ETH Zürich

Data Modeling and Databases

Exam preparation

Author
J. Stauffer

March 14, 2023

Disclaimer

This summary is based on the lecture slides from *Data Modeling and Databases* from the spring semester 2022 as well as the *PostgreSQL* documentation. This is a personal summary and therefore I cannot guarantee completeness or correctness of the content.
Any constructive feedback is welcome and you can reach me under jan.stauffer@inf.ethz.ch

Contents

1 Terminology

- **Database** - collection of Data
- **Database Management System (DBMS)** - Software designed to maintain/utilize large collections of data
- **Workload mismatch** - Application is not what DBMS was designed for
- **Data model mismatch** - Application cannot be naturally modeled by given DBMS
- **Data Independence** - Application should not know how data is stored
- **Declarative Efficient Data Access** - The system should be able to store and retrieve data efficiently, without users worrying about it (too much)
- **Transactional Access** - Abstraction of single user operating an infallible system
- **Generic Abstraction** - Users don't need to worry about DBMS problems for each query
- **Declarative** - Describing what
- **Imperative** - Describing how
- **Information Management System (IMS)** - A hierarchical data model by IBM
- **Network Model** - A data model organized as a network of record types
- **Relational Model** - A data model organized as a set relations
- **Database Schema** - Set of Relation Schema
- **Relation Schema** - Name and a set of Attributes/Fields
- **Field/Attribute** - Name and Domain
- **Candidate Key** - minimal set of fields that identify each tuple uniquely
- **Primary Key** - one candidate key (marked in Schema by underlining)
- **Relational Algebra** - imperative way to query relational databases using set operations
- **Relational Calculus** - declarative way to query relational databases using first-order logic
- **Safe Query** - A query in relational calculus that returns a finite result under every instance
- **Unsafe Query** - Not a safe query
- **SPJR Algebra** - Relational algebra with only selection, projection, join and renaming

2 Introduction

A **database** is a collection of facts or data. To manage and utilize this database, there are special applications called **database management systems (DBMS)**. When thinking of a simplistic application to handle large collections of data a few problems come to mind, which the application has to handle.

- Handle data that is bigger than memory size.
- Retain data after loss of power.
- Implementation should be invariant to way data is stored.
- Query should be executed in optimal way.
- I/O operations should be minimized.
- Support concurrent access.
- Ability to recover after crash.
- Generic way to express queries.

A DBMS tries to solve all these challenges, which leads us to a few principles.
Data Independence. The applications should not know how the data is stored. This means queries do not have to be changed when data is reorganized.
Declarative Efficient Data Access. The system should be able to store and retrieve data efficiently, without users worrying about it too much. By describing data access in a declarative manner (what and not how), we do not constrain the system to a specific way of execution. Therefore, the system is able to find an optimal execution which allows for better performance.
Transactional Access. The system should provide an abstraction of a single user utilizing an infallible system. This allows for concurrent access and crash recovery.
Generic Abstraction. The user should not worry about above issues for every query. The system should provide a query language, in which data access can be expressed.

2.1 Information Management System (IMS)

IMS is a hierarchical data model that organizes record types as a tree. The query language retrieves one record at a time by traversing the tree.
IMS does not provide data independence or declarative efficient data access. This means the queries have to be altered on changes to the physical data representation and queries require manual optimization. Additionally, data not matching the shape of a tree will result in data redundancy. Redundancy is difficult to handle as it is a root cause of inconsistency.
However, there are applications in which above criteria are not crucial and IMS is a powerful tool.

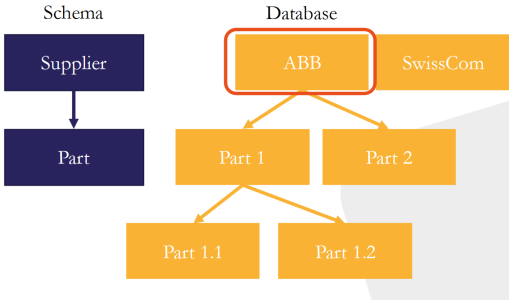
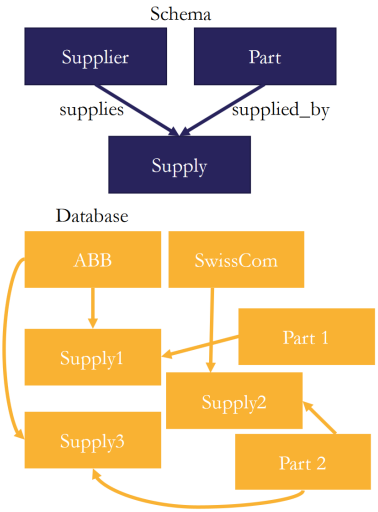


Figure 1: IMS Database Schema/Instance

2.2 Network Model

The network model is a data model that organizes its record types not as a tree but as a network instead. Similar to IMS, the query language retrieves one record at a time by doing a traversal on the network, however, they are more natural compared to the hierarchical ones.
The network model does not provide data independence or declarative efficient data access, which means it has the same problems as IMS.



(a) Network Model Database Schema/Instance

2.3 Relational Model

The amount of work required to maintain data models that do not offer data independence inspired the relational model. It models the data as a set of relations, where each relation is a set of tuples of the same type. A database scheme is a set of relations and a database instance is a set of instances for each relation. Contrary to the record-at-a-time query languages of IMS and the network model the relational model provides a set-at-a-time query language with support for set operations.
The relational model offers both data independence as well as declarative efficient data access. This means the queries do not change on alteration to the data representation since they are declarative and not imperative. Furthermore, this enables automatic query optimization to choose the best execution plan.

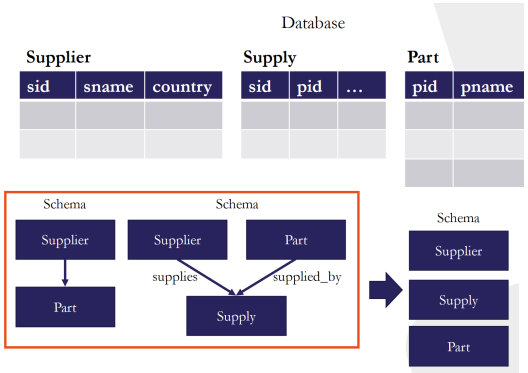


Figure 3: Relational Model Database Schema/Instance

3 Relational Model, Relational Algebra

The Relational Model represents data as **a collection of facts** and inference can be done using **mathematical logic**. It represents data in the following way:

- **Database Schema** - a set of relation schema
- **Relation Schema**
  - Name

- A Set of Attributes/Fields/Columns

- **Field/Attribute**
  - Name
  - Domain (e.g., Integer, String)

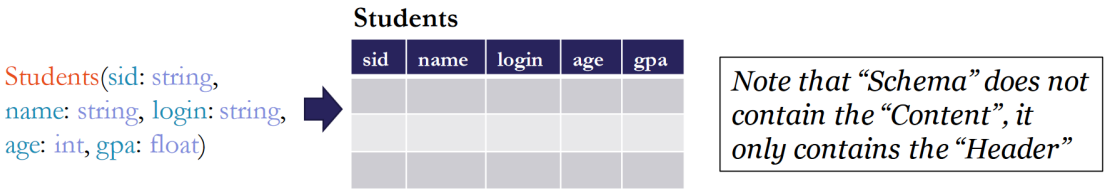


Figure 4: Example of a Relation Schema

Because a relation instance is a **set**, tuples in the relation are unique and their order does not matter. Real databases might follow a **bag semantic**, which allows for duplicate tuples. This lecture however will assume **set semantics** for the relational model.

3.1 Candidate & Primary Key

Candidate Key: The candidate key is a minimal set of fields that identify each tuple uniquely
Primary Key: The primary key is a chosen candidate key, which is marked in the relation scheme by underlining

Every relation must have a primary key. Since all relations are sets, it is always possible to find such a key. The **primary key semantics** are a way to constrain the set of *valid instances*.
From a formal point of view; given a relation R

$$R(k : D_k, a : D_a, b : D_b)$$

with syntax $(attribute\ name : attribute\ domain)$.
The set of valid instances I is:

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

3.2 Relational Algebra

One way to query a database modeled in the relational model is called **relational algebra**. Since relational instances are **sets** they can be queried **imperatively** using **set operations**.

- **Union:** $\cup$
  $x \in R_1 \cup R_2 \iff x \in R_1 \vee x \in R_2$
  Returns tuples that are in either of the two relations.
  Union can only be applied if both relations have the same attributes. Since a relation has a **set of attributes** their order doesn't matter in relational algebra. This means $R(pid, name) \cup S(name, pid)$ is a valid union.

- **Difference:** $-$
  $x \in R_1 - R_2 \iff x \in R_1 \wedge \neg(x \in R_2)$
  Returns tuples that are in the first but not in the second relation.
  Similar to union, difference can only be applied if both relations share the same attributes.

- **Intersection:** $\cap$
  $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
  Returns tuples that are in both relations.

- **Selection:** σ
  $x \in \sigma_c(R) \iff x \in R \wedge c(x) = True$ where c is a predicate on R.
  Returns tuples that satisfy a given condition.

- **Projection:** $\Pi_{A_1,...,A_n}(R)$
  Returns only the given attributes of a relation.

- **Cartesian Product:** $\times$
  $(x, y) \in R_1 \times R_2 \iff x \in R_1 \wedge y \in R_2$
  Returns all possible combinations of elements in the first and second relation.

