# Lecture #01: Relational Model & Algebra


**15-445/645 Database Systems (Fall 2024)**
https://15445.courses.cs.cmu.edu/fall2024/
Carnegie Mellon University
Andy Pavlo


## 1 Databases

A *database* is an organized collection of inter-related data that models some aspect of the real-world (e.g modeling the students in a class or a digital music store). Databases are the core component of most computer applications.

People often confuse "databases" with "database management systems" (e.g. MySQL, Oracle, MongoDB, Snowflake). A database management system (DBMS) is the software that manages a database.

## 2 Flat File Strawman

Suppose we have a simple database similar to a digital music store like Spotify. The database holds information about the artists and which albums those artists have released. This database has two entities: an Artists entity and an Albums entity.

Let's say the database's records are stored as comma-separated value (csv) files that the DBMS manages, and each entity is stored in its own file (an `artists.csv` and `albums.csv` file). Both the Artists entity and the Albums entity their own attributes. For each entity file, different records are delimited by new lines, while each of the corresponding attributes within a record are delimited by a comma. The application has to parse these files each time it wants to read or update any records.

In this example, artists each have a `name`, `year`, and `country` attribute. Albums have `name`, `artist`, and `year` attributes.

The following is an example CSV file for information about artists with the schema (name, year, country):

```
"Wu-Tang Clan", 1992, "USA"
"Notorious BIG", 1992, "USA"
"GZA", 1990, "USA"
```

### Issues with Flat Files

There are several problems with using flat files as our database. Here are some questions to consider:

- **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? What if there are multiple artists on an album? What happens if we delete an artist that has albums?
- **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 that application is running on a different machine? 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?

## 3 Database Management System

A **DBMS** is software that allows applications to store and analyze information in a database.

A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases in accordance with some *data model*.

A **data model** is a collection of concepts for describing the data in database. Some examples include:

- **Relational** (most common)
- **NoSQL** (key/value, document, graph)
- **Array / Matrix / Vector** (for machine learning)

A **schema** is a description of a particular collection of data based on a data model. This defines the structure of data for a data model. Without a schema, you will have random bits with no meaning.

### Common Data Models

- Relational (Most DBMSs)
- Key/Value (NoSQL)
- Graph (NoSQL)
- Document/XML/Object (NoSQL)
- Wide-Column/Column-family (NoSQL)
- Array / Matrix / Vector (Machine Learning)
- Hierarchical (Obsolete/Legacy/Rare)
- Network (Obsolete/Legacy/Rare)
- Multi-Value (Obsolete/Legacy/Rare)

### Early DBMSs

Early database management systems were difficult to build and maintain because there was a tight coupling between logical and physical layers. In other words, programmers had to (roughly) know what queries the application would execute before they could deploy the database.

The logical layer describes which entities and attributes the database has, while the physical layer is how those entities and attributes are actually being stored. In early DBMSs, the physical layer was defined in the *application* code (as opposed to being abstracted away), so if database administrators wanted to change how data was stored, they would need to change all of the application code to match the new physical layer.

For example, if we wanted to maintain a key-value store that supported equality lookups (find the value that matches this key), a logical data structure to use would be a hash map. However, if an application developer comes along and asks to perform a range scan over the keys (find all of the values with keys in between $a$ and $b$), a tree or B-tree would be a much better solution. In order to support both range scans and equality lookups, the database would either need to provide multiple interfaces, or the application code would need to be implemented twice, neither of which is appealing.

## 4 Relational Model

A mathematician named Ted Codd at IBM Research in the late 1960s noticed that people were rewriting DBMSs every time they wanted to change the physical layer. In 1969, he proposed the relational model to avoid this.

The relational model defines a database abstraction based on relations to avoid maintenance overhead. It has three key ideas:

- Store database in simple data structures (relations)
- Physical storage left up to the DBMS implementation
- Access data through a high-level language, where the DBMS figures out best execution strategy

The relational data model defines three concepts:

- **Structure:** The definition of relations and their contents independent of their physical representation
- **Integrity:** Ensure the database's contents satisfy certain constraints
- **Manipulation:** Programming interface for accessing and modifying a database's contents

One of the ideas that the relational model provides is *data independence*. If we isolate the user / application from the low-level data representation, the user only has to worry about the high-level application logic. It also allows the DBMS to optimize the data layout according to the operating environment, database contents, and workload. When these factors change, the DBMS can respond and re-optimize the database.

The relational model defines a few more things:

A *relation* is an unordered set that contains the relationship of attributes that represent entities. Since the relationships are unordered, the DBMS can store them in any way it wants, allowing for optimization. *It is possible to have repeated / duplicated elements in a relation.*

A *tuple* is a set of attribute values (also known as its *domain*) in the relation. In the past, values had to be atomic or scalar, but now values can also be lists or nested data structures. Every attribute can be a special value, NULL, which means for a given tuple the attribute is undefined.

A relation with $n$ attributes is called an *$n$-ary relation*. We will interchangeably use *relation* and *table* in this course. An $n$-ary relation is equivalent to a table with $n$ columns.

A relation's *primary key* uniquely identifies a single tuple in a table. Some DBMSs automatically create an internal primary key if you do not define one. A lot of DBMSs have support for autogenerated keys (so an application does not have to manually increment the keys), but a primary key is still required for some DBMSs.

