

Data Modelling & Databases

Janis Hutz
<https://janishutz.com>

July 8, 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	4
1.1	IMS	4
1.2	Network Model	4
1.3	Relational Model	4
2	Relational Model, Logic & Algebra	5
2.1	Relational Model	5
2.1.1	Keys	5
2.1.2	Queries	5
2.2	Relational Algebra	6
2.2.1	Operators	6
3	SQL	7
3.1	Data Definition Language (DDL)	7
3.1.1	Creating Tables	7
3.1.2	Updating / Altering Tables	7
3.1.3	Deleting Tables	7
3.1.4	Constraints	7
3.2	Data Manipulation Language (DML)	9
3.3	Query Language	9
3.3.1	Basic operators	10
3.3.2	Join	11
3.3.3	Aggregations, Grouping, Sorting	12
3.3.4	Null	13
3.3.5	Views	13
4	Theoretical Background	14
4.1	Functional Dependency	14
4.1.1	Inference	14
4.1.2	Armstrong's Axioms	14
4.1.3	Other rules	14
4.1.4	Closure Algorithm	15
4.1.5	Using Functional Dependencies	15
4.2	Normal Forms	16
4.2.1	First Normal Form	16
4.3	Second Normal Form	16
4.3.1	Third Normal Form	16
4.3.2	Boyce-Codd Normal Form (BCNF)	16
4.3.3	BCNF Decomposition Algorithm	16
4.3.4	3NF Synthesis Algorithm	17
5	Systems	18
5.1	Query Optimization	18
5.1.1	Search Space	18
5.1.2	Execution Model	19
5.1.3	Cost Model	20
5.1.4	Search	21
5.2	Database Engines	22
5.2.1	Storage Management	22
5.2.2	Physical to logical storage	22
5.2.3	Record Layout	23
5.2.4	Data in the blocks	23
5.2.5	Row and Column Stores	24
5.2.6	Execution Models	25
5.3	Indexing	26
5.3.1	Access methods	26

5.3.2	B+ Trees	26
5.3.3	Hashing	28
5.3.4	Bitmaps	30
5.3.5	Bloom Filters	31
5.3.6	Zone maps	31
5.3.7	AR Trees	31
5.3.8	R Trees	31
5.3.9	Other Optimizations	31
	Glossary	32

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 $R(f_1 : D_1, \dots, f_n : D_n)$, or sometimes simply $R(D_1, \dots, D_n)$. An **instance** is a **set** of tuples $I_R \subseteq D_1 \times \dots \times D_n$.

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

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

i.e. if the key is equal, so is the rest of the tuple (thus they are one and the same¹).

2.1.2 Queries

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

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

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** $\cap$: $x \in R_1 \cap R_2 \Leftrightarrow x \in R_1 \wedge x \in R_2$ (Tuples that don't appear in both sets)
- **Selection** σ : $x \in \sigma_c(R) \Leftrightarrow x \in R \wedge c(x) = \text{true}$, 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`, $\Pi_{\text{name}}(R)$ 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** $\bowtie_{A=B}$: $R_1 \bowtie_{A=B} R_2 = \sigma_{A=B}(R_1 \times R_2)$ (Join column A with column B)
- **Semi-Join** $\ltimes_C$: $R_1 \ltimes_C R_2 = \Pi_{A_1, \dots, A_n}(R_1 \bowtie_C R_2)$, 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 $R \ltimes S = (R \ltimes \Pi_B S) \bowtie S$.

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 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) | ▪ BIGINT | ▪ BLOB(n) (for large binaries) |
| ▪ VARCHAR(n) | ▪ SMALLINT | ▪ DATETIME |
| ▪ INT | ▪ FLOAT | ▪ 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

```

1 CREATE TABLE TableName (
2     Attribute integer,
3     OtherAttribute varchar (30),
4     NextAttribute character (2) default "AP", -- default value, if unset on insert
5     PRIMARY KEY (Attribute) -- primary key, as in RA
6 );

```

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

```

1 ALTER TABLE TableName ADD COLUMN (col integer);
2 ALTER TABLE TableName ADD COLUMN (AnotherColumn integer default "");
3 ALTER TABLE TableName DROP COLUMN AnotherColumn;

```

3.1.3 Deleting Tables

File: code/sql/ddl/delete.sql

```

1 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

```
1 INSERT INTO TableName (
2     Attribute,
3     OtherAttribute
4 ) VALUES ( 1000, 'Value' );
5
6 DELETE FROM TableName WHERE Attribute > 50;
7
8 UPDATE TableName SET Attribute = Attribute + 1;
9
10 -- Import data from a CSV file (this is postgresql specific syntax)
11 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 $\Pi_I(\sigma_C T)$.

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	$R_1 \cap R_2$	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.