- **Renaming:** $\rho_{B_1,...,B_n}(R)$
  Changes the name of the attributes of R to $B_1, ..., B_n$.

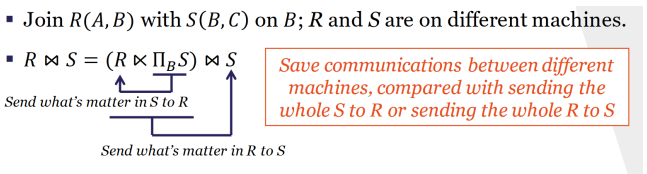
- **Natural Join:** $\bowtie$
  $R_1(A, B) \bowtie R_2(B, C) = \Pi_{A,B,C}(\sigma_{R_1.B=R_2.B}(R_1 \times R_2))$
  Step 1 Selection - check equality on all common attributes
  Step 2 Projection - eliminate duplicated common attributes
  Corner Cases:
  - No shared attributes: $R \bowtie S = R \times S$
  - Share all attributes: $R \bowtie S = R \cap S$

- **Theta Join:** $\bowtie_\theta$
  $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$
  Joins two relations on a given condition θ.

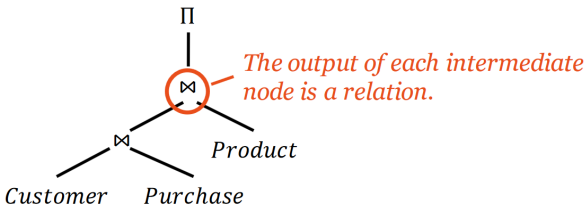
- **Equi-Join:** $\bowtie_{A=B}$
  $R_1 \bowtie_{A=B} R_2 = \sigma_{A=B}(R_1 \times R_2)$
  Joins two relations on equivalence of two attributes.

- **Semi-Join:** $\ltimes_c$
  $R(A_1, ..., A_n), R_2(B_1, ..., B_n)$
  $R_1 \ltimes_c R_2 = \Pi_{A_1,...,A_n}(R_1 \bowtie_c R_2)$
  Useful in a trick called *semi-join reduction* for distributed databases.

- *Relational Division:* $\div$
  $R \div S = \Pi_{R-S}R - \Pi_{R-S}((\Pi_{R-S}R) \times S - R)$
  $R \div S = T$ where T is the *largest* relation such that $S \times T \subseteq R$.
  Only included for completeness, I hate this thing.



(a) Semi-Join Reduction



(b) Relational Algebra Expression

It is possible to compose multiple operations together to form a relational algebra expression. Since relational algebra is not declarative there are different ways to implement the same query. In real world database systems the user does not write relational algebra statements. We hope to have a query language in which the user does not need to worry about different ways of implementing the same thing - we want the system to figure out the optimal way by itself.

One core limitation of relational algebra is the inability to express **recursion**.

3.3 Relational Calculus

Relational calculus takes a different approach than relational algebra. Instead of viewing relations as sets, it considers relations to be **facts** and queries them using **first-order logic**.

First-order logic consists of **syntax** and **semantics**. The syntax determines which finite sequences of symbols are well-formed expressions in first-order logic, while the semantics determines the meanings behind these expressions.

3.3.1 First-Order Logic - Syntax

The **alphabet**, i.e. the collection of all legal symbols, is usually divided into **logical symbols** and **non-logical symbols**.
The logical symbols include the universal and existential quantifiers $\forall$ and $\exists$, the symbols for conjunction $\wedge$, disjunction $\vee$, implication $\implies$, biconditional $\leftrightarrow$, negation $\neg$ as well as parentheses, brackets, other punctuation symbols, an infinite set of variables, equality symbol $=$ and truth constants $\top$ and $\bot$.
The non-logical symbols represent predicates (relations), functions and constants.
The formation rules define the terms and formulas of first-order logic. **Terms** are Variables and Functions (where each argument is a term). A **formula** can be formed by using predicate symbols, equality, negation, binary connectives and quantifiers.
A variable can occur either free or bound. A variable is free, if it is not bound to a quantifier.

3.3.2 First-Order Logic - Semantics

The truth value of a formula depends on the *world we live in.* This *world* is semantically defined by the **interpretation** $\mathbb{I}$. The interpretation determines the **domain** $\mathbb{D}$, which specifies the range of the quantifiers, and and assigns a set of true tuples of elements to each predicate.
A **model** $\mathcal{M}$ is the tuple $(\mathbb{I}, \mathbb{D})$. We write

$$\mathcal{M} \models \phi \iff \phi \text{ evaluates to True under } \mathcal{M}$$
$$\mathcal{M} \not\models \phi \iff \phi \text{ evaluates to False under } \mathcal{M}$$

If there are free variables occurring in ϕ the truth value depends on the value of the free variables. An **assignment** maps all free variables to concrete constants. Similarly we write

$$\mathcal{M}, \alpha \models \phi \iff \phi \text{ evaluates to True under } \mathcal{M} \text{ and } \alpha$$
$$\mathcal{M}, \alpha \not\models \phi \iff \phi \text{ evaluates to False under } \mathcal{M} \text{ and } \alpha$$

3.3.3 Domain Relational Calculus

Queries in relational calculus can be expressed in a declarative way. It tells the system what we want instead of how to get it. One expresses the query as a set and all tuples belonging to the set are the answer to the query.

$$\{(pid, cid) \mid \exists n, p(Product(pid, n, p)) \wedge \exists cn, c(Customer(cid, cn, c)) \wedge \exists s(Purchase(pid, cid, s))\}$$

A formal definition for **(Domain) Relational Calculus.**

- Schema
  - Database Schema: $S = (R_1, ..., R_m)$ where each R_i is a Relation.
  - Relation Schema: $R(A_1 : D_1, ..., A_n : D_n)$
  - Domain: $dom = \cup_i D_i$ – infinite set of constants
- Instance
  - Instance of $R(A_1 : D_1, ..., A_n : D_n) : I_R \subseteq dom^n$ and I_R is finite
  - Instance of DB $S(R_1, ..., R_m) : \mathbb{I}$ a function that maps R_i to an instance of R_i
- An instance of a relation is a finite set of facts over the relation
- An instance of DB is a finite set of facts over all relations.

Since our domain is infinite it is possible to construct queries in relational calculus that return an **infinite result**. However, we don't want our database to output infinite answers. Therefore, we classify queries into **safe queries** and **unsafe queries**. Let Q_ϕ be a relational calculus query. Q_ϕ is considered to be **safe**, if the result $Q_\phi(\mathbb{I})$ is finite for all instances $\mathbb{I}$.

$$\{x \mid \neg R(x)\}$$
$$\{x \mid \exists y.R(x) \vee R(y)\}$$
$$\{y \mid \exists x.R(x)\}$$
$$\{x \mid R(x) \vee \neg R(x)\}$$

Figure 6: Examples of unsafe queries

Unfortunately, the problem of deciding whether a query is safe or not is undecidable, since it can be reduced to the satisfiability problem. To get rid of this problem we try to adapt the semantics.

3.3.4 Domain Independent Relational Calculus

A query where the answer depends not only on the interpretation of relations (i.e. the database instance) but also on the domain is called **domain dependent**. We hope to turn a relational calculus query Q_ϕ into something domain independent. We define the active domain;

$$adom(Q_\phi, \mathbb{I}) = \text{ all constants in } Q_\phi \text{ and } \mathbb{I}$$

From there we define the active domain semantics for Q_ϕ:

$$Q_{adom(\phi,\mathbb{I})} = \{(x_1, ..., x_n) \mid \phi \wedge \forall i, x_i \in adom(\phi, \mathbb{I})\}$$

Additionally, all quantifiers $\forall v_i,\ \exists v_i$ in Q_ϕ become $\forall v_i \in adom(\phi, \mathbb{I})$, $\exists v_i \in adom(\phi, \mathbb{I})$. Since both ϕ and $\mathbb{I}$ are both finite, a query in relational calculus under active domain semantics will always return a finite answer.

From a theory side, relational calculus is more powerful than relational algebra, since there are queries that cannot be expressed in relational algebra but can in relational calculus. But Codd's Theorem proves that domain-independent relational calculus is as powerful as relational algebra.
Still domain independent relational calculus is still tremendously difficult to analyse.

