Lecture 2: Relational Model

CREATING THE NEXT®

Today's Agenda

Relational Model
1.1 Recap
1.2 External Sorting
1.3 Relational Model: Motivation
1.4 Relational Model
1.5 Relational Algebra

Recap

Motivational Example

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.

Problem Statement

- 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

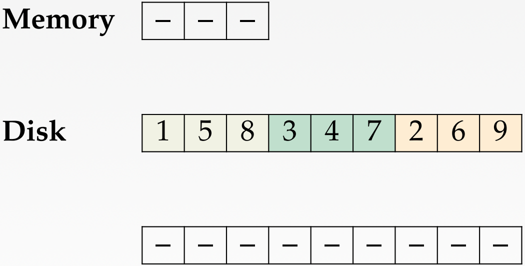
- Partition the list into a set of smaller-sized **chunks** that fit in main memory
- and sort all the **chunks**
- Use std::sort as the internal sorting algorithm.
- With **m** values fitting into main memory and **n** values that should be sorted:
- number of runs (**k**) = $\left\lceil \frac{n}{m}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

Iterative 2-Way Merge

- Iteratively merging the first run with the second, the third with the fourth, and so on.
- As number of runs (k) is halved in each iteration, there are only $\underline{\Theta(\log_2 k)\ \textbf{iterations}}$.
- In each iteration every element is moved exactly once.
- So in each iteration, we read and write out all the input data.
- The running time per iteration is therefore in $\Theta(n)$.
- The total I/O cost is therefore in $\Theta(n \log_2 k)$.

External Sorting

K-Way Merge (1)



Memory

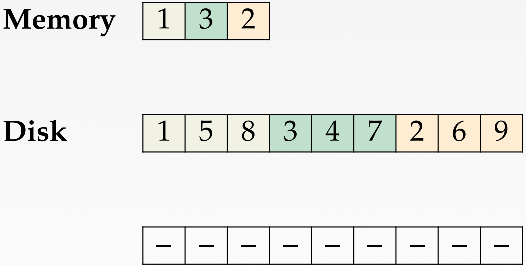
–	–	–

Disk

1	5	8	3	4	7	2	6	9

–	–	–	–	–	–	–	–	–

K-Way Merge (2)

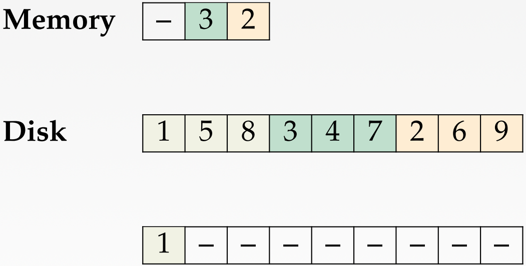


Memory 1 3 2

Disk 1 5 8 3 4 7 2 6 9

– – – – – – – – –

K-Way Merge (3)



Memory

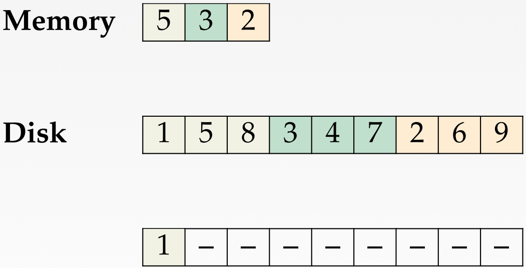
–	3	2

Disk

1	5	8	3	4	7	2	6	9

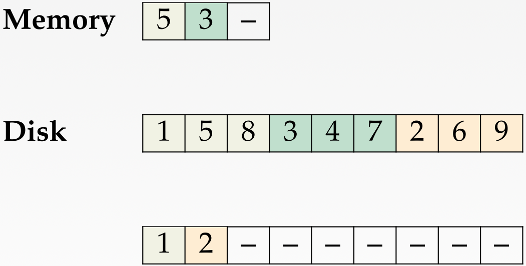
1	–	–	–	–	–	–	–	–

K-Way Merge (4)



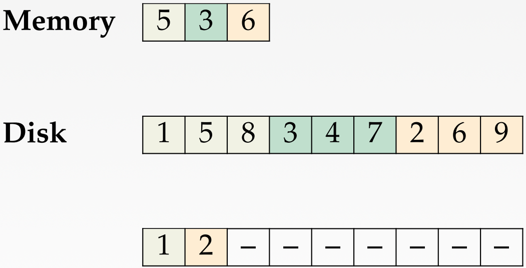
Memory	5	3	2						
Disk	1	5	8	3	4	7	2	6	9
	1	–	–	–	–	–	–	–	–

K-Way Merge (5)



Memory	5	3	–						
Disk	1	5	8	3	4	7	2	6	9
	1	2	–	–	–	–	–	–	–

K-Way Merge (6)



Memory

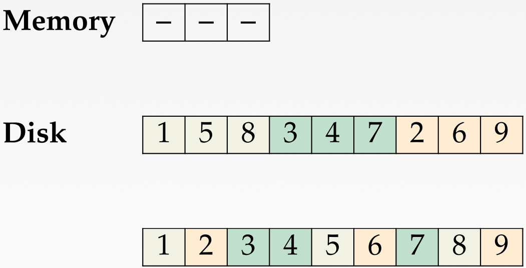
5	3	6

Disk

1	5	8	3	4	7	2	6	9

1	2	–	–	–	–	–	–	–

K-Way Merge (7)



Memory	–	–	–						
Disk	1	5	8	3	4	7	2	6	9
	1	2	3	4	5	6	7	8	9

K-Way Merge (8)

Fewer disk reads

- A straightforward implementation would scan all k runs to determine the minimum.
- This implementation results in a running time of Θ(kn).
- Although it would work, it is not efficient.

