomalies**.
    - Ensures **safe concurrent operations** on data.

Database Schemas

Relational DBMS Terminology

In a relational database, each table is called a **relation**.

Each row of relation is called a **record** or **tuple**. Rows do not have names.

Each column of a relation is called an **attribute** or **field**.

- Attributes have **names** (e.g. temperature, city, legs).
- Attributes have **data types** (e.g. INTEGER, CHAR(20)).
- Attributes may also have **constraints** (e.g. must be non-negative).
- Attributes may be marked as **primary** or **foreign keys**.
  - Primary key must be unique. Example on next slide.
  - Foreign key means that an attribute is some other table's primary key.
    - Explicitly shows how tables are linked.

Set of facts about the attributes is known as the "schema".

Example Relation Schema

```
CREATE TABLE animal(
      name TEXT,
      legs INTEGER CHECK (legs >= 0)
      weight INTEGER CHECK (weight >= 0),
      PRIMARY KEY(name));
```

Name	Legs	Weight
Dog	4	20
Cat	4	10
Ferret	4	10
T-Rex	2	12000
Penguin	2	10
Bird	2	6

Given the table animal above, it is impossible to:

- Insert a record with the same name as another.
- Insert a record with a negative value for legs or weight.
- Insert a record with a non-integer legs or weight.

Database Schema

A **relational database** is a set of relations.

- Set of the schemas of those relations is called the **database schema**.
- If the database schema includes foreign key relations, schema effectively includes a description how the tables refer to one another.

Example Database Schema

```
CREATE TABLE Sailors (
   sid INTEGER,
   sname CHAR(20),
   rating INTEGER,
   age REAL,
   PRIMARY KEY (sid));

CREATE TABLE Boats (
   bid INTEGER,
   bname CHAR (20),
   color CHAR(10),
   PRIMARY KEY (bid));

 CREATE TABLE Reserves (
   sid INTEGER,
   bid INTEGER,
   day DATE,
  PRIMARY KEY (sid, bid, day),
  FOREIGN KEY (sid) REFERENCES Sailors,
  FOREIGN KEY (bid) REFERENCES Boats);
```

Note: Primary key is all 3 attributes!

<u>**sid**</u>	sname	rating	age
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

<u>**bid**</u>	bname	color
101	Nina	red
102	Pinta	blue
103	Santa Maria	red

<u>**sid**</u>	<u>**bid**</u>	<u>**day**</u>
1	102	9/12
2	102	9/13

Database Implementations

We can query relational databases with SQL, but there are many implementations of SQL.

- And many other database implementations that are not SQL based / relational.

359 systems in ranking, August 2020

Rank			DBMS	Database Model	Score		
Aug 2020	Jul 2020	Aug 2019			Aug 2020	Jul 2020	Aug 2019
1.	1.	1.	Oracle	Relational, Multi-model	1355.16	+14.90	+15.68
2.	2.	2.	MySQL	Relational, Multi-model	1261.57	-6.93	+7.89
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1075.87	+16.15	-17.30
4.	4.	4.	PostgreSQL	Relational, Multi-model	536.77	+9.76	+55.43
5.	5.	5.	MongoDB	Document, Multi-model	443.56	+0.08	+38.99
6.	6.	6.	IBM Db2	Relational, Multi-model	162.45	-0.72	-10.50
7.	↑ 8.	↑ 8.	Redis	Key-value, Multi-model	152.87	+2.83	+8.79
8.	↓ 7.	↓ 7.	Elasticsearch	Search engine, Multi-model	152.32	+0.73	+3.23
9.	9.	↑ 11.	SQLite	Relational	126.82	-0.64	+4.10
10.	↑ 11.	↓ 9.	Microsoft Access	Relational	119.86	+3.32	-15.47
11.	↓ 10.	↓ 10.	Cassandra	Wide column	119.84	-1.25	-5.37

The 4 most popular SQL RDBMS implementations.