- Given a DI-RC query ϕ, does there exist a DB instance such that it outputs non-empty answers? - **Satisfiability, undecidable!**
- Given two DI-RC queries ϕ_1, ϕ_2, are they equivalent under all possible instances? - **Equivalence, undecidable!**

3.3.5 Conjunctive Query

To make the aforementioned properties easier to check we can constrain our query language. A conjunctive query has following form:

$$\phi = \exists y_1, ..., y_l(A_1 \wedge ... \wedge A_m),\ Q_\phi = \{(x_1, ..., x_n)|\phi\} \text{ each } A_j \text{ is an atom.}$$

Conjective query is as expressive as relational algebra with only selection, projectoin, join and renaming (SPJR Algebra). A one-to-one correspondence in structure can be established

SPJR Algebra	Conjunctive Query
• R	• $\{x \mid R(x)\}$
• $\sigma_c(E_1)$	• $\{x \mid \phi_{E_1}(x) \wedge \phi_c(x)\}$
• $\Pi_{A_1,...,A_n}(E_1)$	• $\{(x_1, ..., x_n) \mid \exists o_1, ..., o_k.\ \phi_{E_1}(x_1, ..., x_n, o_1, ..., o_k)\}$
• $E_1 \times E_2$	• $\{(x, y) \mid \phi_{E_1}(x) \wedge \phi_{E_2}(y)\}$
• $\rho_{a_1,...,a_n}(E_1)$	

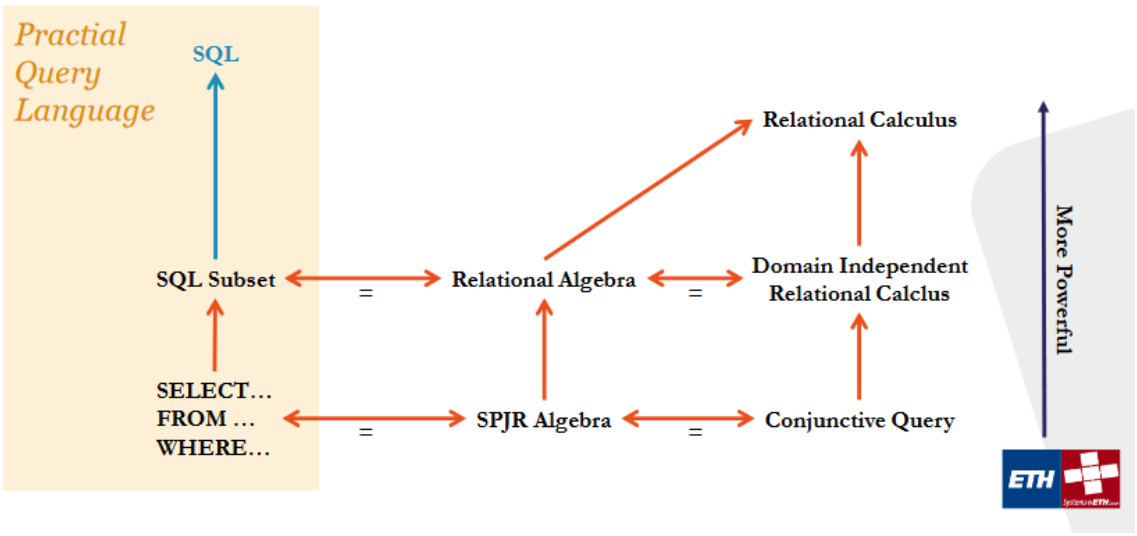


Figure 7: Expressiveness of Languages in Comparison

4 Structured Query Language

SQL is a family of standards consisting of

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Query Language

It is important to note that contrary to relational calculus and relational algebra SQL relations **have bag semantics.**

4.1 Data Definition Language (DDL)

The data definition language provides statements to define the relation schema. In the relational model a relation schema looks like $R(f_1 : D_1, ..., f_n : D_n)$. In SQL the schema is defined by providing the **relation name, a set of attributes and their types**.

4.1.1 Table Basics

A table in a relational database is defined by it's name and columns. The number, type and order of columns is fixed and each column has a name. The data type of the column constrains the set of possible values that can be assigned to a column.
A table is created by the *CREATE TABLE* command. In this command one specifies the name of the table, the name of the columns and their respective datatype. For example;

```
CREATE TABLE products (
        pid integer,
        name text,
        price numeric
);
```

Similarly a table can be deleted using the *DROP TABLE* command.

```
DROP TABLE products;
```

4.1.2 Default Values

A column can be assigned a default value. When a new row is created and no values are specified for some columns, those columns will be filled with their respective default values. If no default value is declared specifically, the default value is the null value.

```
CREATE TABLE products (
        pid SERIAL,
        name text,
        price numeric DEFAULT 420.69,
        buy_time timestamp CURRENT_TIMESTAMP
);
```

Common default values are *SERIAL*, which is a shortcut to supply successive values or *CURRENT_TIMESTAMP* to provide the timestamp at the time of insertion.

4.1.3 Constraints

A **primary key constraint** indicates that a column, or a group of columns, can be used as a unique identifier for rows in the table. This requires that the values are both **unique and not null**.

```
CREATE TABLE products (
        pid integer PRIMARY KEY,
        name text,
        price numeric,
        buy_time timestamp
);
```

```
CREATE TABLE example (
        a integer,
        b integer,
        c integer,
        PRIMARY KEY (a, c)
);
```

4.1.4 Modifying Tables

When requirements of the application change or mistakes have been made it is very easy to drop a table and create it again. However, this is not a convenient option if the table is already filled with data, or if the table is referenced by other database objects (for instance a foreign

key constraint). Postgresql provides a family of commands to make modifications to existing tables. This is conceptually different to altering data contained in the table; here we alter the definition of the relation schema. All operations are performed using the *ALTER TABLE* command.

- **Add/Remove Columns**

```
ALTER TABLE products ADD COLUMN description text;
ALTER TABLE products DROP COLUMN description;
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
ALTER TABLE products DROP COLUMN description CASCADE;
```

When creating a new column, it is initially filled with whatever default value is given. If no default value is specified the column is filled with null. Additionally, all options for columns can also be applied (i.e. default values, constraints).

- **Add/Remove Constraints**

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES
    product_groups;
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

To add a not-null constraint, which cannot be written as a table constraint, the above syntax is used. This constraint will be checked immediately, so the table data must satisfy the constraint before it can be added.
To remove a constraint you need to know its name. If you know the name it is easy. Otherwise the system assigned a generated name, which can be found out with
d tablename.As with dropping a column, you need to add *CASCADE* if you want to drop a constraint that something else depends on. An example is that a foreign key constraint depends on a unique or primary key constraint on the referenced columns. It works the same for all constraint types except not-null constraints. To drop a not-null constraint the syntax below can be used.

```
ALTER TABLE products DROP CONSTRAINT some_name;
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

- **Change/Remove Default Values**

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 6.9;
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

Changing the default value does not affect any existing tuples in the table, it just changes the default for future *INSERT* commands. Removing a default value is essentially the same as setting it to the null value. Dropping a default that does not exist hence does not produce an error.

- **Change Column Data Types**

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

Changing a columns data type will only succeed if the existing entries can be converted to the new type by an implicit cast. For more complex casts the *USING* clause can be used to specify how to compute the new values. Postgres will try to convert the column's default values and constraints that involve the column to the new type. These conversions might fail or produce surprising results. It is often best to drop any constraints on the column before conversion.

- **Rename Columns/Tables**

```
ALTER TABLE products RENAME COLUMN product_no TO pid;
ALTER TABLE products RENAME TO items;
```

4.2 Data Manipulation Language (DML)

4.2.1 Inserting Data

A new tuple can be created using the *INSERT* command. The command requires the table name and the column values.

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

The data values are listed in the order in which the columns are defined in the table, separated by commas. The data values are either literals or scalar expressions. To avoid the need to know the order of the columns, the columns can be listed explicitly.

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
```

If not all columns of the table are stated, the columns are filled from the left with as many values as are given and the rest of the columns assumes their default value. It is also possible to request the default value explicitly using the *DEFAULT* keyword. It is also possible to insert multiple rows in a single statement or insert the result of a query, which may return one or several tuples.

```
INSERT INTO products (product_no, name, price) VALUES
        (1, 'Cheese', 9.99),
        (2, 'Bread', 1.99);
INSERT INTO products (product_no, name, price)
        SELECT product_no, name, price FROM new_products
                WHERE release_date = 'today';
```

4.2.2 Updating Data

Existing rows can be updated via the *UPDATE* command, which requires three pieces of information.

1. The name of the table and column to be updated
2. The new value of the column
3. Which rows to update

Since SQL does generally not provide a unique identifier for each row, rows are identified by whether they specify a given condition. Only if there is a primary key, specified or not, individual rows can be addressed. An update command will try to update all rows that fulfil the given condition, which may be zero, one or multiple rows. It does not produce an error to attempt an update that does not match any rows. If multiple columns need to be updated they are separated by commas.

```
UPDATE products SET price = 10 where price = 5;
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

4.2.3 Deleting Data

Similarly to updating rows, rows are deleted based on whether they match a given condition, since generally rows cannot be addressed individually. Rows are deleted using the adequately named **DELETE** command.

```
DELETE FROM products WHERE price = 10;
```

Simply stating a table name will delete all rows in said table!

```
DELETE FROM products;
```

4.2.4 Returning Data from Modified Rows

Sometimes it is useful to obtain data from modified rows while they are being manipulated. The *INSERT*, *UPDATE*, and *DELETE* commands all have an optional *RETURNING* clause supporting this. The allowed contents of a *RETURNING* clause are the same as a *SELECT* command's output list. It can contain column names, or value expressions using those columns. In an *INSERT* command the rows are returned as they were inserted. In an *UPDATE* statement, the returned data is the content of the modified row. In a *DELETE* command the content of the deleted rows is returned.

```
CREATE TABLE users (firstname text, lastname text, id SERIAL PRIMARY KEY);
INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cole') RETURNING id;

UPDATE products SET price = price * 1.1
        WHERE price <= 99.99
        RETURNING name, price AS new_price
DELTE FROM products WHERE obsoletion_date = 'today'
        RETURNING *;
```

4.2.5 Snapshot Semantics

How does SQL deal with this?

