

Lecture 2: Relational Model & Basic SQL

CREATING THE NEXT®

Today's Agenda

Recap

Relational Model: Motivation

Relational Model

Relational Algebra

Relational Language

Aggregates

Grouping

Administrivia

- Office hours
- Visual Code setup, ZSH shell

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.

External Sorting

- Sorting an arbitrary amount of data, stored on disk
- Accessing data on disk is slow – so we do not want to access each value individually
- Sorting in main memory is fast – but main memory size is limited

Solution

- Partition the list into a set of smaller-sized chunks that fit in main memory
- and sort all the chunks
- Use `std::sort` as the internal sorting algorithm.
- With **b** values fitting into main memory and **n** values that should be sorted:
- number of runs (**r**) = $\left\lceil \frac{n}{b} \right\rceil$ runs

Iterative K-Way Merge

- Iteratively merging the first K runs, the next K runs, and so on.
- As number of runs (r) is halved in each iteration, there are only $\Theta(\log_k r)$ iterations

Relational Model: Motivation

Digital Music Store Application

Consider an application that models a digital music store to keep track of artists and albums.

Things we need store:

- Information about Artists
- What Albums those Artists released

Flat File Strawman (1)

Store our database as comma-separated value (CSV) files that we manage in our own code.

- Use a separate file per entity
- The application has to parse the files each time they want to read/update records

Flat File Strawman (2)

Artists.csv

Artist	Year	City
Mozart	1756	Salzburg
Beethoven	1770	Bonn
Chopin	1810	Warsaw

Albums.csv

Album	Artist	Year
The Marriage of Figaro	Mozart	1786
Requiem Mass In D minor	Mozart	1791
Für Elise	Beethoven	1867

Flat File Strawman (3)

Example: Get the Albums composed by Beethoven.

for line in file:

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

	Album	Artist	Year
Albums.csv	The Marriage of Figaro	Mozart	1786
	Requiem Mass In D minor	Mozart	1791
	Für Elise	Beethoven	1867

Flat File Strawman (4)

Data Integrity

- How do we ensure that the artist is the same for each album entry?
- What if somebody overwrites the album year with an invalid string?
- How do we store that there are multiple artists on an album?

Implementation

- How do you find a sectionicular record?
- What if we now want to create a new application that uses the same database?
- What if two threads try to write to the same file at the same time?

Durability

- What if the machine crashes while our program is updating a record?
- What if we want to replicate the database on multiple machines for high availability?

Early DBMSs

Limitations of early DBMSs (*e.g.*, IBM IMS FastPath in 1966)

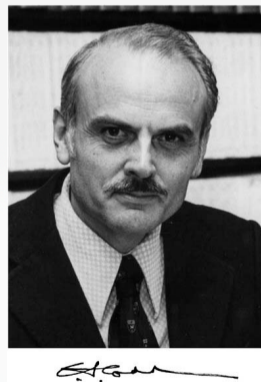
- Database applications were difficult to build and maintain.
- Tight coupling between logical and physical layers.
- You have to (roughly) know what queries your app would execute before you deployed the database.

Relational Model

Relational Model

Proposed in 1970 by Ted Codd (IBM Almaden).
Data model to avoid this maintenance.

- Store database in simple data structures
- Access data through high-level language
- Physical storage left up to implementation



Data Models

A **data model** is collection of concepts for describing the data in a database.

A **schema** is a description of a sectionicular collection of data, using a given data model.

List of data models

- Relational (SQL-based, most DBMSs, focus of this course)
- Non-Relational (*a.k.a.*, NoSQL) models
 - ▶ Key/Value, Graph, Document
 - ▶ Column-family
- Array/Matrix (Machine learning)
- Obsolete models
 - ▶ Hierarchical/Tree

Relation

A **relation** is an unordered **set** of **tuples**. Each tuple represents an entity.

A tuple is a set of **attribute** values.

Values are (normally) atomic/scalar.

Artist	Year	City
Mozart	1756	Salzburg
Beethoven	1770	Bonn
Chopin	1810	Warsaw

Jargon

- Relations are also referred to as tables.
- Tuples are also referred to as records or rows.
- Attributes are also referred to as columns.

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)

	<u>ID</u>	Artist	Year	City
Artists	1	Mozart	1756	Salzburg
	2	Beethoven	1770	Bonn
	3	Chopin	1810	Warsaw

	<u>ID</u>	Album	Artist_ID	Year
Albums	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)

	<u>Artist_ID</u>	<u>Album_ID</u>
ArtistAlbum	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

Core Operators: Projection

- Generate a relation with tuples that contains only the specified attributes.
- Can rearrange attributes' ordering.
- Can manipulate the values.
- Syntax: $\Pi_{A_1, A_2, \dots, A_n}(\mathbf{R})$

`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

Core Operators: Union

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

(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

$R \cup S$

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: $\mathbf{R} \cap \mathbf{S}$

```
(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

$\mathbf{R} \cap \mathbf{S}$

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	
<u>a_id</u>	<u>b_id</u>
a1	101
a2	102
a3	103

S	
<u>c_id</u>	<u>d_id</u>
a2	102
a4	205

$R - S$

R - S	
<u>a_id</u>	<u>b_id</u>
a1	101
a3	103

Core Operators: Product

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

`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

$R \times S$

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: $R \bowtie S$

```
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

$R \bowtie S$

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 \mathbf{R} and \mathbf{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 → JSON, Polymorphic tables
 - ▶ SQL:2011 → Temporal DBs, Pipelined DML
 - ▶ SQL:2008 → TRUNCATE, Fancy sorting
 - ▶ SQL:2003 → XML, windows, sequences, auto-gen IDs.
 - ▶ SQL:1999 → 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

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

	<u>sid</u>	<u>cid</u>	<u>grade</u>
enrolled	1	1	3.5
	1	2	4
	2	3	3
	4	2	2

	<u>cid</u>	<u>name</u>
courses	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) → Return number of values for col.
 - ▶ AVG(col) → Return the average col value.
 - ▶ MIN(col) → Return minimum col value.
 - ▶ MAX(col) → Return maximum col value.
 - ▶ SUM(col) → 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.

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

<u>AVG</u>	<u>COUNT</u>
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;
```

<u>AVG</u>	<u>e.cid</u>
------------	--------------

??	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;
```

<u>e.cid</u>	<u>AVG</u>
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  $\geq$  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 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  $\geq$  3.5
```

<u>e.cid</u>	<u>avg_grade</u>
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.