Data Modelling & Databases

Janis Hutz
https://janishutz.com

July 1, 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)`
- `INT`
- `BIGINT`
- `SMALLINT`
- `FLOAT`
- `BLOB(n)` (for large binaries)
- `DATETIME`
- `MONEY`

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.

Since the primary key must be unique for each entry, it may be useful to configure the primary key attribute with the following constraints `PrimaryKeyField INT NOT NULL AUTO_INCREMENT`

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 3.3.4)

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;
```

3.1.4 Constraints

SQL supports a number of constraints that can be put on the different attributes of a schema.

- `NOT NULL`: Disallows this column to be set to `NULL`, or have a `NULL` value
- `UNIQUE`: The value of each value in this column must be unique, but it can be `NULL`. An arbitrary number of columns can have `NULL` and still be valid

- `PRIMARY KEY`: Sets this column as the primary key. `NOT NULL` and `UNIQUE` is set on it implicitly. Contrary to common intuition, it can be set on multiple columns.
- `CHECK c`: Check that the values fulfil a condition. This condition has the same syntax as for the `WHERE` clause (see 3.3.1)
- `REFERENCES table`: A Foreign Key, used to refer to another tuple from a different relation. It is typically referencing the primary key of the other table.

For `REFERENCES`, there are many options for maintenance, which can be set on creating a reference:

- ***Cascade***: Propagate `UPDATE` and `DELETE`
- ***Restrict***: Prevent deletion of the primary key before the change, causes error
- ***No Action***: Prevent modifications after attempting change, causes error
- ***Set default, Set Default***: Set references to `NULL` or default value

Example 3.1.4

```
CREATE TABLE tab (id integer REFERENCES OtherTable ON DELETE cascade ON UPDATE cascade)
```

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. It is to be applied in the `SELECT` clause (`I` above), e.g.

```sql
SELECT DISTINCT name FROM Data;
```

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

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

In addition, to apply *bag semantics*, as opposed to *set semantics* append `ALL` to the expression (e.g. `UNION ALL`)

We can also name the tables, e.g. `T = One o, Two t`, and then access columns from a specific table using `I = o.Col, b.Col`, or the like.

3.3.1 Basic operators

3.3.1.1 Logical Operators

The following **SQL** logical operators are available (may not be exhaustive for all systems, `cop` is a comparison operator):

Operator	Description
`P1 AND P2`	TRUE if `P1` and `P2` both are true
`P1 OR P2`	TRUE if one of `P1` or `P2`, or both are true
`C cop ALL query`	TRUE if all values `x` returned in **subquery** `query` fulfil `C cop x`
`C cop ANY query`	TRUE if one (or more) values `x` returned in **subquery** `query` fulfil `C cop x`. `SOME` is logically equivalent
`C BETWEEN low AND high`	TRUE if numerical, text or date value of `C` is between `low` and `high`. Both ends inclusive, negatable
`C IN list`	TRUE if value of `C` is in the provided `list`. List format: `(val, val2)`. List can also be a **subquery**, negatable
`C LIKE R`	TRUE if value of `C` matches regex-like pattern `R`. (% matches any number of chars, _ matches a single, arbitray character)
`EXISTS query`	TRUE if **subquery** `query` retuns at least one row

3.3.1.2 Comparison & Arithmetic Operators

Comparison Operators $<, <=, =, >=, >$ are defined as usually, $<>$ is defined as not equal.

Arithmetic Operators $+, -, *, /, \%$ are defined as usual.

We can use the arithmetic operators like this:

```
SELECT Value * 1.1 FROM Data;
```

or in `WHERE` clauses like this:

```
SELECT Value FROM (Data a, Data b) WHERE a.Value = b.Value - 1;
```

Remark 3.3.2 You may have noticed that in the above query, the same table was used twice. This is referred to as a ***Self-Join*** and can come in handy when comparing values in a single table.

3.3.2 Join

Specifying two (or more) tables in the `FROM` clause will perform a cartesian product, e.g.