```
DELETE FROM requires WHERE prerequisite IN
(SELECT follow-up FROM requires);
```

SQL conducts the *DELETE* in two phases.

- **Phase 1:** Mark all tuples which are affected by the updates.
- **Phase 2:** Implement updates on the marked tuples.

Mixing the two phases would lead to non-deterministic results.

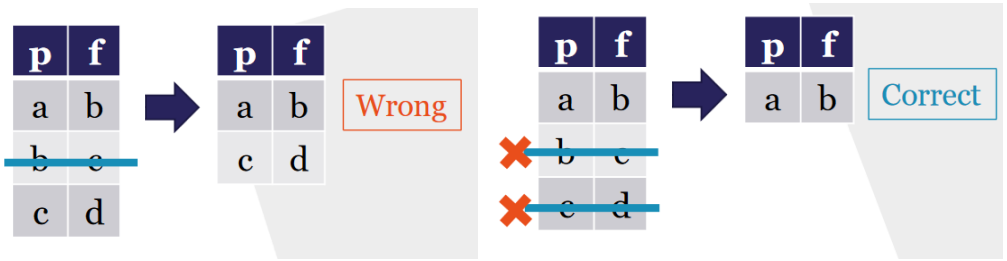


Figure 8: Incorrect and Correct Outcome

4.3 Query Language

In SQL the *SELECT* command is used to specify queries. The general syntax of the *SELECT* command is;

```
[WITH with_queries] SELECT select_list FROM table_expression [
    sort_specification]
```

4.3.1 Table Expressions

A table expression computes a table. The table expression contains a *FROM* clause that is optionally followed by *WHERE*, *GROUP BY,* and *HAVING* clauses.

- **The FROM Clause** The *FROM* clause derives a table from one or more other tables given in a comma-separated table reference list.

  ```
  FROM table_reference [, table_reference [, ...]]
  ```

  A table reference can be a table name or a derived table such as a subquery, a *JOIN* construct, or a complex combination of these. **If more than one table reference is listed in the *FROM* clause, the tables are cross-joined**.

- **Joined Tables** A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available. The general syntax is:

  ```
  T1 join_type T2 [ join_condition ]
  ```

  Joins of all types can be chained together, or nested. Either or both T1 and T2 can be joined tables. Parentheses can be used around join clauses to control the join order. In absence of parentheses the order is left-to-right.

  Cross Join

  ```
  T1 CROSS JOIN T2
  ```

  The cross join is a Cartesian product. It takes all possible combinations of rows from T1 and T2 and appends them in the joined table.

  Qualified Joins

  ```
  T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
  T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column
      list )
  T1 NATURAL { [INNER] | {LEFT | RIGHT | FULL } [OUTER] } JOIN T2
  ```

  Above is the general syntax of a qualified join. *INNER* and *OUTER* are optional in all forms. The default is *INNER* and *LEFT, RIGHT,* and *FULL* imply an outer join. The join condition is specified in the *ON* or *USING* clause, or implicitly by *NATURAL.*

  Inner Join
  Logically, a cross join is performed and each pair of rows that does not satisfy the condition is removed.

  Left Outer Join
  An inner join is performed. Then for each row in T1 that does not match the condition with any row from T2 (,i.e. is not in the joined table of the inner join), a joined row is added with null values for the columns in T2.

Right Outer Join
Same principle as a left outer join but the tables are reversed. For each row in T2 that does not match the condition with any row from T1, a joined row is added with null values for the columns in T1.

Full Outer Join
Same principle as left/right outer join but joined rows are added for rows from both tables that do not match the condition.

ON Clause
The *ON* clause takes a boolean expression of the same kind that is used in a *WHERE* clause. A pair of rows matches the *ON* clause if the expression evaluates to true.

USING Clause
The *USING* clause is a shorthand for an equality check on columns. It takes a comma-separated list of shared column names and creates a join condition that includes an equality comparison for each one. The following join conditions are the same.

```
T1 JOIN T2 USING (a, b);
T1 JOIN T2 ON T1.a = T2.a AND T1.b = T2.b;
```

Additionally, the output of *JOIN USING* suppresses redundant columns.

Natural
NATURAL is a shorthand form of *USING* that includes all column names that appear in both tables. If there are no common columns it behaves like a cross-join.

- **Table And Column Aliases**
A temporary name can be given to tables and complex table references to be used in the rest of the query. This is called a table alias. A table alias can be created using the following syntax.

```
FROM table_reference AS alias
FROM table_reference alias
```

A common application is to assign a short identifier to long table names to keep clauses readable. The alias becomes the new name of the table and the **original name cannot be used in the query.** Aliases are necessary in self-joins and if the table reference is a subquery.
It is also possible to alias the columns of the table as well as the table itself. If fewer column aliases are supplied than the table has columns, the remaining columns are not named.

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

- **Subqueries**
Subqueries specifying a derived table must be enclosed in parentheses and must be assigned a table alias.

```
FROM (SELECT * FROM table1) AS alias_name
```

- **The WHERE Clause**
After processing of the *FROM* clause, each row of the derived table is checked against the search condition. If the result of the condition is true, the row is kept in the output table.

- **The GROUP BY and HAVING Clauses**
After passing the *WHERE* filter, the derived input table can be grouped using *GROUP BY* and groups can be eliminated using the *HAVING* clause.

```
SELECT select_list
       FROM ...
```

```
        [WHERE ...]
        GROUP BY grouping_column_reference [, grouping column_reference]...
        HAVING boolean_expression
```

4.3.2 Distinct

Duplicate results in the output can be eliminated using the *DISTINCT* key word. It is written directly after the *SELECT*. The key word *ALL* can be used to specify the default behaviour of retaining all rows. Null values are considered equal in comparison. An arbitrary condition of equality can be specified using *DISTINCT ON*.

```
SELECT DISTINCT select_list ...
SELECT ALL select_list ... %default behaviour
SELECT DISTINCT ON (expression) select_list ...
```

4.3.3 Combining Queries

The result of two queries can be combined using the set operations union, intersection and difference. The syntax is

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query EXCEPT [ALL] query2
```

To use any of the three operations, the two queries must be "union compatible", which means that they return the same number of columns and the corresponding columns have compatible data types.

- **Union**
  UNION appends all results of query2 to the result of query1. Additionally, it **eliminates duplicates** unless *UNION ALL* is used.

- **Intersect**
  INTERSECT returns all rows that are both in the result of query1 and query2. Duplicate rows are eliminated unless *INTERSECT ALL* is used.

- **Except**
  EXCEPT returns all rows that are in the result of query1 but not in the result of query2. Duplicate rows are eliminated unless *EXCEPT ALL* is used.

4.3.4 Sorting Rows

A query will return the rows in an arbitrary order. With the *ORDER BY* clause it is possible to specify how the rows should be ordered. The sort expression(s) can be any expression that would be valid in the query's select list. When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values.

```
SELECT select_list FROM table_expression
       ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
```

The key words *ASC* and *DESC* determine whether the rows are sorted ascending or descending, with ascending being the default. *NULLS FIRST/LAST* options can be used to choose whether null values should appear before or after non-null values in the order. *NULLS FIRST* is the default for *DESC* order, and *NULLS LAST* otherwise.

4.3.5 Subquery Expressions

- **Exists** The arguments of *EXISTS* is an arbitrary *SELECT* statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of *EXISTS* is true. If the subquery returns no rows, the result is false. The subquery is usually only run until the result is clear, therefore no queries with side effects should be used.

```
EXISTS (subquery)
```

- **In**

```
expression IN (subquery)
```

  The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of *IN* is true if any equal subquery row is found. The result is false if no matching row is found.
  If the left-hand expression returns null or if there are no equal right-hand values and at least one right-hand row is null the result of the *IN* construct is null not false.

- **Not In**

```
expression NOT IN (subquery)
```

  Analogous to *IN*. The result of *NOT IN* is true if only unequal subquery rows are found and false if an equal row is found.

- **Any/Some**

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

  The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a boolean result. The result of *ANY* is true if any true result is obtained and false otherwise. *SOME* is a synonym for *ANY*.

- **All**

```
expression operator ALL (subquery)
```

  The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a boolean result. The result of *ALL* is true if all rows yield true. The result is false if any false results are found. The result is null if no comparison with a subquery returns false, and at least one comparison returns null.

5 Known, Unknowns, and Views

So far we have assumed that all information in a relation is complete, but in practice we often have incomplete information in our database. This can happen if something is not applicable or the data is simply missing.

5.1 Null

One way to model incomplete information is to replace the values that we do not know with a special state **null**. It is important to mention that *NULL* is a state, not a value. A null value introduces a lot of corner cases for many operations. A few are listed below;

- Arithmetic
  - $(NULL + 2) \rightarrow NULL$
  - $(NULL * 0) \rightarrow NULL$
- Comparisons
  - $NULL = NULL \rightarrow Unknown$
  - $(NULL < 13) \rightarrow Unknown$
  - $(NULL > NULL) \rightarrow Unknown$
- 3-value Logic

not	
true	false
unknown	unknown
false	true

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

- State not Value
  - $(NULL = NULL) \rightarrow Unknown$
  - $(NULL\ IS\ NULL) \rightarrow Unknown$
- Null in Aggregations
  - Null in *GROUP BY* - all nulls form a single group
  - Null in Values - Most operators will ignore the NULL value (count(*) will not)

Null can come appear in various contexts. One being *INSERT* commands where for some columns no value is supplied. Another example are outer joins.

