Data Modelling & Databases

Janis Hutz
https://janishutz.com

June 22, 2026

"*A curry can be written in many different ways*"
- Prof. Dr. Gustavo Alonso, 2026

FS2026, ETHZ
Summary of the Lecture Slides
https://systems.ethz.ch/education/courses/2026-spring/dmdb.html

Contents

1 Introduction

A note on the quote on the title page: He did not *actually* mean "curry", but of course "query", but he just pronounced it that way.

This summary assumes some basic knowledge about databases. All words written in **bold** should be clickable and should link you to the corresponding term in the glossary. If I did not forget a term that is, of course.

1.1 IMS

Early databases were not relational and they also required the user of the system to know *how the data was stored*. This of course is not ideal, as any update to the data structure also required an update to the code interacting with it. In addition, manual query optimization has to be done.

Non-relational databases commonly have data redundancy, due to non-hierarchical applications being forced into a tree-based model. This causes issues when updating the data, as one needs to update the data in all places where a copy of the data exists.

1.2 Network Model

In a network model, one **record** is fetched at a time, so it essentially traverses the network. However, we again do not have data independence, thus the application again needs to change when the physical data representation changes. In addition, manual query optimization has to happen again, thus, any change to the physical data representation may cause the need to redo the optimization.

1.3 Relational Model

This is the focus of this course

The relational model has the benefit of having data independence: The application doesn't know how the data is stored and represented. The queries stay the same, even if the data representation changes. In addition, the end user does not have to worry about optimization too much. This, of course, only applies to a limited extent.

2 Relational Model, Logic & Algebra

2.1 Relational Model

In the relational model, relations are sets of tuples (similar to **records**). They are denoted $R(f_1 : D_1, \ldots, f_n : D_n)$, or sometimes simply $R(D_1, \ldots, D_n)$. An ***instance*** is a ***set*** of tuples $I_R \subseteq D_1 \times \ldots \times D_n$.

Intuition: You can think of the instance to be the rows in the table.

Important We define relations as ***mathematical sets***. This means that they cannot contain duplicates and the order does not matter. Database engines may elect to do this differently, but this course assumes set semantics for the relational model.

2.1.1 Keys

Definition 2.1.1 *(Candidate Key)* The minimal set of fields that identify each tuple uniquely

Definition 2.1.2 *(Primary Key)* A single candidate key, i.e. just a single field. We mark the primary key using underlining in visual **schemas**. Formally, we write $R(\underline{k : D_k}, a : D_a, b : D_b)$ for a given relation, with all valid instances fulfilling

$$I \subseteq D_k \times D_a \times D_b \land \forall (k, a, b), (k', a', b') \in I : k = k' \rightarrow (a, b) = (a', b')$$

i.e. if the key is equal, so is the rest of the tuple (thus they are one and the same[1]).

2.1.2 Queries

We write queries in a relational model using the following set notation (using **SQL** for an actual DB):

$$\{(r, p) \mid r \in \texttt{Supplies} \land p \in \texttt{Parts} \land \texttt{r.pid = p.pid} \land \texttt{p.name = "A"}\}$$

The query language thus processes an entire set at a time and applies set operations over the relations.