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

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

```
(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.Col1, b.Col1, 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
<code>P1 AND P2</code>	TRUE if P1 and P2 both are true
<code>P1 OR P2</code>	TRUE if one of P1 or P2, or both are true
<code>C cop ALL query</code>	TRUE if all values <code>x</code> returned in subquery query fulfil <code>C cop x</code>
<code>C cop ANY query</code>	TRUE if one (or more) values <code>x</code> returned in subquery query fulfil <code>C cop x</code> . SOME is logically equivalent
<code>C BETWEEN low AND high</code>	TRUE if numerical, text or date value of <code>C</code> is between <code>low</code> and <code>high</code> . Both ends inclusive, negatable
<code>C IN list</code>	TRUE if value of <code>C</code> is in the provided list. List format: <code>(val, val2)</code> . List can also be a subquery , negatable
<code>C LIKE R</code>	TRUE if value of <code>C</code> matches regex-like pattern <code>R</code> . (<code>%</code> matches any number of chars, <code>_</code> matches a single, arbitrary character)
<code>EXISTS query</code>	TRUE if subquery query returns 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;
```

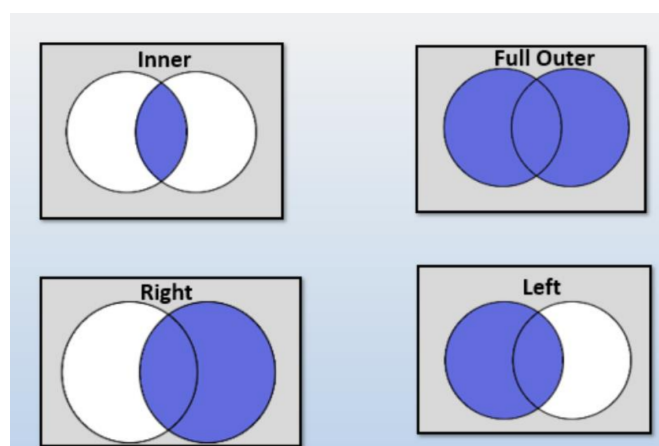
```
SELECT * FROM Student JOIN Tests ON Student.PersNr = Tests.PersNr;
```

```
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:

```
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:

```
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:

```
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

```

1 SELECT MAX(price) as MAX_PRICE FROM Products; -- Get the most expensive product
2 SELECT AVG(grade) as AVG_GRADE FROM Exams WHERE id = 0; -- Get the average grade for an exam
3
4 -- Aggregation with grouping (Remember, only aggregates and cols in GROUP BY clause can
5 -- appear in SELECT clause!)
6 SELECT id, AVG(grade) as "Average Grade", COUNT(grade) as "Count"
7 FROM ExamResults GROUP BY id; -- Average grade for each exam and number of students taking it
8
9 SELECT id, AVG(grade) as avg, COUNT(grade) as cnt FROM ExamResults
10 GROUP BY id HAVING cnt > 0 ORDER BY avg DESC LIMIT 20;
11
12 -- Get exam details for all exams as filtered before
13 SELECT * FROM (
14     SELECT id, AVG(grade) as avg, COUNT(grade) as cnt FROM ExamResults
15     GROUP BY id HAVING cnt > 0 ORDER BY avg DESC LIMIT 20
16 ) 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 NULLs 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 $\mathcal{R}$.

$\alpha \rightarrow \beta$ (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 $R \models \alpha \rightarrow \beta$ if R satisfies $\alpha \rightarrow \beta$ semantically, and $R \vdash \alpha \rightarrow \beta$ 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 $\alpha \subseteq \mathcal{R}$ is a **superkey** if and only if $\alpha \rightarrow \mathcal{R}$.

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

4.1.1 Inference

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

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

- F implies $\alpha \rightarrow \beta$, if every relation instance R of $\mathcal{R}$ that satisfies all FDs in F also satisfies $\alpha \rightarrow \beta$.
- 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** $\alpha \subseteq \beta \implies \beta \rightarrow \alpha$, special case: $\mathcal{R} \rightarrow \alpha$ (referred to as *trivial FDs*)
- **Augmentation** $\alpha \rightarrow \beta \implies \alpha\gamma \rightarrow \beta\gamma$, with $\alpha\gamma = \alpha \cup \gamma$
- **Transitivity** $\alpha \rightarrow \beta \wedge \beta \rightarrow \gamma \implies \alpha \rightarrow \gamma$

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

- F **derives** $f = \alpha \rightarrow \beta$, 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 $F \vdash f \implies F \models f$

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 $\alpha \rightarrow y$ can be derived from F using Armstrong's axioms:

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

4.1.3 Other rules

- **Union of FDs** $\alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma \implies \alpha \rightarrow \beta\gamma$
- **Decomposition** $\alpha \rightarrow \beta\gamma \implies \alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma$
- **Pseudo Transitivity** $\alpha \rightarrow \beta \wedge \beta\gamma \rightarrow \theta \implies \alpha\gamma \rightarrow \theta$

Algorithm 1 Finding Closure