```
SELECT * FROM (Table1, Table2);
```

returns all rows of the tables, next to each other.

Here, naming the tables in the `FROM` clause may come in handy, to not write out the entire name of the tables:

```
SELECT * FROM (Table1 t, Table2 s) WHERE t.id = s.id;
```

```
SELECT * FROM (Table1, Table2) WHERE Table1.id = Table2.id;
```

Both of the above statements produce the same result.

3.3.2.1 Nested Queries

To not have the cartesian product executed in the `FROM` clause, or to get a much more complex starting table, we can use a **subquery**. These are parenthesized normal SQL queries.

We can create an alias for them using an `AS name` clause after the parenthesis as follows

```
SELECT r.name FROM (SELECT * FROM Table) AS r;
```

or alternatively, using a `WITH` clause (replace the dummy queries with real queries):

```
WITH CteName (SELECT * FROM Table) SELECT name FROM CteName;
```

3.3.2.2 Join Operations

If we don't specify the kind of join, an `INNER JOIN` is executed. The following three queries are equivalent

```
SELECT * FROM Student, Tests WHERE Student.PersNr = Tests.PersNr;
```

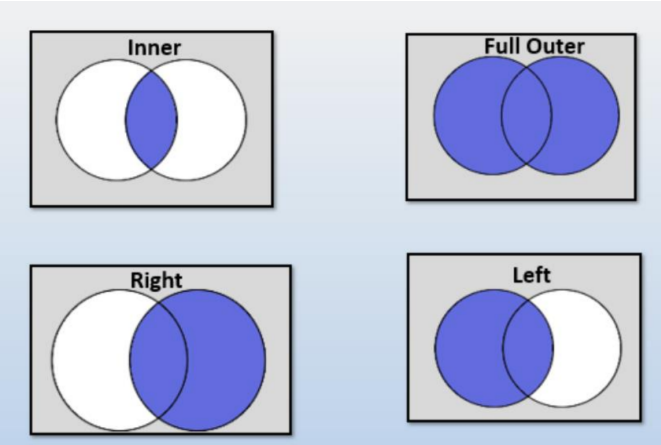
```
SELECT * FROM Student JOIN Tests ON Student.PersNr = Tests.PersNr;
```

```
SELECT * FROM Student JOIN Tests USING (PersNr);
```

For the latter of the three, the parenthesis around the column are important, as omitting them is invalid syntax.

Other types of `JOIN` are:

- `NATURAL JOIN` (no `ON` required), joins on columns with same name
- `LEFT/RIGHT/FULL OUTER JOIN` (requires `ON`), also returns non-matching rows from left, right or both sides, respectively



3.3.3 Aggregations, Grouping, Sorting

3.3.3.1 Aggregations

SQL supports (at least) the aggregation functions `SUM`, `MIN`, `MAX`, `AVG`, `COUNT`, whose output should be evident.

Please note that you cannot use the aggregation functions and still output another column, as they produce a single value.

For each of the functions, you can optionally specify `DISTINCT`.

We typically want to specify an `AS` clause, to make the output nicer to read

3.3.3.2 Grouping

Grouping is used to, well, group together the outputs on a column. Using SQL, this is achieved using the `GROUP BY` clause. We specify the columns on which the grouping should occur.

IMPORTANT We can only use aggregates and columns appearing in the `GROUP BY` clause in the `SELECT` clause.

Using a `JOIN` function, it is of course then possible to go back retrieve the other columns.

To apply filtering on the columns, the `HAVING` clause can be used. It applies a comparison to a grouping column or an aggregate as specified in the `SELECT` clause:

```sql
SELECT COUNT(name) as cnt FROM Students GROUP BY Department HAVING cnt > 1;
```

3.3.3.3 Sorting

We can sort the output of any query using the `ORDER BY column DIRECTION` clause, where `DIRECTION` is to be replaced with `ASC` or `DESC`:

```sql
SELECT name FROM Students ORDER BY name DESC;
```

3.3.3.4 Limit

We can limit the number of columns to return using the `LIMIT` clause. Example:

```sql
SELECT * FROM Exams LIMIT 10;
```

3.3.3.5 Example

Below some examples of **SQL** Queries using grouping, aggregations, sorting and grouping

File: code/sql/aggregations.sql

```sql
SELECT MAX(price) as MAX_PRICE FROM Products; -- Get the most expensive product
SELECT AVG(grade) as AVG_GRADE FROM Exams WHERE id = 0; -- Get the average grade for an exam

-- Aggregation with grouping (Remember, only aggregates and cols in GROUP BY clause can
-- appear in SELECT clause!)
SELECT id, AVG(grade) as "Average Grade", COUNT(grade) as "Count"
FROM ExamResults GROUP BY id; -- Average grade for each exam and number of students taking it

