Lecture 2: Relational Model & Basic SQL

CREATING THE NEXT®

Today's Agenda

- Recap
- BuzzDB
- Relational Algebra
- Relational Language
- Aggregates
- Grouping

Administrivia

- Office hours
- Visual Code setup, ZSH shell
- Development Environment Setup Instructions

Recap

Complexity of Database Systems

Designing a robust, scalable algorithm is hard:

- must cope with very large instances
- hard even when the database fits in main memory
- billions of data items
- rules out the possibility of using $O(n^2)$ algorithms
- external algorithms (*i.e.*, database does not fit in memory) are even harder

This is why a DBMS is a complex software system.

Hardware Trends

This affects the design of a DBMS

- CPU costs are now more important
- I/O operations are eliminated or greatly reduced
- the classical architecture (**disk-oriented database systems**) has become suboptimal

But this is more of an evolution as opposed to a revolution. Many of the old techniques are still relevant for scalability.

C++: Tuple

```
#include <iostream>
#include <map>
#include <vector>

// A "class" in C++ is a user-defined data type.
// It is a blueprint for creating objects of a particular type,
// providing initial values for state (member variables or fields),
// and implementations of behavior (member functions or methods)
class Tuple {
public:
    int key;
    int value;
};
```

BuzzDB

BuzzDB

- BuzzDB – version 1
- BuzzDB – version 2
- BuzzDB – version 3

Machine Setup

- Instructions
- Operating System (OS): Ubuntu 22.04 (Linux Distribution)
- **Build System**: cmake
- Testing Library: Google Testing Library (gtest)
- **Continuous Integration (CI)** System: Gradescope
- Memory Error Detector: valgrind memcheck

C++: Tuple

```
#include <iostream>
#include <map>
#include <vector>

// A "class" in C++ is a user-defined data type.
// It is a blueprint for creating objects of a particular type,
// providing initial values for state (member variables or fields),
// and implementations of behavior (member functions or methods)
class Tuple {
public:
    int key;
    int value;
};
```

C++: Database

```
class BuzzDB {
private:
    // a map is an ordered key-value container
    std::map<int, std::vector<int>> index;

public:
    // a vector of Tuple structs acting as a table
    std::vector<Tuple> table;
    ...
};
```

C++: Loading into Database

```
BuzzDB db;

db.insert(1, 100);
db.insert(1, 200);
db.insert(2, 50);
db.insert(3, 200);
db.insert(3, 200);
db.insert(3, 100);
db.insert(4, 500);

db.selectGroupBySum();
```

C++: Inserting into Database

```
class BuzzDB {

public:
  // insert function
  void insert(int key, int value) {
      Tuple newTuple = {key, value};
      table.push_back(newTuple);
      index[key].push_back(value);
  }

};
```

C++: Aggregation Query

```
class BuzzDB {

public:
  // perform a SELECT ... GROUP BY ... SUM query
  void selectGroupBySum() {
      for (auto const& pair : index) { // for each unique key
          int sum = 0;
          for (auto const& value : pair.second) {
              sum += value; // sum all values for the key
          }
          std::cout << "key: " << pair.first << ", sum: " << sum << '\n';
      }
  }

};
```

C++: Loading Data From File

```
int main() {
    BuzzDB db;
    std::ifstream inputFile("output.txt");
    if (!inputFile) {
        std::cerr << "Unable to open file" << std::endl;
        return 1;
    }

    int field1, field2;
    while (inputFile >> field1 >> field2) {
        db.insert(field1, field2);
    }

    db.selectGroupBySum();
    return 0;
}
```

Relational Model: Definition

Relational Model

- **Structure:** The definition of relations and their contents.
- **Integrity:** Ensure the database's contents satisfy constraints.
- **Manipulation:** How to access and modify a database's contents.

Structure: Primary Key

- A relation's **primary key** uniquely identifies a single tuple.
- Some DBMSs automatically create an internal primary key if you don't define one.
- Auto-generation of unique integer primary keys (SEQUENCE in SQL:2003)

Schema: **Artists** (**ID**, **Artist**, **Year**, **City**)

ID	**Artist**	**Year**	**City**
1	1756	Salzburg	
2	1770	Bonn	
3	1810	Warsaw	

Structure: Foreign Key (1)

- A **foreign key** specifies that an tuple from one relation must map to a tuple in another relation.
- Mapping artists to albums?

Structure: Foreign Key (2)

Artists (ID, Artist, Year, City)
Albums (ID, Album, Artist_ID, Year)

Artists

ID	Artist	Year	City
1	Mozart	1756	Salzburg
2	Beethoven	1770	Bonn
3	Chopin	1810	Warsaw

Albums

ID	Album	Artist_ID	Year
1	The Marriage of Figaro	1	1786
2	Requiem Mass In D minor	1	1791
3	Für Elise	2	1867

Structure: Foreign Key (3)

What if an album is composed by two artists?

Structure: Foreign Key (3)

What if an album is composed by two artists?