```

1 procedure FINDCLOSURE( $F$ )
2    $\alpha^+ \leftarrow \alpha$ 
3   repeat
4      $\alpha_{\text{old}}^+ \leftarrow \alpha^+$ 
5     for each FD  $\beta \rightarrow \gamma \in F$  do
6       if  $\beta \subseteq \alpha^+$  then
7          $\alpha^+ \leftarrow \alpha^+ \cup \gamma$ 
8   until  $\alpha^+ = \alpha_{\text{old}}^+$ 
9   return  $\alpha^+$ 

```

4.1.4 Closure Algorithm

For the definition of the closure, see above.

Finding a closure α^+ using an algorithm²:

We can use that to check the following:

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

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 $X \rightarrow A$, with A being a single attribute
- It is not possible to make G “smaller”, i.e. if deleting a FD, $G - \{X \rightarrow A\} \not\equiv G$, or $G - \{XA \rightarrow B\} + \{X \rightarrow B\} \not\equiv G$, for any FD $XA \rightarrow B \in G$

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 $X \rightarrow Y$ taking each attribute $x \in X$ and, if $X - \{x\} \rightarrow Y$ implies $X \rightarrow Y$, replacing $X \rightarrow Y$ with $X - \{x\} \rightarrow Y$)
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.

²Yes, 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 $\alpha \rightarrow B$, at least one of the following conditions holds:

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

Intuition: if $\alpha \rightarrow B$ 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. $A \rightarrow B, B \rightarrow C$)

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 $\alpha \rightarrow B$ at least one of the following conditions holds:

- $B \in \alpha$ (thus, $\alpha \rightarrow B$ 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.

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

4.3.3 BCNF Decomposition Algorithm

In words, this algorithm takes as input a schema $\mathcal{R}$ and outputs a list of schemas $\mathcal{L} = \mathcal{R}_1, \dots, \mathcal{R}_n$, 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( $\mathcal{R}$ )
2    $\text{result} \leftarrow \{\mathcal{R}\}$ 
3   while  $\exists \mathcal{R}_i$  in  $\text{result}$  such that  $\mathcal{R}_i$  is not in BCNF do
4     let  $\alpha \rightarrow \beta$  be the evil FD
5      $\mathcal{R}_i^1 \leftarrow \alpha \cup \beta$ 
6      $\mathcal{R}_i^2 \leftarrow \mathcal{R}_i - \beta$ 
7      $\text{result} \leftarrow (\text{result} - \mathcal{R}_i) \cup \{\mathcal{R}_i^1, \mathcal{R}_i^2\}$ 
8   return  $\text{result}$ 

```

Algorithm 3 Synthesis algorithm

```

1 procedure SYNTHESIS( $\mathcal{R}$ )
2   Compute the minimal basis  $F_c$  of  $F$ 
3   for all  $\alpha \rightarrow \beta \in F_c$  do
4     create  $R_{\alpha \cup \beta}(\alpha \cup \beta)$ 
5   If none of the above relations contains a superkey, add a relation with a key
6   Eliminate  $R_\alpha$  if there exists  $R'_\alpha$  such that  $\alpha \subseteq \alpha'$ 

```

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.

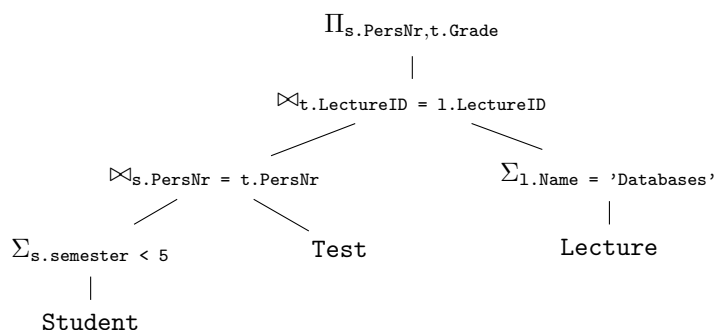
A query optimizer very broadly works as follows:

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

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

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 can 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)

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

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

$$E_1 \cup E_2 = E_2 \cup E_1 \quad E_1 \cap E_2 = E_2 \cap E_1$$

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

R10 Set union and intersection are associative

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

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 $-$

$$\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2) \quad \sigma_{\theta}(E_1 \cup E_2) = \sigma_{\theta}(E_1) \cup \sigma_{\theta}(E_2) \quad \sigma_{\theta}(E_1 \cap E_2) = \sigma_{\theta}(E_1) \cap \sigma_{\theta}(E_2)$$

In addition, we have

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

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**

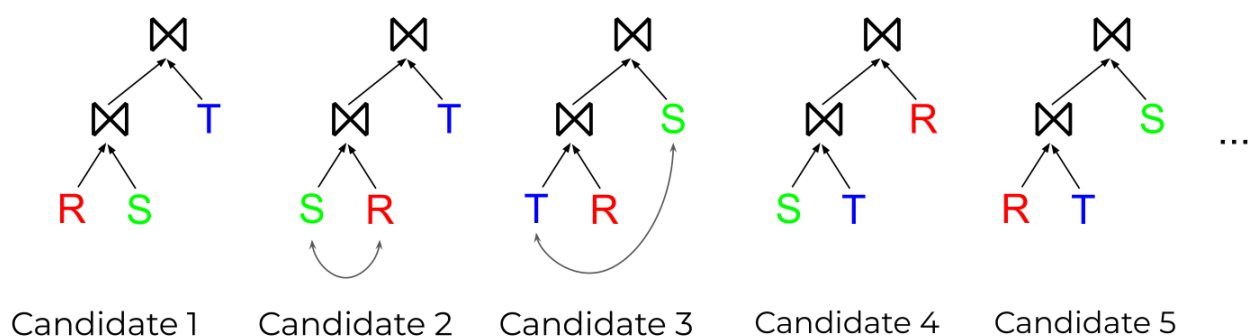


Figure 5.1: Example application of the rewriting rules. (Figure from lecture slides 08, slide 21)

5.1.2 Execution Model

There are a number of ways in which the execution can happen. More details can be found in section 5.2.6

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 cost 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
<code>col = val</code>	$1 / \text{\#UniqueValues}(\text{col})$	<code>col = 2</code>
<code>col > val</code>	$ \text{max}(\text{col}) - \text{val} \div (\text{max}(\text{col}) - \text{min}(\text{col}))$	<code>col > 5</code>
<code>col < val</code>	$ \text{val} - \text{min}(\text{col}) \div (\text{max}(\text{col}) - \text{min}(\text{col}))$	<code>col < 5</code>
<code>col1 = col2</code>	$1 \div \text{max}(\text{\#UniqueValues}(\text{col1}), \text{\#UniqueValues}(\text{col2}))$	<code>t.name = s.name</code>

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 $\text{max}(\text{\#tuples}) \cdot \prod_{r \in RF} r$, 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 involving sort operations. This brings the complexity down to $\mathcal{O}(n!)$, where n is the number of relations 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.