SELECT id, AVG(grade) as avg, COUNT(grade) as cnt FROM ExamResults
GROUP BY id HAVING cnt > 0 ORDER BY avg DESC LIMIT 20;

-- Get exam details for all exams as filtered before
SELECT * FROM (
    SELECT id, AVG(grade) as avg, COUNT(grade) as cnt FROM ExamResults
    GROUP BY id HAVING cnt > 0 ORDER BY avg DESC LIMIT 20
) as filtered JOIN Exams e ON filtered.id = e.id;
```

3.3.4 Null

Whenever we have incomplete information, we can set a column to be `NULL`. If you have a column that has `NULL` values, when using `WHERE` clauses, these values are skipped. *However* if you are not filtering on the column, a `NULL` value is returned.

To check if a column is `NULL`, you can use the `IS` keyword:

```
SELECT * FROM Data WHERE name IS NULL
```

- In `GROUP BY`, all `NULL`s form a single group.
- `COUNT(*)`, also counts rows with `NULL`
- `COUNT(TheColWithNull)` ignores rows with `NULL`
- Most other aggregations ignore `NULL`
- If all rows in an aggregations are `NULL`, so is the result

The `OUTER JOIN` operators may (and likely will) introduce `NULL`

3.3.5 Views

Views are a useful way of abstracting a complex query and is the result of a stored query.

We can create views using the following template:

```
CREATE VIEW name AS query;
```

We can then operate on it as if it were a table.

Common applications for views are privacy / access control, because you can limit access to tables only via certain views for every user; Usability is also often much better due to having less complex queries to write - you are just writing them once.

How a view is evaluated This is quite simple in essence, the `VIEW`'s query is inserted into the query using the view. It then is then optimized as it normally would be, too. Of course, it is likely that more optimization can and has to be done on views.

3.3.5.1 Updatable Views

Updating Views refers to updating values of the tables in the views. This is not straight forward and thus, many restrictions are placed on what kind of views are considered updatable.

In **SQL**, a view is updatable if and only if it involves ***one*** base relation and its key, it does ***not*** involve aggregates, groups or duplicate elimination.

In short, if there is one-to-one mapping between the view and base relation, it is updatable.

4 Theoretical Background

4.1 Functional Dependency

As with relational algebra, we again define a schema to be a relation, e.g. $\mathcal{R}(A : D_A, B : D_B, C : D_C, D : D_D)$, with an instance of it being $R \subseteq D_A \times D_B \times D_C \times D_D$.

Then, we let $\alpha, \beta \subseteq \mathcal{R}$, i.e. they each have a subset of the columns of $\mathcal{R}$.

$\alpha \to \beta$ (β is a functional dependency of α) if and only if $\forall r, s \in R : r.\alpha = s.\alpha \implies r.\beta = s.\beta$

We write $R \models \alpha \to \beta$ if R satisfies $\alpha \to \beta$ semantically, and $R \vdash \alpha \to \beta$ if the same applies syntactically (i.e. we can apply inference rules over Functional Dependencies).