A	B	C
a1	b1	c1
a2	b2	c2

⋈

C	D	E
c1	d1	e1
c3	d2	e2

=

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	NULL	NULL

Figure 9: Left Outer Join Introducing Null

5.1.1 View

A relation is a powerful abstraction that provides us with *physical data independence*. A view aims at raising the level of abstraction even higher, and provide us with *logical data independence*, i.e. the user can use view without caring about how relations are organized underneath the cover.
In a database, a view is the result set of a stored query on the data, which the database users can query just as they would in a persistent database collection object.

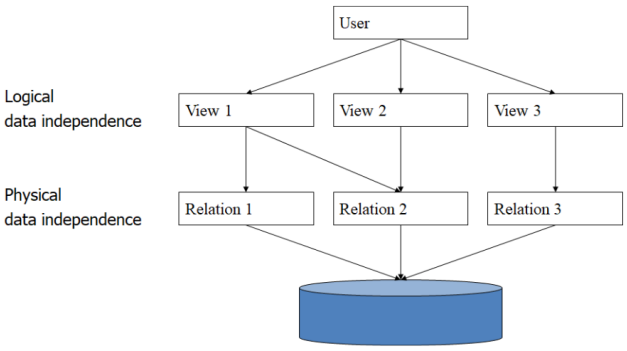


Figure 10: Logical and Physical Data Independence

A view can be created using the *CREATE* command.

```
CREATE VIEW name_of_view AS (subuery);
```

Then the view can be used just like a table, even if it logically has never been materialized. Views have several different applications:

- **Privacy**
  A view can be used for access control and preserving privacy. A user who has access to a relation, knows all the information in said table. A user who only has access to the view, only knows the information that is present in the view.
- **Usability**
  A view can be used to make writing some queries easier. It is essentially like a named subquery. It may not look like a big saving but in practice the saving can be significant. Views make the queries a lot more readable.
- **Is-a-Relationship**
  A view can be used to implement generalization.

```
CREATE VIEW Employee AS (
SELECT id, name
FROM Professor
UNION
SELECT id, name
FROM Assistant
UNION
SELECT id, name
FROM OtherEmps
);
```

```
CREATE TABLE Professor
(id integer not null,
name varchar(30) not null,
room integer);
```

```
CREATE TABLE Assistant
(id integer not null,
name varchar(30) not null,
TACourse integer);
```

```
CREATE TABLE OtherEmps
(id integer not null,
name varchar(30) not null);
```

Figure 11: Generalization using View

A view gets evaluated by query rewriting. First the view is replaced by its defining query and then optimized by the database management system.

Is it possible to update a view? First we have to define what that means exactly. Assuming there are base relations $R_1, ..., R_n$ and a view $V = Q(R_1, ..., R_n)$ defined over these relations. If the user wants to update V into V', the task is to find a set of updates to the base relation $R'_1, ..., R'_n$ such that $Q(R'_1, ..., R'_n) = V'$. It is not always possible to find such updates. Following are a few examples of views, that are not updateable.

Apply

sID	cName	major	decision
1	S	CS	Y
1	S	EE	N
1	B	CS	Y
1	C	EE	Y
2	B	BIO	N
3	M	BIOE	Y
3	C	BIOE	N
3	S	CS	Y
3	C	EE	N

```
CREATE VIEW CSaccept AS
SELECT sID
FROM Apply
WHERE major = 'CS'
and decision = 'Y';
```

```
INSERT INTO Csaccept
VALUES (2);
```

Problem? Don't know what to fill in in the original table.

(a) Non updatable Projection View

R

r
1
2
3
4
5
6
7
8
9

```
CREATE VIEW V (v) AS
SELECT SUM(r)
FROM R;
```

```
UPDATE V
SET v = v + 1;
```

Problem? Updates on the base relations are not unique.

(b) Non updatable Aggregation View

But it is not all bad news. There also exist views that are easier to update.

R

r	c
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H
9	I

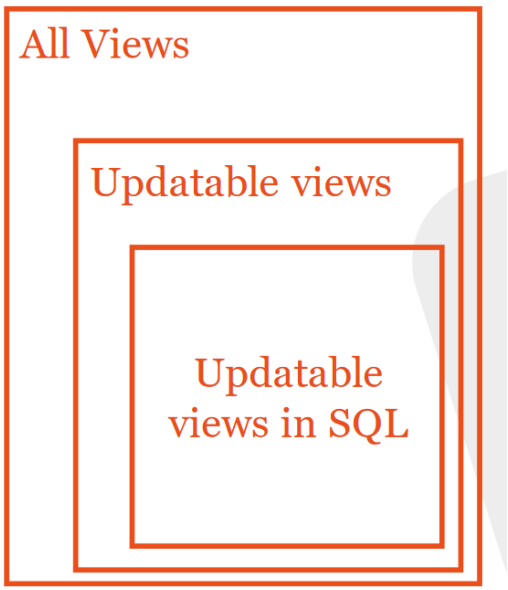
```
CREATE VIEW V (v) AS
SELECT r
FROM R;
```

```
DELETE FROM V
WHERE r = 5;
```

Can find a unique tuple in the original table to delete (r is the key).

Figure 13: Updatable View

- Principle:
  - SQL tries to avid indeterminism.
  - SQL is conservative with view updates.
- A SQL view is updatable if and only if:
  - The view involves only ONE base relation.
  - The view involves the key of that base relation.
  - The view does NOT involve aggregates, group by, or duplicate-elimination.
- What's the intuition of these three rules?
  - Try to maintain one-to-one mapping between the view and the base relation



(a) Hierarchy of Views

6 Recursion and Integrity Constraints

6.1 Recursion

Recursion provides the functionality to execute the same query again and again until it converges.

```
WITH RECURSIVE R as
(base query)
UNION
(recursive query)
<Query involving R and other tables>
```

Recursion can be tricky and lead to non terminating queries. Since recursion is mostly used to determine the transitive closure over graphs, it makes sense to look into graph databases when the workload contains a lot of recursion queries.

6.2 Integrity Constraints

Constraints define the set of valid database instances.

6.2.1 Not Null Constraint

The values of a certain attribute cannot be null.

```
CREATE TABLE student (
        sID int,
        sName text,
        GPA real not null);
```

6.2.2 Primary Key Constraint

The attributes need to be unique and not null.

```
CREATE TABLE student (
        sID int PRIMARY KEY,
        sName text,
        GPA real);
```

A primary key can be defined over multiple attributes but there can only be one primary key per relation. The combination needs to be unique and no attribute can be null.

```
CREATE TABLE students (
        sID int,
        sName text,
        GPA real
        PRIMARY KEY (sID, sName));
```

6.2.3 Unique Constraint

A certain attribute needs to be unique. It is possible to define multiple unique constraints for one relation, which can be over one or several attributes. Unique columns may have multiple null rows.

```
CREATE TABLE student (
        sID int UNIQUE,
        sName text,
        GPA real
        UNIQUE (sID, sName)
        UNIQUE (sID, GPA));
```

6.2.4 Check Constraint

CHECK constraints can be used to conduct local value checking of attributes. Check constraints only reject false tuples, unknown and true are accepted.

```
CREATE TABLE student (
        sID int,
        sName text,
        semester int,
        GPA real
        check(GPA + semester = 10));
```

6.2.5 Referential Constraint

Foreign keys refer to tuples from a different relation. The attribute cannot take a value that does not exist in the referenced table or it has to be null.

```
CREATE TABLE R (
a integer PRIMARY KEY, b varchar(3) UNIQUE);

CREATE TABLE S (...,
        ka integer references R);
```

6.2.6 Referential Constraints: Maintenance

What happens if we delete a row that is referenced in another table? This is problematic because referenced values have to exist. There are different options to handle this case;

- **Cascade**
  Propagate update or delete
- **Restrict**
  Prevent deletion of the primary key before trying to do the change, cause an error
- **No Action**
  Prevent modifications after attempting the change, cause an error
- **Set default, set null**
  Set references to null or a default value.