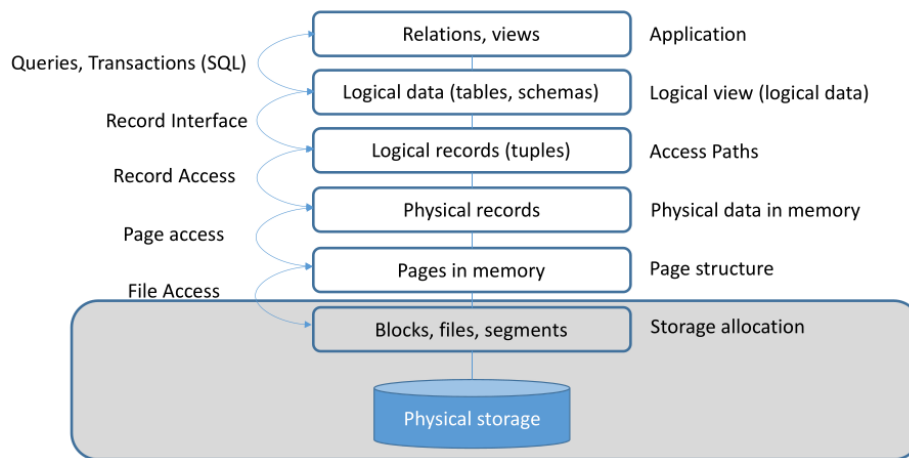


Figure 5.1: Overview of a typical abstraction for a database system (Figure from lecture slides 09, slide 3)

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.

5.2.2 Physical to logical storage

To the left, in figure 5.1, the relationship between the different storage abstractions are connected using crow's feet.

Since database tables often include large amounts of data, that would span many OS pages, thus database systems use blocks as opposed to OS pages.

These database blocks are structured as follows:

- **Header:** Contains the address and type of segment (that being index, table, etc)
- **Table directory:** Contains the schema of the table stored in the block
- **Row directory:** Contains pointers to the actual tuples stored in the block. Grows downwards (like the stack)
- **Free Space:** Well... it's free real estate, isn't it?
- **Row data:** The tuples stored in the block. Grows upwards (like the heap). The pointers in the row directory point here

Typically, database systems allow inserts of new tuples until there is about 20% of free space left. This space is allocated to allow for updates of existing tuples, as they may become larger. Thus, a larger storage footprint is used to prevent having to move all the data to a different location when the data block inevitably fills up, avoiding thrashing on the page.

Some systems even go a step further, only again unlocking a data block for writing if its used percentage falls below 40%.

Of course, as with any storage system, fragmentation becomes an issue. Compaction is only done when the block has enough space for an INSERT or UPDATE, but the space is not contiguous.

An UPDATE may require that a tuple is moved into a different block. In that case, the tuple is inserted into a new block and a pointer is stored in the freed space.

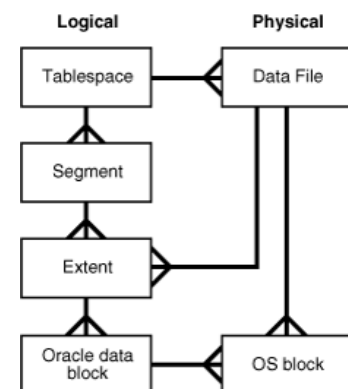


Figure 5.1: Entity relationship diagram, where the crow's feet imply one-to-many relationships (Figure from [Oracle Docs](#))

5.2.3 Record Layout

Each record (commonly called tuple) contains a header (which in turn contains validity flags for deletion, visibility information for concurrency control, bit maps for null values and more), and the non-null attributes, or pointer to those attributes.

Relational engines do not have to store the schema for the tuple inside of it because that is set globally for the entire table.

Schema-less systems however have to store that information because it can differ for each record.

5.2.3.1 Optimizations

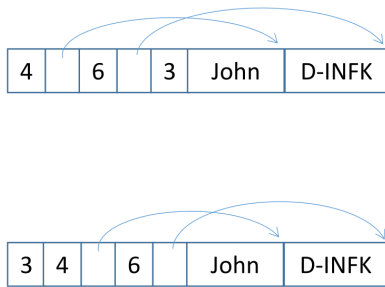


Figure 5.1: Record layouts in memory (Figure from Lecture 09, Slide 19)

It is crucial to store the data in a way that allows quick accesses and updates.

Each variable length attribute also has to store the length, so in a simple system we would simply store them together with the data. This however is not ideal for performance reasons (we have linear access time), thus we keep the fixed size part (attributes and lengths) at the start of the record and move the variable length part to the end, putting pointers after the length attributes that point to the end of the variable length region. Even more performance can be gained by reordering the attributes altogether, moving the variable length attributes to the very end and then applying the above optimization.