Intuition: This means that there is a function that maps the values of columns α to the values of columns β.

Definition 4.1.1 $\alpha \subseteq \mathbb{R}$ is a **superkey** if and only if $\alpha \to \mathcal{R}$.

Intuition: This means that if we know the values of columns α, we know the value of the rest of the columns in $\mathcal{R}$. It is a superset of some candidate keys (see 2.1.1)

Definition 4.1.2 $\alpha \to \beta$ is minimal if and only if $\forall A \in \alpha : (\alpha - \{A\}) \not\to \beta$. These are denoted $\alpha \to^{\cdot} \beta$

4.1.1 Inference

A fundamental result for a given set of Functional Dependencies F is $F \models \alpha \to \beta \Leftrightarrow F \vdash \alpha \to \beta$ (assuming $\vdash$ is defined by Armstrong's Axioms)

The goal is to find new FDs that can are implied from a set of FDs F on scheme $\mathcal{R}$. Let $\alpha \to \beta$ be another FD on $\mathcal{R}$. Then:

- F implies $\alpha \to \beta$, if every relation instance R of $\mathcal{R}$ that satisfies all FDs in F also satisfies $\alpha \to \beta$.
- The ***closure*** $F+$ of F is the set of all FDs implied by F.
- Let G be another set of FDs on scheme $\mathcal{R}$. F and G are ***equivalent*** ($F \equiv G$) if $F \models G$ and $G \models F$

4.1.2 Armstrong's Axioms

- ***Reflexivity*** $\alpha \subseteq \beta \implies \beta \to \alpha$, special case: $\mathcal{R} \to \alpha$ (referred to as *trivial FDs*)
- ***Augmentation*** $\alpha \to \beta \implies \alpha\gamma \to \beta\gamma$, with $\alpha\gamma = \alpha \cup \gamma$
- ***Transitivity*** $\alpha \to \beta \land \beta \to \gamma \implies \alpha \to \gamma$

These three axioms are all complete and sound. All other possible FDs can be implied from these axioms.

- F ***derives*** $f = \alpha \to \beta$, denoted $F \vdash f$, if there is a derivation for f using only Armstrong's axioms.
- F ***implies*** f, denoted $F \models f$, if for every relation instance R of $\mathcal{R}$ that satisfies all FD in F also satisfies f.

Definition 4.1.3 *(Soundness)* $F \vdash f \to \implies F \models f$

Definition 4.1.4 *(Completeness)* $F \models f \implies F \vdash f$

Definition 4.1.5 *(Closure)* of α with respect to F α^+ is the set of all attributes $y \in \mathcal{R}$ such that $\alpha \to y$ can be derived from F using Armstrong's axioms:

$$\alpha^+ = \{y \in \mathcal{R} \mid F \vdash \alpha \to y\} \qquad F \vdash \alpha \to \beta \Leftrightarrow \beta \subseteq \alpha^+$$

4.1.3 Other rules

- ***Union of FDs*** $\alpha \to \beta \land \alpha \to \gamma \implies \alpha \to \beta\gamma$
- ***Decomposition*** $\alpha \to \beta\gamma \implies \alpha \to \beta \land \alpha \to \gamma$
- ***Pseudo Transitivity*** $\alpha \to \beta \land \beta\gamma \to \theta \implies \alpha\gamma \to \theta$

Algorithm 1 Finding Closure