[1] Heidrun is her name

2.2 Relational Algebra

2.2.1 Operators

- ***Union*** $\cup$: $x \in R_1 \cup R_2 \Leftrightarrow x \in R_1 \vee x \in R_2$ (All tuples from both sets (Only valid for same schemas))
- ***Difference*** $-$: $x \in R_1 - R_2 \Leftrightarrow x \in R_1 \wedge x \notin R_2$ (Tuples that appear in R_1, but not in R_2)
- ***Intersection*** $\cap$: $x \in R_1 \cap R_2 = R_1 - (R_1 - R_2)$ (Tuples that don't appear in both sets)
- ***Selection*** σ: $x \in \sigma_c(R) \Leftrightarrow x \in R \wedge c(x) = \texttt{true}$, with c a predicate on the passed in set. (Tuples that fulfil the predicate c)
- ***Projection*** Π: $\Pi_{A_1,\dots,A_n}(R)$ (Keep only a subset of the columns, e.g. for columns `name`, `pid`, $\Pi_{\texttt{name}}(R)$ returns only the column `name`)
- ***Cartesian Product*** $\times$: $(x, y) \in R_1 \times R_2 \Leftrightarrow x \in R_1 \wedge y \in R_2$ (Primarily used to express join operations. This however simply joins together the two tables (i.e. similar to expanding terms in maths))
- ***Renaming*** ρ: $\rho_{B_1,\dots,B_n}(R)$ (Change the name of the attributes of R to B_i)
- ***Natural Join*** $\bowtie$: $R_1(A, B) \bowtie R_2(B, C) = \Pi_{A,B,C}(\sigma_{R_1.B=R_2.B}(R_1 \times R_2))$. (Join two relations into a table on a column. Natural join joins on columns with same name in both (or all) relations) Edge cases:
  - No shared attributes $R(A, B, C)$, $S(D, E)$: $R \bowtie S = R \times S$
  - All attributes shared $R(A, B, C)$, $S(A, B, C)$: $R \bowtie S = R \cap S$
- ***Theta Join*** $\bowtie_\theta$: $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$ (Join with custom predicate θ)
- ***Equi-Join*** $\bowtie_{A=B}$: $R_1 \bowtie_{A=B} = \sigma_{A=B}(R_1 \times R_2)$ (Join column A with column B)
- ***Semi-Join*** $\ltimes_C$: $R_1 \ltimes_C R_2 = \Pi_{A_1,\dots,A_n}(R_1 \bowtie_C$, with $R_1(A_1, \dots, A_n)$ and $R_2(B_1, \dots, B_m)$ (Returns columns only from one side if there is a match in the join)
- ***Relational division*** $\div$: $R \div S = \Pi_{R-S}R - \Pi_{R-S}((\Pi_{R-S}R) \times S - R)$. In other words, $R \div S = T$, with T being the *largest* relation such that $S \times T \subseteq R$. (Find all rows that do not fulfil a condition)

Semi-Join Reduction A useful trick with semi-joins. It is especially useful in distributed DB, where R and S are on different machines. Suppose we want to Join $R(A, B)$ with $S(B, C)$ on B, then we can do the following $R \bowtie S = (R \ltimes \Pi_B S) \bowtie S$.

The idea is to send the column on which the relations are to be joined from the second system to the first, there execute a semi-join (thus only returning the contents of R that matched with contents of S), then sending only that data to the second machine to finish the full join operation.

Of course, in the real world, users of DB systems don't write relational algebra, as in this case, the order of operations matters for performance, whereas we want database systems to figure this out by themselves.

3 SQL

SQL, or unabbreviated, Structured Query Language, is a **declarative** programming language, used to describe and operate on databases.

This is is the point to mention xkcd standards comic, because, as is the case with almost any standard, people deviate from it and new standards form.

Even though *most* **SQL** databases support the standard SQL operations, many have added a lot of features on top.

SQL is split up into the following parts:

- Data Definition Language, used to create, modify and delete schemas.
- Data Manipulation Language, used to insert, update and delete data
- Query Language, used to retrieve data

3.1 Data Definition Language (DDL)

The DDL is used to describe the schema of relations, in **SQL** called tables. For that, we provide a table name, the columns and their data types.

SQL supports the following data types (plus more):

- `char(n)`
- `varchar(n)`
- `integer`
- `blob` or `raw` (for large binaries)
- `date`

3.1.1 Creating Tables

Use the DDL `CREATE TABLE` keyword to create a table. We can set default values for certain attributes, or can add constraints to them.

File: `code/sql/ddl/create.sql`

```sql
CREATE TABLE TableName (
    Attribute integer,
    OtherAttribute varchar (30),
    NextAttribute character (2) default "AP", -- default value, if unset on insert
    PRIMARY KEY (Attribute) -- primary key, as in RA
);
```

3.1.2 Updating / Altering Tables

Use the DDL `ALTER TABLE` keyword to update a table's schema. Be aware that if we ADD a column to the schema, unless a default value is provided, the column is set to `NULL` (see **??**)

File: `code/sql/ddl/alter.sql`

```sql
ALTER TABLE TableName ADD COLUMN (col integer);
ALTER TABLE TableName ADD COLUMN (AnotherColumn integer default "");
ALTER TABLE TableName DROP COLUMN AnotherColumn;
```

3.1.3 Deleting Tables

File: `code/sql/ddl/delete.sql`

```sql
DROP TABLE TableName;
```

3.2 Data Manipulation Language (DML)

The DML is used to insert, update and delete data from the database.