5.2.3.2 Data Types

- **Integers:** Most **DBMS** use the same format as C.
- **Real numbers:** Either in IEEE-754 (for variable precision) or using fixed point representations, which commonly are variable length, by storing all digits, plus where the decimal point is (they are NOT stored as strings)
- **Strings** and **BLOBs:** Store the length and the data
- **Time, Coordinates, points, ...:** System specific.

For large attributes, commonly instead of storing the whole tuple, only the fixed length attributes of the tuples are stored directly in the block, with pointers to the rest of the attributes, that may also lay outside of the current block. BLOBs may also be very large, taking up more than a single block.

5.2.4 Data in the blocks

The pages allocated to a given object (table, index, etc) are managed through lists, which are stored as part of the segment header

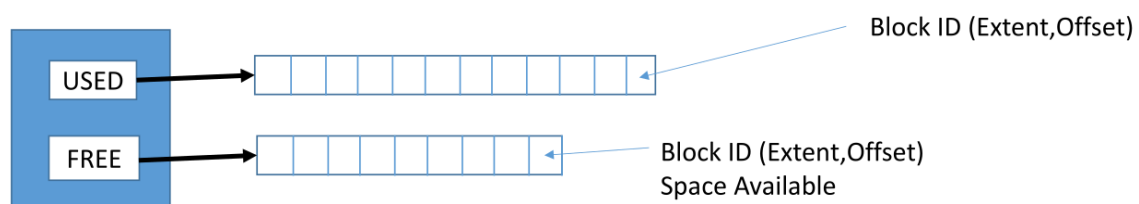


Figure 5.2: Free/used lists in databases (Figure from Lecture 09, Slide 24)

These lists can become a bottleneck, especially for transactions that result in modifications. There are some ways to improve the performance:

- We can use several free lists, so concurrent transactions can find free space in parallel, rather than having conflicts all the time
- Make the traversal of the list faster (sorting, etc)
- Make sure that holes can be found efficiently (store the available space in each page in incremental steps)

5.2.4.1 Slotted Pages

To find tuples within pages, slotted pages are used often. In slotted pages, each page has a header containing checksum, version, transaction visibility, compression information, utilization, etc.

Then, each tuple gets an id (typically a block ID and an offset).

The page maintains a list of the “slots” (each slot is assigned to one tuple and thus the number of tuples in the pages is further constrained by the number of pages available). The offsets, or sometimes pointers, are stored at the beginning of each tuple.

Advantages of slotted pages are:

- storing tuples of different sizes (and variable length tuples)
- Each tuple gets a permanent tuple / record id, that does not change
- When data doesn't change, space is used efficiently
- When data *does* change, we need to be careful

Considerations for when data *does* change:

- If a tuple is modified and becomes larger, the original space is used to store a pointer to the new location of the tuple
- If a tuple is deleted, we just remove the pointer
- If a tuple is modified and becomes smaller, leave the unnecessary space empty
- For insertion, we look for a page with enough free space and compact the page, if needed

The percentage free measure is used to avoid fragmentation caused by pages running out of space to accommodate updates for tuples that become larger. Thus, some space is reserved for growing the tuples, rather than for inserting.

Without this measure, the block can constantly go from FREE to USED and back and forth.

5.2.5 Row and Column Stores

5.2.5.1 Row Store

Here, as we have assumed thus far, tuples are stored with all their attributes together. This means that accesses to tuples is quick and easy.

This scheme is used by actual systems, but they also have many additional details added to improve management and speed.

A typical ideal application for a row store is Online Transaction Processing (OLTP). This is visible for example with banking applications, shopping carts, etc, where the operations mostly are on a single tuple.

However, for complex queries, the row store can become cumbersome.

5.2.5.2 Column Store

This is where the column store comes in, in which the data is stored by columns. This obviously speeds up operations on the columns, as the access is very quick.

Column stores also have the advantage that they present the data exactly in the representation needed for **SIMD** instructions such as **AVX**, which is now widely available on modern CPUs.

Modern database systems have started to adopt column stores to address the memory wall, since Column Stores are a more compact representation and cache lines are likely to bring the data we want to see, thus the cache utilization tends to be higher than with row stores.

Column store databases are today primarily used today for analytical databases and in-memory databases.

An alternative is a hybrid approach, called **Partition Attributes Across (PAX)** representation. There, each block contains several tuples, but it is organized as a column store.

5.2.6 Execution Models

We model execution as a tree of operators (as seen previously). At the root, we have the query results, at the leaves we have the tables. Typically, each operator has *at most* two inputs from lower layers (join operators have two, the rest typically has one).

We remember that we can use any subtree of a query as input for the higher parts of the tree, since we can “materialize” the result as a table.

5.2.6.1 Basic Execution Models

Each node (operator) in the tree operates independently, i.e. does input, putput, output.

Each operator can read their input tuples in one go, or one at a time and the same goes for the output. Some operators may also be *blocking*, i.e. no result can be issued until all operations are completed.

5.2.6.2 More advanced models

As a reminder, we already covered some aspects of three execution models in section 5.1.2.