```
1 procedure FindClosure(F)
2     α⁺ ← α
3     repeat
4         α⁺_old ← α⁺
5         for each FD β → γ ∈ F do
6             if β ⊆ α⁺ then
7                 α⁺ ← α⁺ ∪ γ
8     until α⁺ = α⁺_old
9     return α⁺
```

4.1.4 Closure Algorithm

For the definition of the closure, see above.

Finding a closure α^+ using an algorithm[2]:

We can use that to check the following:

- $F \vdash \alpha \to \gamma$: Calculate α^+ and check $\gamma \in \alpha^+$
- $F \vdash G$: For each $\alpha \to \gamma \in G$, check $F \vdash \alpha \to \gamma$
- F equivalent to G: Check $F \vdash G$ and $G \vdash F$
- $K \subseteq \mathcal{R}$ **superkey** for F: Check $F \vdash K \to \mathcal{R}$:

Minimal Cover **Definition 4.1.6**

A minimal cover (or minimum basis) of F is a set of FDs G that has the following properties:

- $G \equiv F$
- All FDs in G have form $X \to A$, with A being a single attribute
- It is not possible to make G "smaller", i.e. if deleting a FD, $G - \{X \to A\} \not\equiv G$, or $G - \{XA \to B\} + \{X \to B\} \not\equiv G$, for any FD $XA \to B \in G$

Computing the minimum basis:

1. G set of FDs obtained from F by decomposing the right hand sides of each FD to single attribute
2. Remove trivial FDs
3. Remove redundant attributes from LHS of FDs in G (by, for each $X \to Y$ taking each attribute $x \in X$ and, if $X - \{x\} \to Y$ implies $X \to Y$, replacing $X \to Y$ with $X - \{x\} \to Y$)
4. Remove all redundant FDs

4.1.5 Using Functional Dependencies

Functional dependencies help with decomposing relations into several relations that each contain just one concept, since relations combining several concepts are bad.

[2] Yes, I did name the algorithm that. Sorry about that, the pun was too good to skip

4.2 Normal Forms

Definition 4.2.1 *(Normal forms)* describe the properties of concrete schemas based on functional dependencies in terms of data redundancy and data integrity.

For each of the normal forms, we need to be able to decide:

- whether $\{R, F\}$ satisfies the NF
- given $\{R, F\}$ that satisfies the NF, what are the properties?
- how can we generate new schema R', such that $\{R', F\}$ satisfies the given normal form

4.2.1 First Normal Form

In the first normal form, only atomic domains are allowed, i.e. keys are not allowed to be arrays.

Intuition: This makes the concept of a key easier to define, or well defined

Some **SQL** flavours support arrays. However, the semantics and the support varies

This can come in handy to solve one to many relationships, but what if many different persons have the same elements there? This can cause some issues when updating.

Arrays are very tempting to use if we have an unknown and variable number of different data points for a field.

4.3 Second Normal Form

For a relation to be in 2NF, every non-key attribute needs to minimally dependent on ***every*** key, i.e. no attribute depends on part of a key. If relations are not in 2NF, they may experience insert, update and delete anomalies, which can lead to incorrect, redundant or inconsistent updates of the relations.

However, some relations can still suffer from update and / or delete anomalies

4.3.1 Third Normal Form

A relation R is in 3NF if and only if for all $\alpha \to B$, at least one of the following conditions holds:

- $B \in \alpha$ (then $\alpha \to B$ is a trivial FD)
- B is an attribute of at least one key
- α is a **superkey** of R

Intuition: if $\alpha \to B$ does not satisfy any of these conditions, α is a concept in its own right, thus, the 3NF tries to get rid of "transitive dependencies" (e.g. $A \to B, B \to C$)

The 3NF can still experience update and delete anomalies.

4.3.2 Boyce-Codd Normal Form (BCNF)

R is in BCNF if and only if for all $\alpha \to B$ at least one of the following conditions holds:

- $B \in \alpha$ (thus, $\alpha \to B$ is a trivial FD)
- α is a **superkey** of R

Intuition: In each relation, you only store the same information once.

The following FD are okay in both 3NF and BCNF because both left hand sides are candidate keys for the table.

- $\{$PersNr$\} \to$ Name, Level, Room, City, Street, Canton
- $\{$Room$\} \to$ PersNr

4.3.3 BCNF Decomposition Algorithm

In words, this algorithm takes as input a schema $\mathcal{R}$ and outputs a list of schemas $\mathcal{L} = \mathcal{R}_1, \ldots, \mathcal{R}_n$, such that $\mathcal{L}$ is a lossless decomposition of $\mathcal{R}$ (i.e. joining them together will perfectly recover the information in $\mathcal{R}$) and each of the $\mathcal{R}_i$ are in BCNF.

Of note is that this does ***not*** preserve the functional dependencies and some data redundancies will still remain, only the ones caused by functional dependencies.

In addition, the order of picking the evil FD matters, so we use heuristics, which pick the one with the largest right hand side.

Algorithm 2 Decomposition algorithm