We can improve upon this by computing the smallest element faster.

- By using a heap, the smallest element can be determined in O(log k) time.
- Use std::priority_queue (implemented as a heap)
- The resulting running times are therefore in O(n log k).

K-way merge might not fit in memory

- Fall back to regular 2-way merge for a few 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]
```

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 (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 particular 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

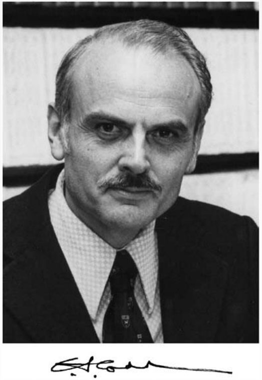
Limitations of early DBMSs (*e.g.*, IBM IMS FastPath in 1966)

- 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 particular 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)

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?
What if an artist composed two albums?

Structure: Foreign Key (3)

What if an album is composed by two artists?
What if an artist composed two albums?

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

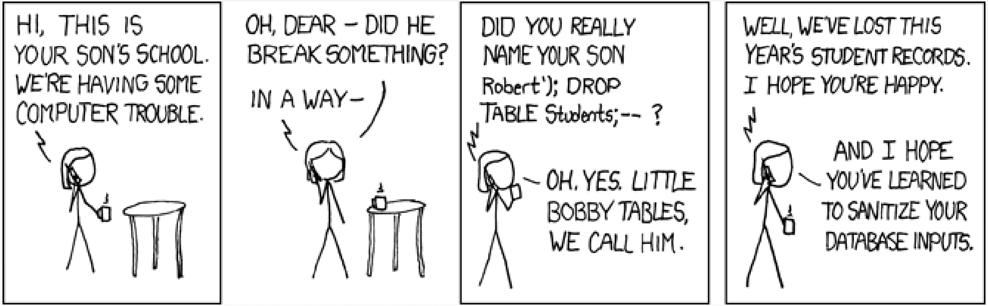
Relational Algebra

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 ($\bowtie$)

XKCD



Source: https://xkcd.com/327/

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

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

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: $\Pi_{A1,A2,...,An}(\mathbf{R})$

```
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	103
3	103

Core Operators: Union

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

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

R

a_id	b_id
a1	101
a2	102
a3	103

S

a_id	b_id
a3	103
a4	104
a5	105

$\mathbf{R} \cup \mathbf{S}$

a_id	b_id
a1	101
a2	102
a3	103
a3	103
a4	104
a5	105

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

a_id	b_id
a3	103
a4	104
a5	105

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

a_id	b_id
a3	103

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

a_id	b_id
a3	103
a4	104
a5	105

R – S

a_id	b_id
a1	101
a2	102

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

a_id	b_id
a3	103
a4	104
a5	105

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

R.a_id	R.b_id	S.a_id	S.b_id
a1	101	a3	103
a1	101	a4	104
a1	101	a5	105
a2	102	a3	103
a2	102	a4	104
a2	102	a5	105
a3	103	a3	103
a3	103	a4	104
a3	103	a5	105

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

```
SELECT * FROM R NATURAL JOIN S
```

R

a_id	b_id
a1	101
a2	102
a3	103

S

a_id	b_id
a3	103
a4	104
a5	105

$\mathbf{R} \bowtie \mathbf{S}$

a_id	b_id
a3	103

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.

- $\sigma_{\text{b¡d}=102}(\mathbf{R} \bowtie \mathbf{S})$ versus
- $(\mathbf{R} \bowtie \sigma_{\text{b¡d}=102}(\mathbf{S}))$

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 b_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"
```

Set-Oriented Processing

Small applications often loop over their data

- one for loop accesses all item x,
- for each item, another loop access item y,
- then both items are combined.

This kind of code of code feels “natural”, but is bad

- $\Omega(n^2)$ runtime
- does not scale

Instead: **set oriented** processing. Perform operations for large batches of data.

Set-Oriented Processing (2)

Processing whole batches of tuples is more efficient:

- can prepare index structures
- or re-organize the data
- sorting/hashing
- runtime ideally $O(n \log n)$

Many different algorithms, we will look at them later.

Conclusion

- External sorting allows us to sort larger-than-memory datasets
- Relational algebra defines the primitives for processing queries on a relational database.
- We will see relational algebra again when we talk about query execution.
- In the next lecture, we will learn about advanced SQL.