5.2.6.2.1 Iterator Model

Some typical dataflow operators:

- **Union**: read both sides, issue all tuples (if they have the same schema)
- **Union without duplicates**: We first union, then we check for duplicates
- **Select**: We apply a filter function to the tuples
- **Projection**: For each tuple, we output only the specified attributes
- **Join**: Trivial nested loop join
- **Join with index on inner relation**: For all tuples, fetch matchin tuple in S_j = another operator

Advantages

- Generic interface for all operators
- Easy to implement operators
- Supports buffer management strategies
- No overhead in terms of main memory
- Supports pipelining
- Supports parallelism and distribution

Disadvantages

- High overhead of method calls
- Poor instruction cache locality

5.2.6.3 Pull & Push Mode

Definition 5.2.3 Pull Mode Intuitively like dynamic programming: Data is obtained by one operator through a function call to a lower operator. It is good for disk based systems and when data doesn't fit into memory and is also really easy to implement.

Definition 5.2.4 Push Mode This is the Pull Mode flipped on its head. While it *can* be faster, there need to be buffers everywhere and it is significantly harder to implement.

5.2.6.4 EXCHANGE operator

The EXCHANGE operator is useful for parallel processing, as it simply moves data from one place to another. This allows to parallelize a plan across one or more machines.

This can be implemented as a sort of “driver” on top of an operator, by calling `next()` on the operator a few times, collecting the results and sending them to the other side. The other side uses the data to respond to further next calls to the original operator.

5.3 Indexing

5.3.1 Access methods

Definition 5.3.1 Access Methods typically refers to the operations and data structures used for reading data from the buffer cache into the private space of the worker thread

The basic access method is **sequential scan**, where all pages of the table are read from beginning to end. Indexing is used to avoid having to scan the whole table.

There are many indexing techniques, but we will focus on three indexing techniques in this section, hashing, B+ trees and bitmaps.

Hashing is primarily used to randomize access to the data, finding data based on a key (hash and access concept) and both as an index for tables and used to implement operators.

B+ trees on the other hand are used to order data along the values of one attribute, but require traversal, unless we are looking for consecutive values. They are thus only used as an indexing mechanism.

5.3.2 B+ Trees

5.3.2.1 B-Tree

In B-Trees of order k , each node (except the root) has between $k \div 2$ and k child nodes. The root node has at least two children, unless it is also a leaf and any non-leaf node contains at exactly $k - 1$ keys.

Even if database systems say they use B-Trees, they do in fact use B+ trees.

5.3.2.2 Basics

B+ Trees are B-Trees, but the data (or the pointers to it) is at the leaves only, which are organized as a linked list. They are balanced trees, their inner nodes correspond to blocks and the leaf nodes correspond to blocks. The blocks are organized like slotted pages for variable length data (see Section 5.2.4.1)

The keys in the inner nodes may not correspond to the actual data and are instead used as separators. Typically, leaf nodes contain pointers to the tuples (value, key), where value is the actually indexed value and key is the pointer to the full data tuple, typically as a row id or tuple id. There are some systems that allow storing the data directly on the leaves.

Some systems create a B+ tree index by default for all tables and index the key. If there is no key, the engine assigns random keys and indexes them.

5.3.2.3 Clustered Indexes

An index orders the table by the attribute it is indexing, but the tuples in the table might not be ordered. This is where **clustered indexes** come into play: They force the tuples to be stored in the same order as the index indicates, thus, the table is physically stored in a sorted manner. This is typically done only for the primary key, and automatically done in systems that store data in the leaf nodes directly.

This eliminates random memory accesses caused by looking up ranges, since the tuples are stored in consecutive order, according to the index.

This is most useful for tables that are infrequently updated, since they require maintenance.

5.3.2.4 What to index?

This is a crucial question, as it can massively speed things up, but equally significantly slow things down.

B+ trees can use one or *more* attributes as the key to the index. If we have just one attribute, those are the values. If there are several attributes, it builds a composite key, which is useful when we are looking for combinations of values on certain attributes, or a table is searched by several attributes.

We then compare the keys lexicographically:

$$(a_1, a_2) < (b_1, b_2) \Leftrightarrow (a_1 < b_1) \vee (a_1 = b_1 \wedge a_2 < b_2)$$

Some values can also be left unspecified, but this is typically only done to the ones at the beginning of the key.

5.3.2.5 Composite Index

For a **SQL** statement such as

```
1 SELECT department_id, last_name, salary
2 FROM employees WHERE salary > 5000
3 ORDER BY department_id, last_name;
```

if we assume a composite index on `department_id`, `last_name`, `salary`, then the data is sorted by the three attributes in that order.

We can then perform a **full index scan**, where read the data from the leaves in sorted order and filter on salary. This avoids having to scan the entire table and the results is also already sorted.

5.3.2.6 Non-unique values

A B+ tree index can also be built on attributes containing also non-unique values.

This is a problem, because we cannot find all the duplicates with the basic design. To resolve this, we can repeat the key at the leaf nodes for every duplicated entry, or we can store the key once, but point to a linked list of all matching entries.

If the data is stored in the leaves, we append the tuple ID to know what tuple the entry refers to, because otherwise, they are all the same.

5.3.2.7 Operations

5.3.2.7.1 Direct Lookup

We traverse the tree from the root, then within each node, we use binary search to look for the correct entry. When we reach a leaf node, we return the corresponding pointer and the position.

5.3.2.7.2 Range Lookup

We return all tuples that match, by first finding the first tuple matching, the continue traversal until the last match is found. We can determine the last match easily because the tree's leafs are lexicographically sorted. Since the leafs are linked as linked lists, we can simply move along the linked lists, cutting down on processing time.