Lightweight SQL implementation that we'll use. Missing many features.

https://db-engines.com/en/ranking

SQL Overview

SQL Query Syntax

SELECT [DISTINCT] *<column expression list>*
FROM *<list of tables>*
[WHERE *<predicate>*]
[GROUP BY *<column list>*]
[HAVING *<predicate>*]
[ORDER BY *<column list>*]
[LIMIT *<number of rows>*];

DISTINCT

SELECT DISTINCT dept from students;

SELECT COUNT(DISTINCT dept) from students;

students

name	gpa	age	dept	gender
Sergey Brin	2.8	45	CS	M
Danah Boyd	3.9	40	CS	F
Bill Gates	1	63	CS	M
Hillary Mason	4	39	DATASCI	F
Mike Olson	3.7	53	CS	M
Mark Zuckerberg	3.8	34	CS	M
Sheryl Sandberg	3.6	49	BUSINESS	F
Susan Wojcicki	3.8	50	BUSINESS	F
Marissa Mayer	2.6	43	BUSINESS	F

DISTINCT

SELECT DISTINCT dept from students;
[Output: DATASCI, CS, BUSINESS]

SELECT COUNT(DISTINCT dept) from students;
[Output: 3]

students

name	gpa	age	dept	gender
Sergey Brin	2.8	45	CS	M
Danah Boyd	3.9	40	CS	F
Bill Gates	1	63	CS	M
Hillary Mason	4	39	DATASCI	F
Mike Olson	3.7	53	CS	M
Mark Zuckerberg	3.8	34	CS	M
Sheryl Sandberg	3.6	49	BUSINESS	F
Susan Wojcicki	3.8	50	BUSINESS	F
Marissa Mayer	2.6	43	BUSINESS	F

Types of Joins

Cross Join - Querying Multiple Relations

All pairs of rows appear in the result.

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

```
SELECT * FROM s, t;
```

Cross Join - Querying Multiple Relations

All pairs of rows appear in the result.

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

```
SELECT * FROM s, t;
```

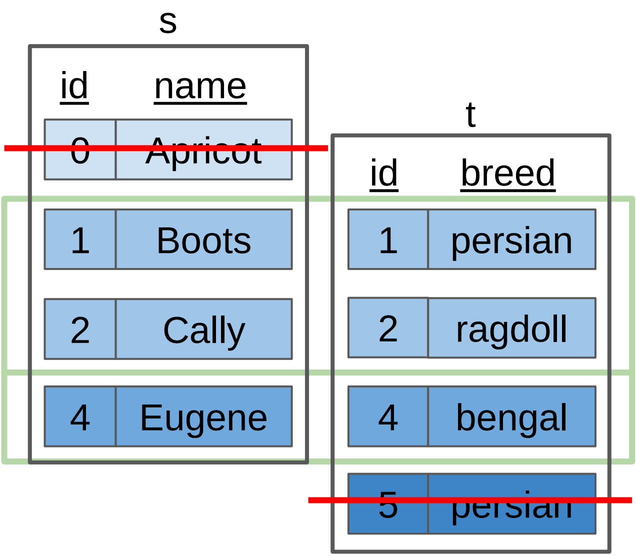
s.id	name	t.id	breed
0	Apricot	1	persian
1	Boots	1	persian
2	Cally	1	persian
4	Eugene	1	persian
0	Apricot	2	ragdoll
1	Boots	2	ragdoll
2	Cally	2	ragdoll
4	Eugene	2	ragdoll

(to be continued …)

(... continued)

s.id	name	t.id	breed
0	Apricot	4	bengal
1	Boots	4	bengal
2	Cally	4	bengal
4	Eugene	4	bengal
0	Apricot	5	persian
1	Boots	5	persian
2	Cally	5	persian
4	Eugene	5	persian

Inner Join



Only pairs of matching rows appear in the result.

```
SELECT * FROM s JOIN t ON s.id = t.id;
SELECT * FROM s INNER JOIN t ON s.id = t.id;
SELECT * FROM s, t WHERE s.id = t.id;
```

Inner Join

s

id	name
~~0~~	~~Apricot~~
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
~~5~~	~~persian~~

s.id	name	t.id	breed
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

Only pairs of matching rows appear in the result.

```
SELECT * FROM s JOIN t ON s.id = t.id;
SELECT * FROM s INNER JOIN t ON s.id = t.id;
SELECT * FROM s, t WHERE s.id = t.id;
```