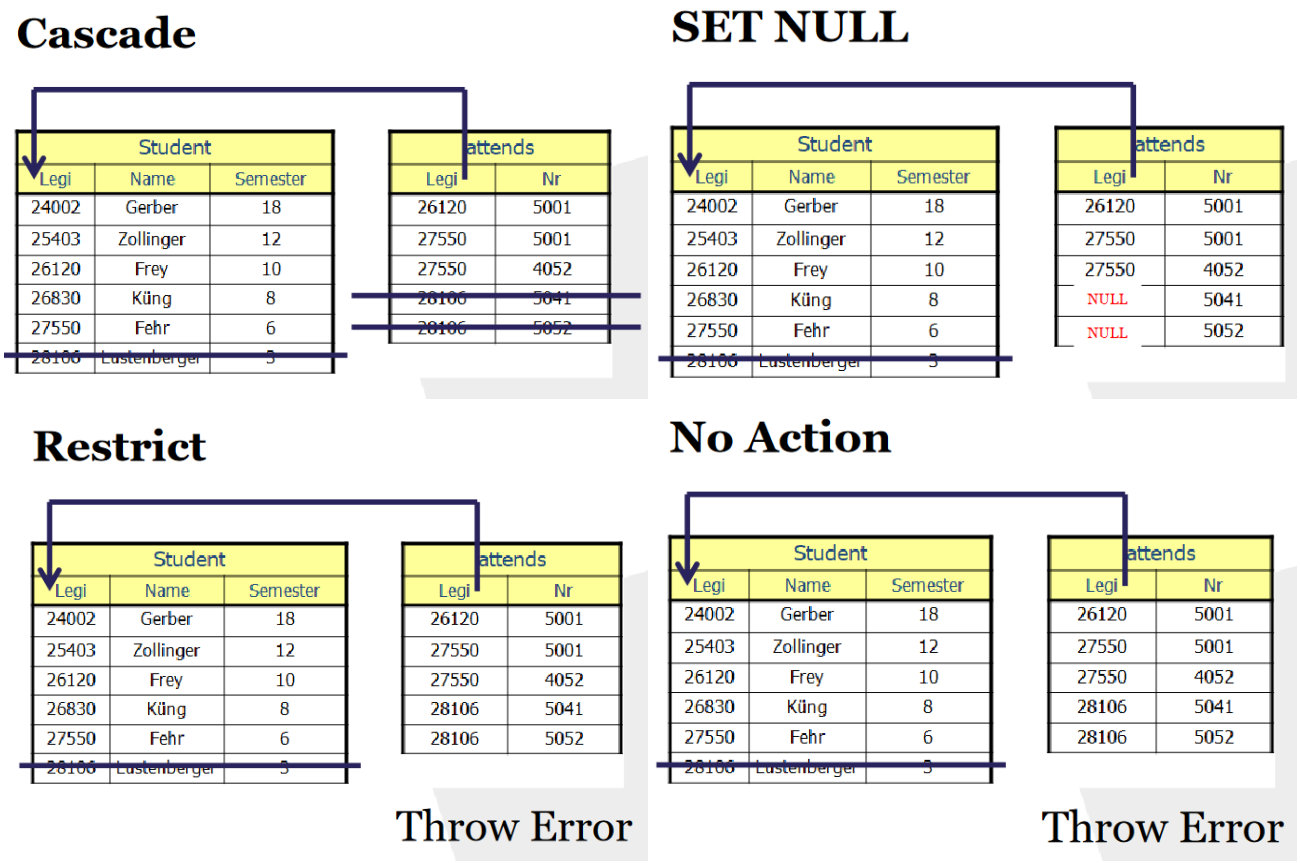


Figure 15: Update Modes

The difference between restrict and no action is very subtle and in many database management systems there is no difference at all. In principle, restrict throws an error immediately while no action throws an error after trying.

7 Entity-Relationship Model

The process of implementing a real-world application contains the following steps:

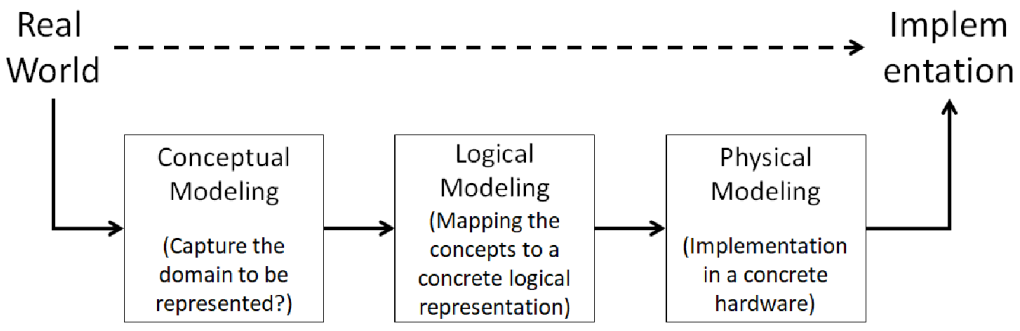


Figure 16: Modeling Process

Conceptual modeling tries to capture the domain to be represented. It only contains elements that are explicitly stated and not things that seem like common sense. The logical modeling converts the conceptual model into a logical representation, which is then concretely implemented by the physical modeling.
The conceptual model specifies all valid instances.

One way of conceptual modeling is the creation of an **entity-relationship model**. An ER model, models an application in the following three element types:

- **Entity Sets**
  - A set of similar entities
  - Entity: An object in the real world that is distinguishable from other objects
  - "Similar": Entities in the same entity set share the same attributes
  - Roughly equivalent to classes in object-oriented programming
- **Attributes**
- **Relationships**
  - Relationships are connections among two or more entity sets.

An ER-diagram is a graphical way of representing entities and the relationships among them.

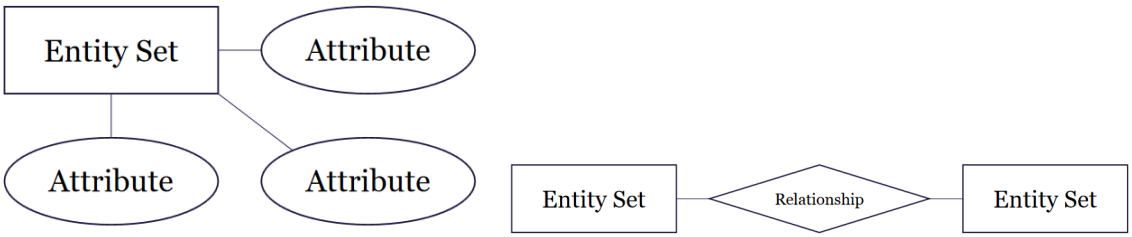


Figure 17: Entity-Relationship Diagram

7.1 Keys

A key is a minimal set of attributes whose values uniquely identify an entity in the set. There might be multiple keys, all of them are called candidate keys. Only one candidate key is called the primary key and it is underlined in the schema. Each identity set must have a primary key.

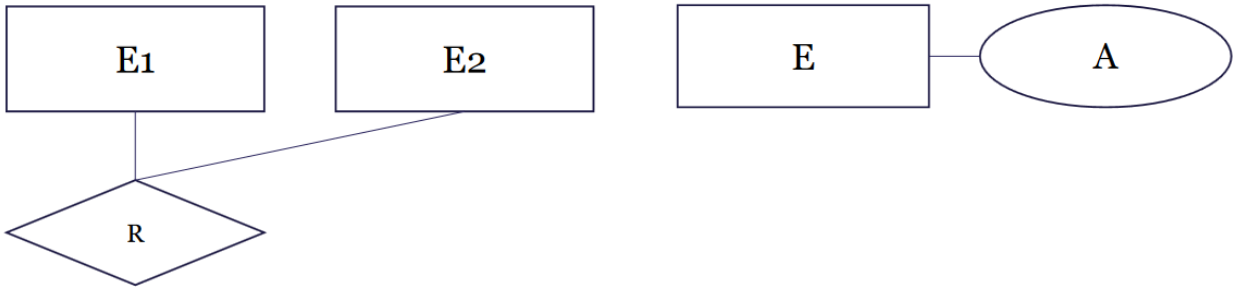
7.2 Formal Semantics

ER-diagrams are a graphical way of modeling data, but that does not make them vague or imprecise. ER-diagrams can be made formal by defining a mapping from an ER-diagram to a set of database instances.

- All values we can take $\mathcal{D} = \mathcal{B} \cup \Delta$
  - Concrete Values (Int, String, Floating Point, etc.): $\mathcal{B}$
  - Abstract Values (Corresponding to an entity): Δ
- Entity Set $E \implies$ 1-ary Predicate $E(x)$. $[E(x) = True$ if x is of Entity Type $E]$
- Attribute $A \implies$ binary Predicate $A(x, y)$. $[A(x, y) = True$ if x has attribute $y]$
- n-ary Relation $R \implies$ n-ary Predicate $R(x_1, ..., x_n)$. $[R(x_1, ..., x_n) = True$ if $(x_1, ..., x_n)$ participate in $R]$
- A first-order interpretation $\mathcal{J}$ satisfies:
  - $E^{\mathcal{J}} \subseteq \Delta$
  - $A^{\mathcal{J}} \subseteq \Delta \times \mathcal{B}$

- $R^{\mathcal{J}} \subseteq \Delta^n$

Each subgraph introduces a first-order logic sentence. This tells us what a valid database instance should satisfy. A relationship is simply a type constraint.



$$\forall x_1 \in \Delta, x_2 \in \Delta.\ R(x_1, x_2) \implies E_1(x_1) \wedge E_2(x_2) \qquad \forall x, E(x) \implies \exists!\ y.\ A(x, y) \wedge y \in \mathcal{B}.$$

Figure 18: Entity-Relation Introducing Logic Constraints

Checking whether two ER-diagrams are equivalent comes down to checking whether the first-order logic sentences are equivalent.

7.3 Cardinality

- **1-to-many relationship** - A is in a 1-to-may relationship with B if one A entity can have relationship with multiple B entities, and one B entity can only have relationship with only one A entity.