Below some examples showing the use of the common operations. Note that you can use a `WHERE` clause from the query language to filter data for both `DELETE` and `UPDATE` statements.

`File: code/sql/dml.sql`

```sql
INSERT INTO TableName (
    Attribute,
    OtherAttribute
) VALUES ( 1000, 'Value' );

DELETE FROM TableName WHERE Attribute > 50;

UPDATE TableName SET Attribute = Attribute + 1;

-- Import data from a CSV file (this is pgsql specific syntax)
COPY TableName FROM '/data.csv' WITH FORMAT csv;
```

3.3 Query Language

The basic structure of a **SQL** statement is `SELECT I FROM T WHERE C`, which corresponds to $\Pi_{\mathtt{I}}(\sigma_{\mathtt{C}}\mathtt{T})$.

We can set `I = *` if we want to return all columns for a table. To rename, we can set `I = Column as Name, Column2 as Name2`, etc

Theorem 3.3.1 Every SPJR RA expression can be written in `SELECT ... FROM ... WHERE ...` form:

Operation	Notation	SQL
Selection	$\sigma_c(R)$	`SELECT * FROM R WHERE c;`
Projection	$\Pi_{A_1,\dots,A_n}R$	`SELECT A1, ..., An FROM R;`
Cartesian Product	$R_1 \times R_2$	`SELECT * FROM (R1, R2);`
Rename	$\rho_{a,b,c}R$	`SELECT A as a, ..., B as c FROM R;`
Union	$R_1 \cup R_2$	`R1 UNION R2;`
Difference	$R_1 - R_2$	`R1 EXCEPT R2;`
Intersection	$R_1 - R_2$	`R1 INTERSECT R2;`

Since **SQL** implements bag and not set semantics, there is a `DISTINCT` keyword, which is used to remove duplicates.

For the last three operations, `R1` and `R2` typically are other **SQL** statements:

```sql
(SELECT name FROM FirstTable) UNION (SELECT name FROM SecondTable)
```

3.4 Operations

Keyword	Description
`SELECT val, val2`	returns the columns `val` and `val2`
`FROM table d, table2 c`	selects the target tables. Can use range variables (`d, c` here)
`WHERE`	Filter results. Can also target non-acquired columns
`DISTINCT`	Returns elements without duplicates
`IN`	Checks if the element is present in subquery
`UNION`	Set union (also removes duplicates)
`INTERSECT`	Set intersection
`EXCEPT`	Set difference
`JOIN`	Join tables, always requires a ON clause
`ON`	Specify on which column with what condition to join the tables
`DATE`	Used to do comparisons against fixed Dates (followed by Datestring)
`AS`	Rename a column
`AVG(column)`	Computes the average of the specified column
`SUM(column)`	Computes the sum of the specified column

We can directly add new columns to our result using statements like (also note that we do not actually return salary and employment columns)

```sql
SELECT name, salary * employment / 100 AS RealSalary FROM employees;
```

When applying `GROUP BY`, COUNT and the like operate on each group, not entire dataset.

`NATURAL JOIN` doesn't require a ON clause (it looks for columns with same name)

After `GROUP BY` we can't use `WHERE`, instead we can use `HAVING`, which operates on groups instead of on rows.

We can compare against a query with a single result column and row using a `WHERE` clause.

The `CROSS JOIN` is the both-sided extension to `LEFT JOIN` and `RIGHT JOIN`

We can compute the number of all passed grades for a person using something like

```sql
SUM(CASE Grade >= 1 THEN 1 ELSE 0 END) AS PassedSubjectsCount
```

Glossary

declarative Describes *what* to achieve, but not how.

imperative Describes how to achieve something.

record .

schema A set of relations.

SQL Structured Query Language.