A *foreign key* specifies that an attribute from one relation maps to a tuple in another relation. Generally, the foreign key will point / be equal to a primary key in another table.

A *constraint* is a user-defined condition that must hold for *any* instance of the database. Unique key and referential (foreign key) constraints are the most common.

## 5 Data Manipulation Languages (DMLs)

Methods to store and retrieve information from a database. There are two classes of languages for this:

- **Procedural:** The query specifies the (high-level) strategy the DBMS should use to find the desired result based on sets / bags. For example, use a `for` loop to scan all records and count how many records are there to retrieve the number of records in the table.
- **Non-Procedural (Declarative):** The query specifies only *what* data is wanted and not *how* to find it. For example, we can use SQL `SELECT COUNT(*) FROM artist` to count how many records are there in the table.

## 6 Relational Algebra

*Relational Algebra* is a set of fundamental operations to retrieve and manipulate tuples in a relation. Each operator takes in one or more relations as inputs, and outputs a new relation. To write queries we can "chain" these operators together (often in a tree or directed acyclic graph) to create more complex operations.

### Selection

Select takes in a relation and outputs a subset of the tuples from that relation that satisfy a selection predicate. The predicate acts as a filter, and we can combine multiple predicates using conjunctions and disjunctions.

Syntax: $\sigma_{\text{predicate}}(R)$.

Example: $\sigma_{\text{a\_id='a2'}}(R)$

SQL: `SELECT * FROM R WHERE a_id = 'a2'`

### Projection

Projection takes in a relation and outputs a relation with tuples that contain only specified attributes. You can rearrange the ordering of the attributes in the input relation as well as manipulate the values.

Syntax: $\pi_{\text{A1,A2,}\ldots\text{,An}}(R)$.

Example: $\pi_{\text{b\_id-100, a\_id}}(\sigma_{\text{a\_id='a2'}}(R))$

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

### Union

Union takes in two relations and outputs a relation that contains all tuples that appear in at least one of the input relations. Note: The two input relations have to have the exact same attributes.

Syntax: $(R \cup S)$.

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

### Intersection

Intersection takes in two relations and outputs a relation that contains all tuples that appear in both of the input relations. Note: The two input relations have to have the exact same attributes.

Syntax: $(R \cap S)$.

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

### Difference

Difference takes in two relations and outputs a relation that contains all tuples that appear in the first relation but not the second relation. Note: The two input relations have to have the exact same attributes.

Syntax: $(R - S)$.

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

### Product

Product takes in two relations and outputs a relation that contains all possible combinations for tuples from the input relations.

Syntax: $(R \times S)$.

SQL: `(SELECT * FROM R) CROSS JOIN (SELECT * FROM S)`, or simply `SELECT * FROM R, S`

### Join

Join takes in two relations and outputs a relation that contains all the tuples that are a combination of two tuples where for each attribute that the two relations share, the values for that attribute of both tuples is the same.

Syntax: $(R \bowtie S)$.

SQL: `SELECT * FROM R JOIN S USING (ATTRIBUTE1, ATTRIBUTE2...)`

### Observation

Relational algebra defines the fundamental operations to retrieve and manipulate tuples in a relation. It also defines an ordering of the high-level steps to compute a query.

For example, $\sigma_{\text{b\_id=102}}(R \bowtie S)$ represents joining $R$ and $S$ and then selecting / filtering the result. However, $(R \bowtie (\sigma_{\text{b\_id=102}}(S)))$ will do the selection on $S$ first, and then join the result of the selection with $R$.

These two statements will always produce the same answer. However, if $S$ has 1 billion tuples and there is only 1 tuple in $S$ with b_id=102, then $(R \bowtie (\sigma_{\text{b\_id=102}}(S)))$ will be significantly faster than $\sigma_{\text{b\_id=102}}(R \bowtie S)$.

A better approach is to state the high-level result you want (retrieve the joined tuples from R and S where $b_{id}$ equals 102), and let the DBMS decide the steps it should take to compute the query. SQL will do exactly this, and it is the de facto standard for writing queries on a relational DBMS.

## 7 Other Data Models

### Document Data Model

The document data model is a collection of record documents containing a hierarchy of named field/value pairs. A field's value can be either a scalar type, an array of values, or a pointer to another document.

Modern implementations use JSON. Older systems use XML or custom object representations.

The document model avoid "relation-object impedance mismatch" by tightly coupling objects and database.

While there are certainly use cases for this model, it still runs into many of the problems discussed in the flat file strawman example discussed earlier.

### Vector Data Model

The vector data model represents one-dimensional arrays used for nearest-neighbor search (exact or approximate). Vector databases are generally used for semantic search on embeddings generated by ML-trained transformer models (think ChatGPT), and native integration with modern ML tools and APIs (e.g., LangChain, OpenAI). At their core, these systems use specialized indexes to perform NN searches quickly.

Recently, many relational DBMSs have shipped vector index features or extensions (`pgvector`) that allow NN search within the relational model.

## 8 Conclusion

**Databases are ubiquitous.**

Relational algebra defines the primitives for processing queries on a relational database.

We will see relational algebra again when we talk about query optimization and execution.