5.3.2.7.3 Inserting into the index

We first look up the corresponding leaf. Then, if there is space, we simply insert the value. If there is no space in the leaf, we split the leaf into two new leafs and insert the new item on the corresponding leaf.

All that remains to do now is to insert the separator on the parent node. If that node is full, we split the node in two and insert the separator into the parent node and propagate up to the root, if necessary.

5.3.2.7.4 Deleting from the index

We proceed by looking up the corresponding leaf, then we remove the entry from the leaf. If the leaf now is less than half full, we check the neighboring leaf

- If it is more than half full, we balance the two leafs
- If it is half full, we merge both leaves

For both operations, we update the separator in the parent node.

5.3.2.8 Concurrent Access

Indexes are heavily used while they are being maintained, thus creating conflicting, concurrent accesses.

A way to prevent this is using **lock coupling**, where we lock a page and its parent and move down the tree.

This prevents problems when the page must be split, however, this is not enough, if we have to go further up to apply the changes.

To prevent this, on every node we check if there is space for one more entry. If not, we split the node. This way, the split of a page never leads to propagating changes.

Alternatively, we can try lock coupling, and if we have to do a change that propagates up, we abort, release everything and start again from the top, but now lock the entire path.

5.3.2.9 Bulk Inserts

To create a B+ Tree index, we proceed as follows (the tree is created bottom up (from the leaves up))

1. Sort the data
2. Use the sorted data to fill one block after the other
3. Remember the largest value in each block
4. create the inner nodes by using the largest value in each block as separator
5. iterate upwards to the next level, until there is only one block left (the root)

If the blocks are filled completely, we get a clustered and compact tree. However, since we usually expect that data has to be updated, it is advised to leave some space in the each block to accommodate that without expensive propagating changes.

5.3.2.10 Optimizations

Reverse Index If the indexed attribute is a sequence and new items are constantly produced, we may encounter an issue where values next to each other go into the same block, leading to concurrent updates fighting to insert. Depending on how updates are handled, many copies could be produced.

Reverse indexing solves that issue by reversing the indices, meaning that consecutive indices go to different pages.

Many Keys Depending on attributes, many keys can become expensive to process. There are several possible optimizations:

- Replace separators that correspond to actual keys with shorter separators with the same effect (can be hard)
- Factor common prefix (this makes a lot of sense and is usually easy, too)

Ignoring rules We can also ignore the rules of the B+ trees and e.g. not merge nodes when they do not have enough data, delaying a merge (and instead rebuilding the tree periodically)

Variable Length Keys This is a bit of a problem. We can mitigate it by no longer forcing the number of entries to lay between $k \div 2$ and k , and using smaller values as separators, placing long values in leafs.

Making sure there is space We don't ever fill blocks to the max and make sure there is space on creation.

Depth vs Breadth For slow devices, prefer larger node sizes, thus shallower trees; For fast devices, prefer deeper trees.

5.3.3 Hashing

5.3.3.1 Hashing in Databases

Hashing is a very common operation in many systems, including databases. Many internal data structures, such as buffer caches and lock tables, are implemented as hash tables. Some operators also use hashing to speed things up, such as hash join and group by operators.

It can also be used as an indexing and partitioning strategy.

Hashing is on the other side compared to B+ trees when it comes to the trade-off between storage and compute, by using more compute, it can save on storage space requirements.

5.3.3.2 Hash Tables

There are many hashing functions with strong properties, but in databases, the hash function has to be very cheap, as it is used very often.

Perfect hash functions do exist, but they need a hash table that is as big as the cardinality of the attributes, so for 4 byte keys, it needs a 4 GB table. Thus, to save on storage, we use a hash table that is smaller than that. If we were to choose it too small, there would be too many collisions, if it is chosen too large, then we waste lots of space.

Collisions may also occur when the index key is not **UNIQUE**, and thus it could be that multiple tuples have the same key.

That may lead you to think, why not grow it if we have a collision. That is a valid idea, but growing hash tables is expensive, thus should be avoided.

Another shortcoming of hash indexes is that it only supports **point queries**, i.e. it only works for the primary key, but barely anything else.

5.3.3.3 Collisions

A way to handle collisions safely is to use **chaining**. When a collision occurs, we add another entry in a linked list. If these lists are reasonably short, then it is reasonably efficient. We can also already reserve space for the linked lists to speed things up, if we expect (many) collisions to occur.

Another option is to use **open addressing**. Here, when a collision occurs, we look for an empty slot in the hash table using some rule. Here, two ways of achieving this are covered in the course:

- **Linear probing**, where we go to the next slot(s) and place the attribute there
- **Cuckoo hashing**, where we use several hash functions and move to the next one, if we have a collision with the first, and so on

5.3.3.4 Extensible Hashing

When the hash table is full, in a basic system, we would need to create a new, larger hash table with more buckets (typically double the number) and then rehash all existing items. This of course is very expensive both computationally and in terms of memory accesses, as we do lots of random accesses, which leads to cache misses.

This is where extensible comes in. It is based on sharing buckets and unsharing them when needed. These hash table buckets are mapped into blocks and initially several buckets share the same block

Then, when a bucket gets full, we split the bucket and move entries as needed into a new bucket.

In **local sharing**, the size of a table can be doubled without immediately adding more space, allowing for more splitting. We simply increase the number of buckets and the degree of sharing and split the buckets as they become full.