Relationship Between Cross Joins and Inner Joins

s

id	name
~~0~~	~~Apricot~~
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
~~5~~	~~persian~~

s.id	name	t.id	breed
~~0~~	~~Apricot~~	~~1~~	~~persian~~
1	Boots	1	persian
~~2~~	~~Cally~~	~~1~~	~~persian~~
~~4~~	~~Eugene~~	~~1~~	~~persian~~
~~0~~	~~Apricot~~	~~2~~	~~ragdoll~~
~~1~~	~~Boots~~	~~2~~	~~ragdoll~~
2	Cally	2	ragdoll
~~4~~	~~Eugene~~	~~2~~	~~ragdoll~~

(to be continued …)

(... continued)

s.id	name	t.id	breed
~~0~~	~~Apricot~~	~~4~~	~~bengal~~
~~1~~	~~Boots~~	~~4~~	~~bengal~~
~~2~~	~~Cally~~	~~4~~	~~bengal~~
4	Eugene	4	bengal
~~0~~	~~Apricot~~	~~5~~	~~persian~~
~~1~~	~~Boots~~	~~5~~	~~persian~~
~~2~~	~~Cally~~	~~5~~	~~persian~~
~~4~~	~~Eugene~~	~~5~~	~~persian~~

Conceptually, an inner join is a cross join followed by removal of bad rows.

```
SELECT * FROM s, t WHERE s.id = t.id;
```

Relationship Between Cross Joins and Inner Joins

s

id	name
~~0~~	~~Apricot~~
1	Boots
2	Cally
4	Eugene

t

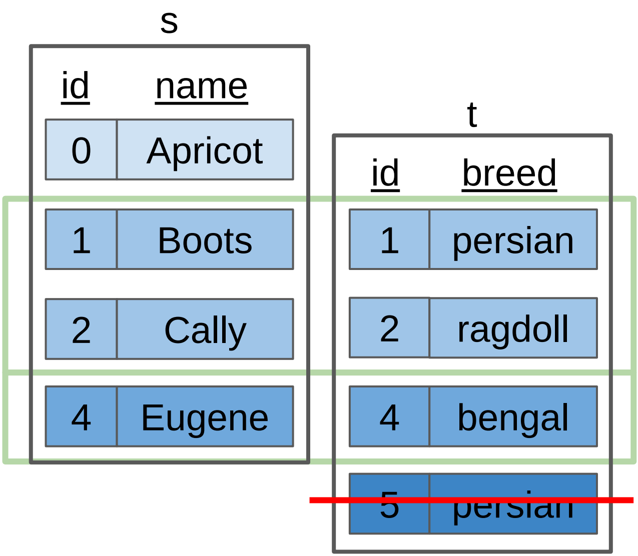
id	breed
1	persian
2	ragdoll
4	bengal
~~5~~	~~persian~~

s.id	name	t.id	breed
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

Conceptually, an inner join is a cross join followed by removal of bad rows.

```
SELECT * FROM s, t WHERE s.id = t.id;
```

Left Outer Join




```
SELECT * FROM s LEFT JOIN t ON s.id = t.id;
```

Every row in the first table appears in the result, matching or not.

Left Outer Join

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
~~5~~	~~persian~~

s.id	name	t.id	breed
0	Apricot		
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

Missing values are null.

```
SELECT * FROM s LEFT JOIN t ON s.id = t.id;
```

Every row in the first table appears in the result, matching or not.

Right Outer Join

s

id	name
~~0~~	~~Apricot~~
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

s.id	name	t.id	breed
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal
		5	persian

```
SELECT * FROM s RIGHT JOIN t ON s.id = t.id;
```

Note: SQLite does not implement RIGHT JOIN.

Every row in the second table appears in the result, matching or not.

Full Outer Join

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

s.id	name	t.id	breed
0	Apricot		
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal
		5	persian

```
SELECT * FROM s FULL OUTER JOIN t ON s.id = t.id;
```

Note: SQLite does not support FULL OUTER JOIN.

Every row in both tables appears, matching or not.