```
1 procedure DECOMPOSITION(R)
2     result ← {R}
3     while ∃R_i in result such that R_i is not in BCFN do
4         let α → β be the evil FD
5         R_i^1 ← α ∪ β
6         R_i^2 ← R_i − β
7         result ← (result − R_i) ∪ {R_i^1, R_i^2}
8     return result
```

Algorithm 3 Synthesis algorithm

```
1 procedure SYNTHESIS(R)
2     Compute the minimal basis F_c of F
3     for all α → β ∈ F_c do
4         create R_{α∪β}(α ∪ β)
5     If none of the above relations contains a superkey, add a relation with a key
6     Eliminate R_α if there exists R'_α such that α ⊆ α'
```

4.3.4 3NF Synthesis Algorithm

In words, this algorithm takes as input a schema $\mathcal{R}$ and outputs a list of schemas $\mathcal{L} = \mathcal{R}_1, \dots, \mathcal{R}_n$, such that $\mathcal{L}$ is a lossless decomposition of $\mathcal{R}$ and each of the $\mathcal{R}_i$ are in 3NF.

This algorithm preserves all functional dependencies, however does not remove redundancies because it is not BCNF.

5 Systems

5.1 Query Optimization

A lot of performance can be gained in a database system by executing the query in the correct order. For that, a query optimizer exists, which uses a quite large amount of metadata to speed up query execution.

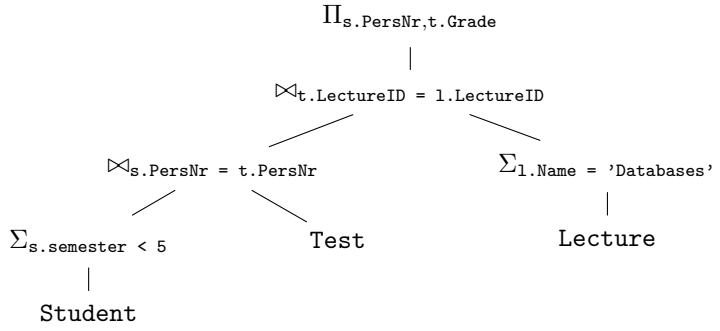
A query optimizer very broadly works as follows:

1. It searches the space of equivalent execution plans. This space tends to be huge, so efficient exploration is needed.
2. Of all the discovered plans, the best is chosen using a cost model, which describes the most efficient plan

5.1.1 Search Space

To create the search space, the optimizer splits a query into a collection of query blocks, where each query block has one `SELECT` and `FROM` clause and *at most* one `WHERE`, `GROUP BY` and `HAVING` clause.

Query plans are typically drawn up as trees, as they are typically easier to understand that way than as both SQL query or in relational algebra notation. In addition, the query parser commonly transforms it into a tree-like data structure, so thinking about query plans in this way is a good habit. For the symbols used here, see section 2.2.



5.1.1.1 Rewriting Rules

Given a logical plan as above, there are multiple ways in which we can construct a physical plan. We can use different join orders, decide when selection happens and specific implementations of operators.

The following rules were discussed in the lectures:

R1 Conjunctive selection operators can be deconstructed into a sequence of individual selections:

$$\sigma_{\theta_1 \wedge \theta_2} = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

This is useful to split selections to push them earlier in the execution plan (reduces the number of elements to process subsequently)

R2 Selection operators are commutative:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

This is useful to allow to first put the one that e.g. has an index.

R3 Only the last in a sequence of projections is needed, others can be omitted

$$\Pi_{t_1}(\Pi_{t_2}(E)) = \Pi_{t_1}(E)$$

This avoids going over the data twice

R4 Selections can be combined with cartesian products and theta joins

$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2 \qquad \sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

This combines the join and selection (thus avoiding having to go over the data twice)

R5 Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

This is useful so we ccan choose as the smaller table as the outer table or as the inner one the one with an index.

R6 Natural join operations are associative, so are theta joins (with restrictions)

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3) \qquad (E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

This allows for joins to be performed in different orders, allowing us to do the most selective first (fewer rows)

R7 Pushdown selection:

$$\sigma_\theta(E_1 \bowtie E_2) = \sigma_\theta(E_1) \bowtie E_2$$

This allows us to first remove the data that is not needed before doing the more expensive operations

R8 Projections distribute over theta joins (applies if θ involves only attributes from $L_1 \cup L_2$ of course)

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1}(E_1) \bowtie_\theta \Pi_{L_2}(E_2)$$

This allows the table width to be reduced before the join to reduce ***data movement***

R9 Union and Intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1 \qquad E_1 \cap E_2 = E_2 \cap E_1$$

This allows us to perform one side before the other, depending on resource constraints