Again, if we initially provide more space than needed, we can allow for the first growth to happen without disruption.

We want (ideally) to have two page lookups to access an item, so we typically have a lookup in the bucket directory and the data block and they both can grow independently.

5.3.3.5 Linear hashing

Here, a split pointer is used to indicate which bucket will be split in case of overflow. When it occurs, we chain the block that overflows, split the bucket indicated by the pointer and move the pointer.

This allows for a gradual increase in the size of the table and the idea is to gradually redistribute the data. We always split on the pointer, even if the overflow is elsewhere. The pointer will eventually reach a bucket with chains and when the bucket is split, the data is reorganized.

This has the advantage that the directory (i.e. the list of buckets) grows page by page, instead of doubling.

When all buckets have been split, start fresh.

5.3.3.6 In Postgres

PostgreSQL uses a variation of linear (overflow) hashing and extensible hashing (with doubling). Buckets are initially allocated in groups with powers of two. Overflows are captured in overflow pages (like in linear hashing). When a split occurs, we do not allocate all the pages, but a fraction of them.

These ideas can be combined in many different ways.

5.3.3.7 When to use

For memory structures:

- For shared resources (System Global Area)
- Database Buffer Cache for I/O
- Redo log buffer for recovery
- For large memory pools

Then of course, it can be used for a hash join, for the `SELECT DISTINCT` function and for aggregation (`GROUP BY`).

5.3.4 Bitmaps

Bitmap indexes are used for attributes with low cardinality (low number of distinct values), attributes that are retrieved by category and queries with low selectivity.

In addition, they are used to perform aggregation functions, such as `SUM` and to perform selection and operators over column stores.

Bitmap indexes are an alternative to B-trees and very space efficient.

Bitmaps work as follows:

- List all distinct values of an attribute
- For every distinct value (and for NULLs), create an array with 1 bit entries (i.e. 0 or 1)
- In the array corresponding to a given value of the attribute, set the positions in the array to one that correspond to tuples that have that value in that attribute.
- Now, each array indicates the tuples that have the value corresponding to that array.

5.3.4.1 Bitmaps as indexes

While a B-tree or hash index stores the row ID or tuple ID, a bitmap index only stores a single bit at a given position, which is indicating the of the tuple from the beginning of the table.

This means that to map a 1 in the beatmap to a tuple address requires applying a function that calculates the address. One way to achieve this is to layout the bitmap in page ID order. Then to map a bit position to a record id, we fix the number of records per page, lay the pages out such that the page numbers are sequential and increasing and then construct the row ID, page ID and slot number.

As a result, $\text{pageID} = \text{bit-pos} \div \text{\#records per page}$ and $\text{slot-number} = \text{bit-pos} \% \text{\#records per page}$

We can also compress bitmaps, in Postgres, this is done using run-length encoding.

5.3.4.2 Using bitmaps

Bitmaps are used in database systems to solve simple predicate selections, such as

```
SELECT NAME FROM STUDENTS WHERE Students.Department = D-INFK
```

or for multiple operations, we can perform operations directly on the bitmaps

```
SELECT NAME FROM STUDENTS WHERE Students.Department = D-INFK and Students.Degree = Bachelor
```

In addition, they are usually very sparse and long lists with just a few 1s compress very well, using run length encoding.

They are also often used for large tables in analytics as it is faster than other indexes for low selectivity queries and they are especially useful for expensive comparisons, such as for strings.

They can also be used to implement operators:

- `NOT IN`: get a bitmap with the results of the inner query, match it against a bitmap for the outer query, then retrieve the tuples
- `COUNT`: Count how many 1s appear in the bitmap
- `JOIN`: Match (`AND` operation) the bitmaps of the values of the two attributes
- `RANGE` predicates: Combine the bitmaps (`OR` operations) of all the values in the range
- ...

5.3.5 Bloom Filters

A bloom filter is an index that says whether a tuple with a given value is **NOT** in the original table. It acts as a filter, eliminating unnecessary accesses to the data. However, it can have false positives, i.e. it may state that a given tuple is there, but it is not in fact. Most importantly, it doesn't have false negatives, which is good, because this means we don't accidentally miss any tuples.

A bloom filter works by creating an array of size m , with values hashed using k hash functions. Each hash point points to a position in the array that is set to 1. Then, to check if an item is in the data, we check all the k hash positions for the value and if they all are all 1, then the item is in the data and if one (or more) is 0, then the element *definitely* isn't in the data.

The bloom filter's error rate depends on the number of hashes (k), the size of the array (m) and the number of items being hashed (n)

5.3.6 Zone maps

There are some systems that do not use indexes, or very sparingly. Instead, they keep metadata about the blocks and store them in memory and use it to check if the data in the block is needed.

This is common in column-store engines such as DuckDB and Snowflake. This metadata can be as simple as min/max for each column, or complete stats, bitmaps, precomputed aggregates, or even bloom filters.

5.3.7 AR Trees

5.3.8 R Trees

5.3.9 Other Optimizations

Glossary

AVX The Advanced Vector Extensions are an extension to the x86 architecture, to allow CPUs to execute SIMD instruction (commonly called Vector Instructions).

AVX Advanced Vector Extensions, also see **AVX**.

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

DBMS Database Management System.

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

imperative Describes how to achieve something.

record .

schema A set of relations.

SIMD Single Instruction Multiple Data.

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.