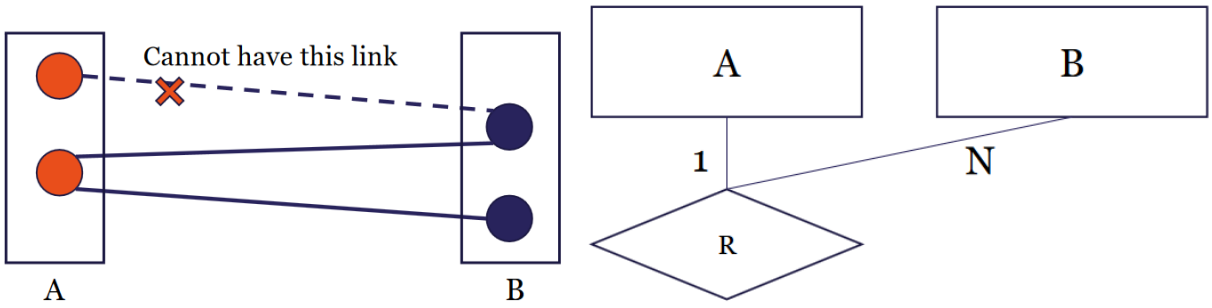


Figure 19: 1-to-many Relationship

- **many-to-1 relationship** - A is in a many-to-1 relationship with B if one B entity can have relationship with multiple A entities, and one A entity can only have relationship with only one B entity.

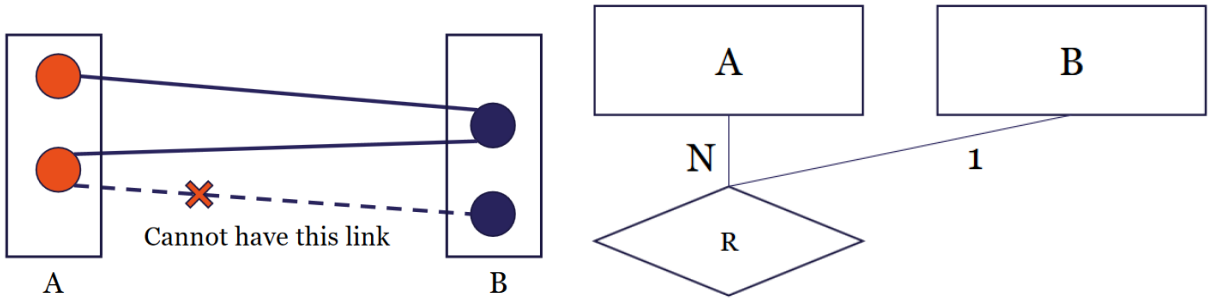


Figure 20: many-to-1 Relationship

- **1-to-1 relationship** - A is in a 1-to-1 relationship with B if one A entity can only have relationship with one B entity, and one B entity can only have relationship with one A entity.

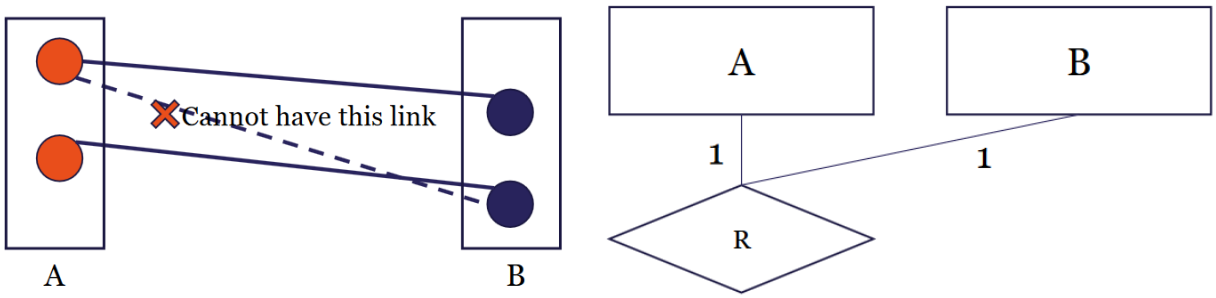


Figure 21: 1-to-1 Relationship

- **many-to-many relationship** - A is in a many-to-many relationship with B if one A entity can have relationships with multiple B entities, and one B entity can have relationships with multiple A entity.

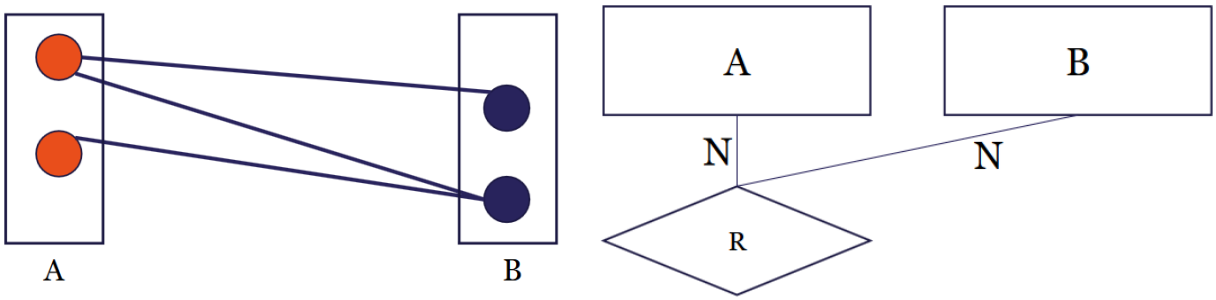
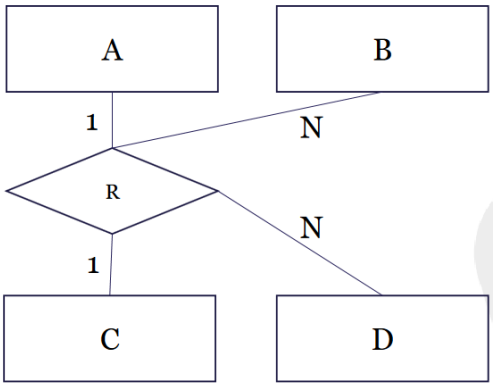


Figure 22: many-to-many Relationship

Every "1" in the diagram is associated with a first-order logic constraint.

The ER-diagram on the right specifies the following two constraints:

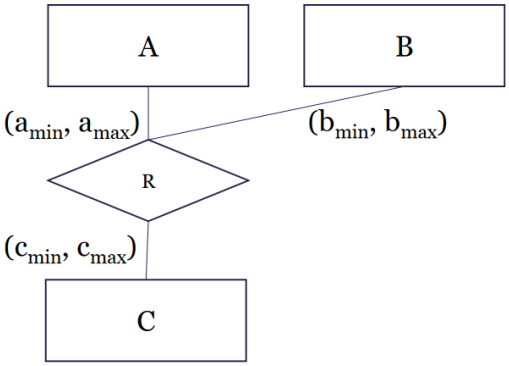
- $\forall x_A, x_B, x_C, x_D.\ R(x_A, x_B, x_C, x_D) \implies \neg\exists x'_A.\ R(x'_A, x_B, x_C, x_D) \wedge (x'_A \neq x_A)$
- $\forall x_A, x_B, x_C, x_D.\ R(x_A, x_B, x_C, x_D) \implies \neg\exists x'_C.\ R(x_A, x_B, x'_C, x_D) \wedge (x'_C \neq x_C)$



We can also have a more expressive (min, max)-notation.

This specifies the following constraints:

- $\forall x_A.\ A(x_A) \implies \exists^{\geq a_{min}, \leq a_{max}} x'_B, x'_C.\ R(x_A, x'_B, x'_C)$
- $\forall x_B.\ B(x_B) \implies \exists^{\geq b_{min}, \leq b_{max}} x'_A, x'_C.\ R(x'_A, x_B, x'_C)$
- $\forall x_C.\ C(x_C) \implies \exists^{\geq c_{min}, \leq c_{max}} x'_A, x'_B.\ R(x'_A, x'_B, x_C)$

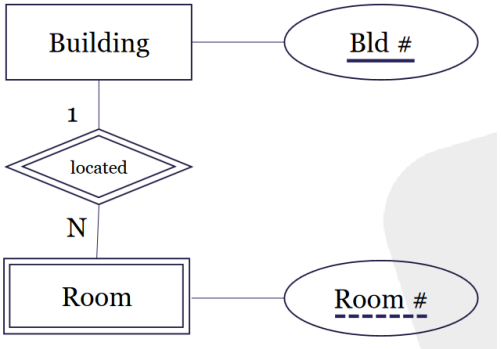


7.4 Weak Entity

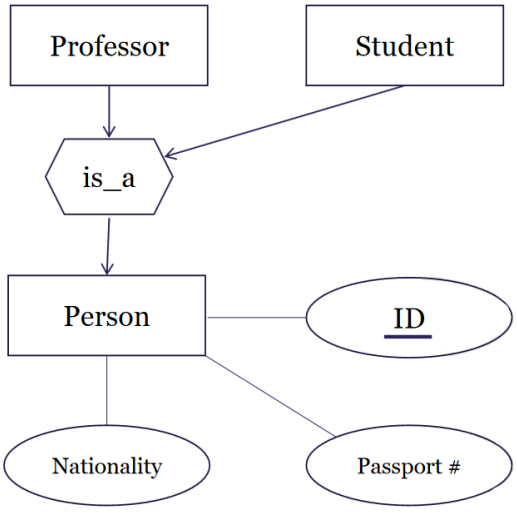
Some entity's existence relies on other entries. For example a room cannot exist without a building. This means room is a weak entity relying on building. The key of room is therefore (Bld#, Room#).

7.5 Generalization

Generalization are "is-a" relationships. This means that the entities share the generalized attributes (and primary key). For example both a professor and a student are people and therefore share people attributes.



(a) Weak Entity



(b) Generalization

7.6 ER-Model to Relational Model

If we have conceptually modeled our application in the entity-relationship model, we can logically model it in the relational model. The following principles apply to the conversion.