Other Join Conditions

student

age	name
29	Jameel
37	Jian
20	John
20	Emma

teacher

age	name
52	Ira
41	Husain
27	John
36	Anuja

We can join on conditions other than equality.

```
SELECT * FROM student, teacher WHERE student.age > teacher.age;
```

Other Join Conditions

student

age	name
29	Jameel
37	Jian
20	John
20	Emma

teacher

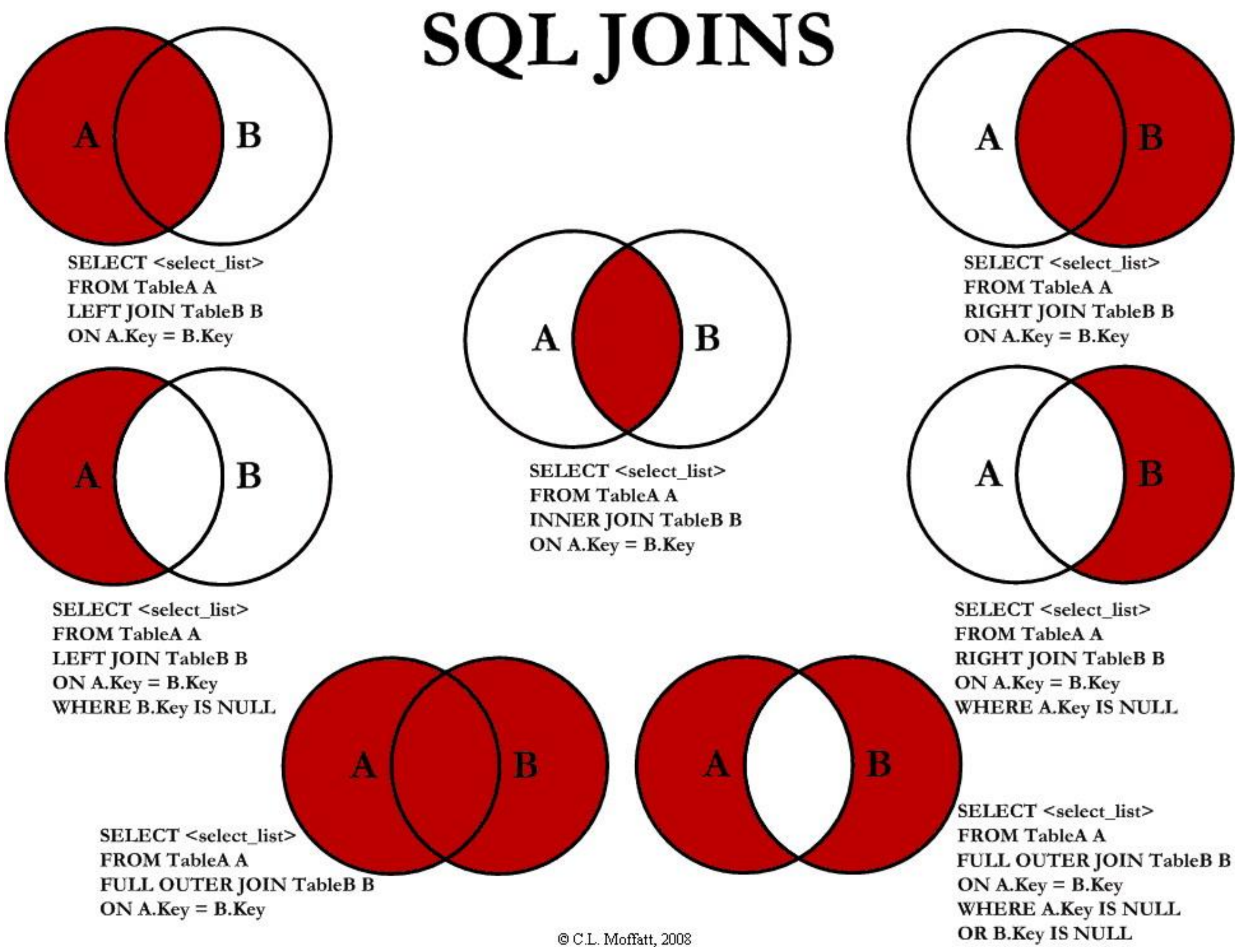
age	name
52	Ira
41	Husain
27	John
36	Anuja

29	Jameel	27	John
37	Jian	27	John
37	Jian	36	Anuja

We can join on conditions other than equality. Note that every satisfying pair appears.