Artists (ID, Artist, Year, City)
Albums (ID, Album, Year)
ArtistAlbum (Artist_ID, Album_ID)

ArtistAlbum

Artist_ID	Album_ID
1	1
2	1
2	2

Data Manipulation Languages

How to store and retrieve information from a database.

- **Relational Algebra**
  - The query specifies the (high-level) strategy the DBMS should use to find the desired result.
  - Procedural
- **Relational Calculus**
  - The query specifies only what data is wanted and not how to find it.
  - Non-Procedural

Relational Algebra

Core Operators

- These operators take in **relations** (*i.e.*, tables) as input and return a relation as output.
- We can "chain" operators together to create more complex operations.

- Selection (σ)
- Projection (Π)
- Union ($\cup$)
- Intersection ($\cap$)
- Difference ($-$)
- Product ($\times$)
- Join ($\bowtie$)

Core Operators: Selection

- Choose a subset of the tuples from a relation that satisfies a selection predicate.
- Predicate acts as a filter to retain only tuples that fulfill its qualifying requirement.
- Can combine multiple predicates using conjunctions / disjunctions.
- Syntax: $\sigma_{predicate}(\mathbf{R})$

```
SELECT * FROM R WHERE a_id = 'a2' AND b_id > 102;
```

R

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\sigma_{a\_id='a2' \wedge b\_id>102}(\mathbf{R})$:

a_id	b_id
a2	103

Georgia Tech

Core Operators: Projection

- Generate a relation with tuples that contains only the specified attributes.
- Can rearrange attributes' ordering.
- Can manipulate the values.
- Syntax: $\Pi_{A1,A2,...,An}(\mathbf{R})$

```
SELECT b_id - 100, a_id FROM R WHERE a_id = 'a2';
```

R

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\Pi_{b\_id-100,a\_id}(\sigma_{a\_id='a2'}(\mathbf{R}))$:

b_id - 100	a_id
2	a2
3	a2

Georgia Tech

Core Operators: Union

- Generate a relation that contains all tuples that appear in either only one or both input relations.
- Syntax: **R** ∪ **S**

```
(SELECT * FROM R)
  UNION ALL
(SELECT * FROM S)
```

R

a_id	**b_id**
a1	101
a2	102
a3	103

S

c_id	**d_id**
a2	102
a4	205

R ∪ S

a_id	**b_id**
a1	101
a2	102
a3	103
a2	102
a4	205

Semantics of Relational Operators

Set semantics: Duplicates tuples are **not** allowed
Bag semantics: Duplicates tuples are allowed

We will assume **bag (*a.k.a.*, multi-set)** semantics.

Core Operators: Intersection

- Generate a relation that contains only the tuples that appear in both of the input relations.
- Syntax: **R ∩ S**

```
(SELECT * FROM R)
  INTERSECT
(SELECT * FROM S)
```

R

a_id	b_id
a1	101
a2	102
a3	103

S

c_id	d_id
a2	102
a4	205

R ∩ S

a_id	b_id
a2	102

Core Operators: Difference

- Generate a relation that contains only the tuples that appear in the first and not the second of the input relations.
- Syntax: **R – S**

```
(SELECT * FROM R)
   EXCEPT
(SELECT * FROM S)
```

R

a_id	b_id
a1	101
a2	102
a3	103

S

c_id	d_id
a2	102
a4	205

R – S

a_id	b_id
a1	101
a3	103

Core Operators: Product

- Generate a relation that contains all possible combinations of tuples from the input relations.
- Syntax: $\mathbf{R} \times \mathbf{S}$

```
SELECT * FROM R CROSS JOIN S
```

R

a_id	b_id
a1	101
a2	102
a3	103

S

c_id	d_id
a2	102
a4	205

$\mathbf{R} \times \mathbf{S}$

a_id	b_id	c_id	d_id
a1	101	a2	102
a1	101	a4	205
a2	102	a2	102
a2	102	a4	205
a3	103	a2	102
a3	103	a4	205

Core Operators: Join

- Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.
- Syntax: **R** ⋈ **S**

```
SELECT * FROM R, S
WHERE R.a_id = S.c_id
```

R

a_id	**b_id**
a1	101
a2	102
a3	103

S

c_id	**d_id**
a2	102
a4	205

R ⋈ **S**

a_id	**b_id**	**c_id**	**d_id**
a2	102	a2	102

Derived Operators

Additional (derived) operators are often useful:

- Rename (ρ)
- Assignment (R$\leftarrow$S)
- Duplicate Elimination (δ)
- Aggregation (γ)
- Sorting (τ)
- Division (R $\div$ S)

Observation

Relational algebra still defines the high-level steps of how to execute a query.

- $\sigma_{S.c\_id=102}(\mathbf{R} \bowtie_{a\_id=c\_id} \mathbf{S})$ versus
- $(\mathbf{R} \bowtie_{a\_id=c\_id} (\sigma_{c\_id=102}(\mathbf{S})))$

A better approach is to state the high-level answer that you want the DBMS to compute.

- Retrieve the joined tuples from **R** and **S** where $a\_id = c\_id$ and c_id equals 102.

Relational Model

The relational model is independent of any query language implementation.

However, SQL is the **de facto** standard.