R10 Set union and intersection are associative

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3) \qquad (E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

This allows us to change the order in which operations are done and how the operator tree is executed.

R11 The selection operation distributes over $\cup, \cap$ and $-$

$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2) \qquad \sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2) \qquad \sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$$

In addition, we have

$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2 \qquad \sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap E_2 \qquad \text{not for } \cup$$

This allows us to remove unneeded tuples before performing the next operation, which may be involving comparisons

R12 The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = \Pi_L(E_1) \cup \Pi_L(E_2)$$

This allows us to remove columns before the union, requiring less ***data movement***

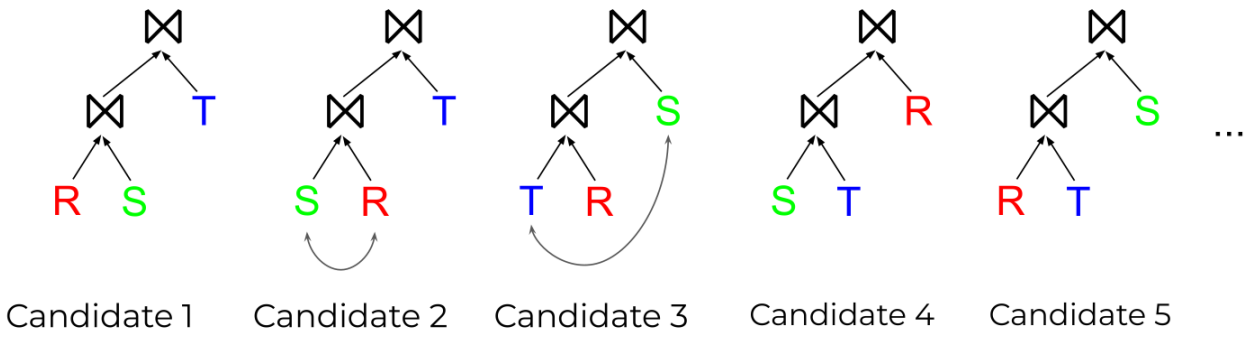


Figure 5.1: Example application of the rewriting rules. (Figure from lecture slides 08, slide 21)

5.1.2 Execution Model

There are a number of ways in which the execution can happen

5.1.2.1 Materialization Model

Here, each operator processes its entire input all at once and then emits its output all at once.

This is ideal, if the intermediate results are not much larger than the final results, as very little overhead is created. The efficiency of this approach however is more limited the larger the intermediate results are.

5.1.2.2 Iterator Model

Here, each operator is an iterator, it takes as input a set of streams of tuples (each of which provides a `next()` function) and outputs a stream of tuples, that can in turn again be accessed using a `next()` function. To get a result, we run `root.next()` again and again, until no more data is available to be pulled in.

This is a ***Volcano Mode***, data flows from bottom to top.

This method has the benefits of there being a single interface for all operators, thus we don't (need to) know what the previous operator did. Furthermore, iterators are easy to implement and there are little to no memory overheads and the whole thing can be pipelined, parallelized and distributed.

On the other hand, method calls introduce a large overhead and suffer from poor instruction cache locality.

5.1.2.3 Vectorization Model

This method is similar to the iterator model, but instead of a single tuple, the `next()` invocation returns a batch of tuples.

We can also use vector instructions on the batch of tuple, which reduces overhead by requiring fewer function and instruction calls.

Of course, the limitation here is that the hardware needs to support vector instructions, though nowadays that is hardly an issue, since AVX instructions are supported on CPUs as old as the Sandy Bridge Architecture (e.g. Intel Core i7-2600K) for Intel (2011) and Bulldozer (e.g. AMD FX-8100) for AMD (2011).

5.1.3 Cost Model

A cost model is used to ***estimate*** the computation cost of a given physical plan, without actually running it.

Since the ccost of running each operator depends on the input and output size of it, we can easily convert this into cost.

Thus, to estimate the cost, we need to know the runtime of an operator and the number of inputs and outputs. The first of these two aspects is of course very easily attainable, whereas the ***cardinality estimation***, i.e. the estimation of the number of tuples an operator returns, is very hard.

To make this a bit easier, commonly a ***selectivity*** or ***reduction factor*** is used, which tells us how much of the input an operator returns.

Below some reduction factor estimators (we assume that the values are uniformly distributed)

Predicate	Reduction Factor	Predicate Example
`col = val`	`1 / #UniqueValues(col)`	`col = 2`
`col > val`	$\lvert\max(\texttt{col}) - \texttt{val}\rvert \div (\max(\texttt{col}) - \min(\texttt{col}))$	`col > 5`
`col < val`	$\lvert\texttt{val} - \min(\texttt{col})\rvert \div (\max(\texttt{col}) - \min(\texttt{col}))$	`col < 5`
`col1 = col2`	$1 \div \max(\#\texttt{UniqueValues(col1)}, \#\texttt{UniqueValues(col2)})$	`t.name = s.name`

This table shows the overall concept for the cardinality estimation. For each predication, a reduction factor is determined, as each predicate "filters" out some tuples.

Thus, the resulting cardinality is $\max(\#\text{tuples}) \cdot \prod_{r \in RF} r$, where RF is the **bag** of all reduction factors.