- Inner joins are just cross joins followed by removing rows that don't match.

```
SELECT * FROM student, teacher WHERE student.age > teacher.age;
```



https://www.codeproject.com/KB/database/Visual_SQL_Joins/Visual_SQL_JOINS_orig.jpg

NULL Values

Brief Detour: NULL Values

- Field values are sometimes **unknown**
  - SQL provides a special value **NULL** for such situations
  - Every data type can be NULL
- The presence of null complicates many issues. E.g.:
  - Selection predicates (WHERE)
  - Aggregation
- But NULLs are common after outer joins

NULL in comparators

```
SELECT name = NULL FROM student;

SELECT name < NULL FROM student;

SELECT name >= NULL FROM student;
```

All of these queries evaluate to null!

Even this one!

```
SELECT * FROM student WHERE name = NULL;
```

Rule: (x *op* NULL) evaluates to … NULL!

Explicit NULL Checks

To check if a value is NULL you must use explicit NULL checks

```
SELECT * FROM student WHERE name IS NULL;

SELECT * FROM student WHERE name IS NOT NULL;
```

Aggregation with NULLs

Aggregates ignore NULL-valued inputs.

s.id	name	t.id	breed
0	Apricot		
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

```
SELECT COUNT(t.id) FROM s LEFT JOIN t ON s.id = t.id;
```

```
SELECT SUM(t.id) FROM s LEFT JOIN t ON s.id = t.id;
```

```
SELECT AVG(t.id) FROM s LEFT JOIN t ON s.id = t.id;
```

```
SELECT COUNT(*) FROM s LEFT JOIN t ON s.id = t.id;
```

Aggregation with NULLs

Aggregates ignore NULL-valued inputs.

s.id	name	t.id	breed
0	Apricot		
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

```
SELECT COUNT(t.id) FROM s LEFT JOIN t ON s.id = t.id; [Output: 3]

SELECT SUM(t.id) FROM s LEFT JOIN t ON s.id = t.id; [Output: 7]

SELECT AVG(t.id) FROM s LEFT JOIN t ON s.id = t.id; [Output: 7/3]

SELECT COUNT(*) FROM s LEFT JOIN t ON s.id = t.id; [Output: 4]
```

SQL Predicates and Casting

SQL Predicates

In addition to numerical comparisons (=, <, >), SQL has built-in predicates.

- Example: The `IN` operator tests whether a value is in a list.
    - E.g., select rows whose month is either January, March, or May:

```
SELECT * FROM t WHERE t.month IN ('January', 'March', 'May')
```

- Example: The LIKE operator tests whether a string matches a pattern (similar to a regex, but much simpler syntax):
    - E.g. select rows where the time string is on the hour, such as 8:00 or 12:00 pm.

```
SELECT * FROM t WHERE t.time LIKE ‘%:00%’;
```

SQL Casting

Can use CAST to convert fields from one type to another:

- Handy when combined with WHERE:

```
SELECT primaryTitle AS title,
  CAST(runtimeMinutes as int) AS time
FROM titles
WHERE time > 500
LIMIT 10;
```

SQL Sampling, Subqueries, and Common Table Expressions

Sampling with LIMIT?

```
SELECT * FROM students
LIMIT 5;
```

```
SELECT * FROM students
ORDER BY name LIMIT 5;
```

students

name	gpa	age	dept	gender
Sergey Brin	2.8	45	CS	M
Danah Boyd	3.9	40	CS	F
Bill Gates	1	63	CS	M
Hillary Mason	4	39	DATASCI	F
Mike Olson	3.7	53	CS	M
Mark Zuckerberg	3.8	34	CS	M
Sheryl Sandberg	3.6	49	BUSINESS	F
Susan Wojcicki	3.8	50	BUSINESS	F
Marissa Mayer	2.6	43	BUSINESS	F

Sampling with LIMIT?

```
SELECT * FROM students
LIMIT 5;
```

- Convenience sample?

```
SELECT * FROM students
ORDER BY name LIMIT 5;
```

- Probability sample
- Not a Simple Random Sample

students

name	gpa	age	dept	gender
Sergey Brin	2.8	45	CS	M
Danah Boyd	3.9	40	CS	F
Bill Gates	1	63	CS	M
Hillary Mason	4	39	DATASCI	F
Mike Olson	3.7	53	CS	M
Mark Zuckerberg	3.8	34	CS	M
Sheryl Sandberg	3.6	49	BUSINESS	F
Susan Wojcicki	3.8	50	BUSINESS	F
Marissa Mayer	2.6	43	BUSINESS	F

Random Sampling

The random sampling methods available depend on the database engine. Suppose we want to draw a SRS from an SQL table.