Example: Get the Albums composed by Beethoven.

```
for line in file:
  record = parse(line)
  if "Beethoven" == record[1]:
    print record[0]
```

```
SELECT Year
FROM Artists
WHERE Artist = "Beethoven"
```

Relational Language

Relational Language

- User only needs to specify the answer that they want, not how to compute it.
- The DBMS is responsible for efficient evaluation of the query.
  - Query optimizer: re-orders operations and generates query plan

SQL History

- Originally "SEQUEL" from IBM's **System R** prototype.
  - Structured English Query Language
  - Adopted by Oracle in the 1970s.
  - IBM releases DB2 in 1983.
  - ANSI Standard in 1986. ISO in 1987
  - Structured Query Language

SQL History

- Current standard is SQL:2016
  - SQL:2016 $\longrightarrow$ JSON, Polymorphic tables
  - SQL:2011 $\longrightarrow$ Temporal DBs, Pipelined DML
  - SQL:2008 $\longrightarrow$ TRUNCATE, Fancy sorting
  - SQL:2003 $\longrightarrow$ XML, windows, sequences, auto-gen IDs.
  - SQL:1999 $\longrightarrow$ Regex, triggers, OO
- Most DBMSs at least support SQL-92
- Comparison of different SQL implementations

Relational Language

- Data Manipulation Language (**DML**)
- Data Definition Language (**DDL**)
- Data Control Language (DCL)
- Also includes:
  - View definition
  - Integrity & Referential Constraints
  - Transactions
- Important: SQL is based on bag semantics (duplicates) not set semantics (no duplicates).

List of SQL Features

- Aggregations + Group By
- String / Date / Time Operations
- Output Control + Redirection
- Nested Queries
- Join
- Common Table Expressions
- Window Functions

Example Database

students

sid	name	login	age
1	Maria	maria@cs	19
2	Rahul	rahul@cs	22
3	Shiyi	shiyi@cs	26
4	Peter	peter@ece	35

enrolled

sid	cid	grade
1	1	3.5
1	2	4
2	3	3
4	2	2

courses

cid	name
1	Computer Architecture
2	Machine Learning
3	Database Systems
4	Programming Languages

Aggregates

Aggregates

- Functions that return a single value from a bag of tuples:
  - COUNT(col) $\longrightarrow$ Return number of values for col.
  - AVG(col) $\longrightarrow$ Return the average col value.
  - MIN(col) $\longrightarrow$ Return minimum col value.
  - MAX(col) $\longrightarrow$ Return maximum col value.
  - SUM(col) $\longrightarrow$ Return sum of values in col.

Aggregates

- Aggregate functions can only be used in the SELECT output list.
- **Task:** Get number of students with a "@cs" login:

```
SELECT COUNT(login) AS cnt
FROM students WHERE login LIKE '%@cs'
```

CNT
3

Multiple Aggregates

- **Task:** Get the average age and the the number of students and that have a "@cs" login.

```sql
SELECT AVG(age), COUNT(sid)
FROM students WHERE login LIKE '%@cs'
```

AVG	COUNT
23.33	3

Distinct Aggregates

- COUNT, SUM, AVG support DISTINCT
- **Task:** Get the number of unique students that have an "@cs" login.

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

COUNT
3

Aggregates

- Output of columns outside of an aggregate.
- **Task:** Get the average grade of students enrolled in each course.

```
SELECT e.cid, AVG(e.grade)
  FROM enrolled AS e;
```

AVG	e.cid
??	3

Aggregates

- Output of columns outside of an aggregate.
- **Task:** Get the average grade of students enrolled in each course.

```
SELECT e.cid, AVG(e.grade)
  FROM enrolled AS e;
```

AVG	e.cid
??	3

- column ”e.cid” must appear in the GROUP BY clause or be used in an aggregate function

Grouping

Group By

- Project tuples into subsets and calculate aggregates of each subset.
- **Task:** Get the average grade of students enrolled in each course.

```
SELECT e.cid, AVG(e.grade)
  FROM enrolled AS e
  GROUP BY e.cid;
```

e.cid	**AVG**
1	3.5
2	3
3	3

Having

- Filters results based on aggregate value.
- Predicate defined over a group (WHERE clause for a GROUP BY)
- **Task:** Get courses where is the average grade is >= 3.5.

```
SELECT e.cid, AVG(e.grade) AS avg_grade
  FROM enrolled AS e
  WHERE avg_grade >= 3.5
  GROUP BY e.cid;
```

Having

- Filters results based on aggregate value.
- Predicate defined over a group (WHERE clause for a GROUP BY)
- **Task:** Get courses where is the average grade is >= 3.5.

```
SELECT e.cid, AVG(e.grade) AS avg_grade
  FROM enrolled AS e
  GROUP BY e.cid
  HAVING avg_grade >= 3.5
```

e.cid	avg_grade
1	3.5

Conclusion

- Relational algebra defines the primitives for processing queries on a relational database.
- We will see relational algebra again when we talk about query execution.
- We covered basic SQL in this lecture
- In the next lecture, we will learn about advanced SQL.