In all of the above estimators, we assumed that the values are uniformly distributed. This of course is not always true — there are many cases where data is heavily skewed — in which cases, histograms are used.

Two types of histograms are commonly used:

- ***Equi-Width***: Here, the heights (tuple counts) of the buckets differ, as we fix the values for each bucket. This is useful for identifying hot-spots.
- ***Equi-Depth*** (or Equi-Height): Here, the width differs, because we want all buckets to contain the same number of tuples. The size of a range helps with cardinality estimates
- ***Singleton*** (or Frequency): Plots how often a single item appears in a table. This is very useful to compute the selectivity of queries

To enable quick processing of the cost function, the following metadata, or statistics, are typically stored:

Table statistics

- Number of rows
- Number of blocks
- Average row length

Column statistics

- Number of distinct values (NDV) in columns
- Number of nulls in column
- Data distribution (histogram)

Extended statistics

- Index statistics
- Number of leaf blocks
- Levels
- Clustering factor

System statistics

- I/O performance and utilization
- CPU performance and utilization

5.1.4 Search

We now know how to generate a set of semantically equivalent physical plans and how to estimate the cost for each of them. The question that remains is how to efficiently find the best plan.

Since real-world queries can be very complex, searching for the best physical plan can be hard. There are a few options, but we only covered a simple approach.

We can constrain the search space by only considering ***left-deep join trees***, i.e. trees that have only leafs on the right hand side of each join statement. The rationale behind this is that many of these are fully pipelined plans and no intermediate results have to be written to temporary files. This is the concept `System R`, the first relational database used.

An example for a non-pipelined left-deep join tree is one in volving sort operations. This brings the complexity down to $\mathcal{O}(n!)$, where n is the number of realations in the join.

- Enumerate join orders (different left deep trees)
- Enumerate plans for each operator (hash join, sort merge join, nested loop join, ...)
- Enumerate access methods for each operator (B+-Tree, hash table, sequential scan, ...)

This can be achieved using dynamic programming.

We then find the best 1-relation plan, then the best 2-relation plan by finding ways to join a relation with the best 1-relation plan, etc.

While this is still slow, it's faster than enumerating all possible join trees.

Later, systems started introducing heuristics to reduce the number of choices that must be made in a cost-based fashion.

Heuristic optimization transforms the query-tree using a set of rules typically improve execution performance:

- Perform selection early (reduces the number of tuples to process)
- Perform projection early (reduces the number of attributes to process)
- Perform most restrictive selection and join operations before other similar operations

Some systems only use heuristics, while others combine them with partial cost-based optimization.

5.2 Database Engines

5.2.1 Storage Management

To make database engines easy to use, some abstractions have to happen.

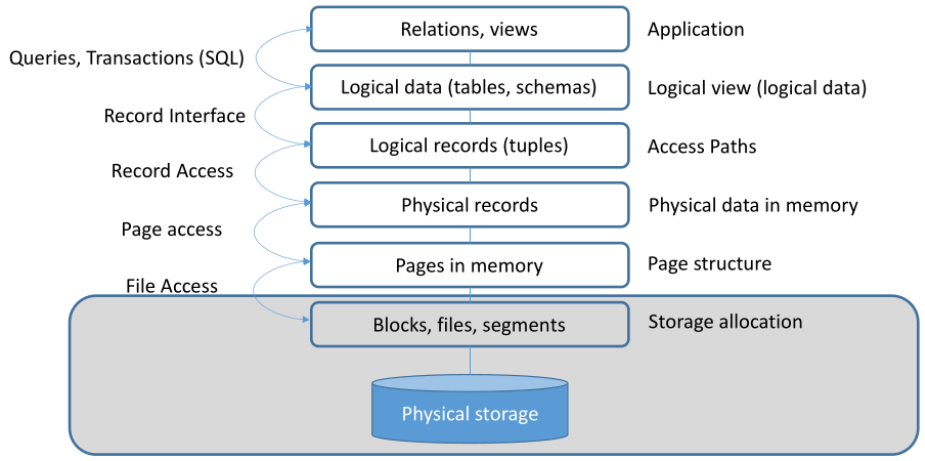


Figure 5.1: Overview of a typical abstraction for a database system (Figure from lecture slides 09, slide 3)

In the design of database engines it is commonly assumed that the data is stored on spinning disks, i.e. HDDs, with a high latency for seeks (random accesses) and that most of the DB is not in memory.

Very modern database systems may assume that a higher percentage of the DB is stored in memory due to the higher quantity of memory available, as well as lower access latencies due to the more readily available SSDs and high-speed NASes.

Some database systems operate similarly to operating systems in certain aspects, in that they manage their own processes and all of the memory to get the absolute maximum performance out of the systems.

Glossary

bag A bag, or multiset, is a set, in which, (as is usual with sets) order does not matter, but duplicates are allowed.

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

imperative Describes how to achieve something.

record .

schema A set of relations.

SQL Structured Query Language.

subquery Also Nested Query, a separate query that can be used in the `FROM` or `WHERE` clause of another query. See 3.3.2.1.

superkey See Section 4.1.