One common approach (with SQLite):

- `SELECT * FROM action_movie ORDER BY RANDOM() LIMIT 3`

May seem inefficient to order the entire table by some random values, then to only select 3.

- Query optimization under the hood will make this much more efficient.
- Reminder: SQL is a declarative language. You say “what”, not “how”.

Random Sampling

Suppose we want to pick 3 random years.

```
SELECT year FROM action_movie
    GROUP BY year
    ORDER BY RANDOM()
    LIMIT 3
```

Random Sampling

Suppose we want to get all movies from 3 randomly selected years.

```
SELECT * FROM action_movie
WHERE year IN (
SELECT year FROM action_movie
    GROUP BY year
    ORDER BY RANDOM()
    LIMIT 3
)
```

Effectively creates a temporary unnamed table that contains 3 randomly selected years.

Note: This is sometimes known as a “cluster sample.”

Random Sampling

Suppose we want to get all movies from 3 randomly selected years.

```
SELECT * FROM action_movie
WHERE year IN (
SELECT year FROM action_movie
    GROUP BY year
    ORDER BY RANDOM()
    LIMIT 3
)
```

Effectively creates a temporary unnamed table that contains 3 randomly selected years.

Known as a “subquery”.

Note: This is sometimes known as a “cluster sample.”

Subqueries

A query within another query can be used to create a temporary table.

- In a FROM clause: Describe a table instead of naming it.

E.g., join table u with a simple random sample from table t:

```
SELECT * FROM (SELECT * FROM t ORDER BY RANDOM() LIMIT 10), u;
```

- In a WHERE clause: Describe a one-column table instead of a list; used with IN.

E.g., select rows in a top-3 most popular month:

```
SELECT * FROM t WHERE t.month
   IN (SELECT month FROM months ORDER BY popularity DESC
       LIMIT 3);
```

Common Table Expressions

- A **Common Table Expressions (CTE)** allows for the creation of **“temporary tables”** to help organize complex queries.
  - CTEs can help make complex queries more readable.
  - CTEs can be used instead of subqueries.

```
WITH t2 AS (
        SELECT * FROM t ORDER BY RANDOM() LIMIT 10
)
SELECT * FROM t2, u;
```

SQL CASE Expressions and SUBSTR

CASE Expressions

- Without a base expression:

```
CASE WHEN born < 1980 THEN 'old'
     WHEN born < 2000 THEN 'not too old'
     ELSE 'young'
     END
```

A CASE expression chooses among alternative values.

base expression

- With a base expression:

```
CASE year % 10 WHEN 0 THEN 'start of decade'
               WHEN 5 THEN 'middle of decade'
               END
```

SUBSTR

SUBSTR allows you to extract substrings.

```
SELECT name, SUBSTR(knownForTitles, 1, INSTR(knownForTitles, ',')-1)
        AS most_popular_id
  FROM names
```

	name	knownForTitles
0	Fred Astaire	tt0050419,tt0043044,tt0053137,tt0072308
1	Lauren Bacall	tt0117057,tt0038355,tt0071877,tt0037382
...	...	...
8	Richard Burton	tt0087803,tt0061184,tt0057877,tt0059749
9	James Cagney	tt0042041,tt0035575,tt0031867,tt0029870

	name	most_popular_id
0	Fred Astaire	tt0050419
1	Lauren Bacall	tt0117057
...	...	...
8	Richard Burton	tt0087803
9	James Cagney	tt0042041

Conclusion

Summary

- SQL is a programming language designed for data queries
- SQL databases enable large-scale data processing
  - Databases are limited by disk size while Python is limited by memory size (disk size is usually much larger than memory size)
- Sampling procedures can be directly implemented in SQL
- Subqueries and common table expressions allow for the composition of complex queries