- Entities become relations.
- Relationships become relations.
- Merge relations with the same key.
- Generalization.
- Weak entities.
- entity sets become tables.
- attributes of entity sets become attributes of the table.
- When there are no cardinality constraints, relationships become tables, containing the keys of all participating entity sets.

8 Functional Dependency

A badly designed database will experience a list of problems.

- Update-Anomaly
- Insert-Anomaly
- Delete-Anomaly

8.1 Redundancy

One of the causes for such anomalies can be redundancy. It is a waste of storage space and it incurs additional work to keep multiple copies of data consistent. However redundancy is not always bad. Through redundancy it is possible to improve locality. It is a tradeoff between space and performance. Sometimes redundancy is even needed for example to provide fault tolerance and ensure availability.
No matter whether redundancy is good or bad, we want to understand:

- How to *model* data redundancy?
- How to *reason* about redundancy?

The notion on how to do this is called **functional dependency**.

Given

- Schema: Relation e.g. $\mathcal{R}(A : D_A, B : D_B, C : D_C, D : D_D)$
- Instance: $R \subseteq D_A \times D_B \times D_C \times D_D$
- Let $\alpha \subseteq \mathcal{R}, \beta \subseteq \mathcal{R}$
  α is a subset of attributes.

There exists a **functional dependency** $\alpha \to \beta$ if and only if $\forall r, s \in R : \; r.\alpha = s.\alpha \implies r.\beta = s.\beta$. In other words, there exists a functional dependency $\alpha \to \beta$ if and only if for any two tuples r and s in database instance R, if r and s share the same value on columns α, then they share the same values on column β.
We write $R \models \alpha \to \beta$ if R satisfies $\alpha \to \beta$.

8.1.1 Keys & Cardinalities

- $\alpha \subseteq R$ is a superkey iff
  - $a \rightarrow \mathcal{R}$ ($\mathcal{R}$ means all columns)
  - This means that if we know the value of columns α, we know the value for the rest of the columns in $\mathcal{R}$. $\implies$ α must be a superset of a key.
- $\alpha \rightarrow \beta$ is minimal iff
  - $\forall A \in \alpha : \ (\alpha - \{A\} \not\rightarrow \beta$
- Notation for minimal functional dependencies:
  - $\alpha \dot{\rightarrow} \beta$
- $\alpha \subseteq \mathcal{R}$ is a candidate key iff
  - $\alpha \dot{\rightarrow} \mathcal{R}$

There also exists a relationship between keys and cardinalities.

- Cardinalities define functional dependencies
- Functional dependencies determine keys
- Not all functional dependencies are derived from the cardinality information.

8.1.2 Inference

Assuming we are given a set of functional dependencies F. We want to figure out how new functional dependencies can be *implied* from this set of functional dependencies. There are two ways to define which functional dependencies will follow:

- $F \models \alpha \rightarrow \beta$: Semantics - Any R that satisfies F will also satisfy $\alpha \rightarrow \beta$
- $F \vdash \alpha \rightarrow \beta$: Syntax - We can derive $\alpha \rightarrow \beta$ by applying a set of inference rules over F. More precisely:
  - Let F be a set of FDs on scheme $\mathcal{R}$ and $\alpha \rightarrow \beta$ be another FD on $\mathcal{R}$. Then F implies $\alpha \rightarrow \beta$, denoted by $F \models \alpha \rightarrow \beta$, if every relation instance R of $\mathcal{R}$ that satisfies all FDs in F also satisfies $\alpha \rightarrow \beta$.
  - Let F be a set of FDs on scheme $\mathcal{R}$. Then, the **closure** of F, denoted by F^+, is the set of all FDs implied by F.
  - Let F and G be sets of FDs on scheme $\mathcal{R}$. Then F, and G are **equivalent**, denoted by $F \equiv G$, if $F \models G$ and $G \models F$.

A fundamental result shows that inference by syntax and inference by semantics are equivalent.

$$F \models \alpha \rightarrow \beta \iff F \vdash \alpha \rightarrow \beta. \text{ when } \vdash \text{ is defined by Armstrong's Axioms)}$$

Armstrong's axioms provide us with inference rules:

- **Reflexivity**
  - $\alpha \subseteq \beta \implies \beta \rightarrow \alpha$
  - Special case: $\mathcal{R} \rightarrow \alpha$
  - We call these FDs **trivial FDs**.
- **Augmentation**

- $\alpha \to \beta \implies \alpha\gamma \to \beta\gamma$
- (Notation $\alpha\gamma := \alpha \cup \gamma$)

- **Transitivity**
  - $\alpha \to \beta \wedge \beta \to \gamma \implies \alpha \to \gamma$

These three axioms are both **complete** and **sound**. All possible functional dependencies can be implied from these axioms.
Some other rules:

- Union of FDs:
  $\alpha \to \beta \wedge \alpha \to \gamma \implies \alpha \to \beta\gamma$
- Decomposition:
  $\alpha \to \beta\gamma \implies \alpha \to \beta \wedge \alpha \to \gamma$
- Pseudo transitivity:
  $\alpha \to \beta \wedge \beta\gamma \to \theta \implies \alpha\gamma \to \theta$

Let F be a set of FDs over $\mathcal{R}$, $\alpha \subseteq \mathcal{R}$ is a set of attributes of $\mathcal{R}$. The **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.

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

An algorithm to, given F a set of FDs over $\mathcal{R}$ and $\alpha \subseteq \mathcal{R}$ a set of attributes of $\mathcal{R}$, find α^+. The closure allows us to answer many questions easily.

Algorithm 1 *Closure*(F, α)

```
α⁺ := α
repeat
    α⁺_old := α⁺
    for each FD β → γ ∈ F do
        if β ⊆ α⁺ then
            α⁺ := α⁺ ∪ γ
until α⁺ = α⁺_old
return α⁺
```

- How to check if $F \vdash \alpha \to \gamma$?
  Calculate α^+ and check $\gamma \in \alpha^+$
- How to check if $F \vdash G$?
  For each $\alpha \to \gamma \in G$, check $F \vdash \alpha \to \gamma$
- How to check if F is equivalent to G?
  Check $F \vdash G$ and $G \vdash F$
- how to check if $K \subseteq \mathcal{R}$ is a superkey given FDs F?
  Check $F \vdash K \to \mathcal{R}$

Let F be a set of FDs. There might be redundant FDs in F that can be derived from other FDs. It is natural to ask, how to simplify F to remove such redundant FDs.
F is a set of FDs. A **minimal cover** of F is a set of FDs G that has the following properties:

- G is equivalent to F.

- All FDs in G have the form $X \to A$, where A is a single attribute.
- It is not possible to make G smaller
  - Deleting a FD, i.e. $G - \{X \to A\} \not\equiv G$, for any FD $X \to A \in G$
  - Deleting an attribute from the left hand side of a FD, i.e. $G - \{XA \to B\} + \{X \to B\} \not\equiv G$, for any FD $XA \to B \in G$.

The minimum basis can be computed as follows:

- Step 1: Let G be the set of FDs obtained from F by decomposing the right hand sides of each FD to a signle attribute.
- Step 2: Remove FDs that are trivial.
- Step 3: Remove all redundant attributes from teh left hand sides of FDs in G.
  For each $X \to Y$, take each attribute x in X, if $X - \{x\} \to Y$ implies $X \to Y$, then replace $X \to Y$ with $X - \{x\} \to Y$
- Step 4: From the resulting set of FDs, remove all redundant FDs.

8.1.3 Decomposition of Relations

Bad relations combine several concepts. Decompose the relations such that each concept is in one relation $R \rightsquigarrow R_1 ... R_n$.
Decomposition is **lossless** if $R = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$.
Dependencies are **preserved** if $FD(R)^+ = [FD(R_1) \cup ... \cup FD(R_n)]^+$.

When is a decomposition lossless? Let $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_\in$. We consider the following decomposition of $\mathcal{R}$:

- $R_1 = \Pi_{\mathcal{R}_1}(R)$
- $R_2 = \Pi_{\mathcal{R}_2}(R)$

This decomposition is lossless, i.e. $R = R_1 \bowtie R_2$, if

- $(\mathcal{R}_1 \cap \mathcal{R}_2) \to \mathcal{R}_1$ **OR**
- $(\mathcal{R}_1 \cap \mathcal{R}_2) \to \mathcal{R}_2$

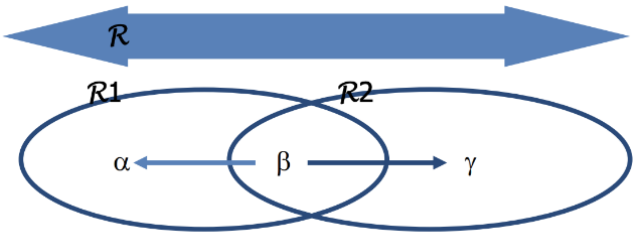


Figure 24: Lossless Decompositon

Losslessness however, does not guarantee preservation of functional dependencies.

9 Normal Form

We want to formalize what is a good database design and formally model and enforce the properties we want in a good design. For this we need:

- A precise way to describe the property of the data - e.g. **functional dependency** as a precise way to describe **data redundancy**
- Given the property of the data, describe the property of the DB design - **normal form**

What is a normal form. Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity.
Normal forms are common ways for normalization. For each normal form, we consider the following problems:

- Given a relation schema R, and a set of functional dependencies FD, how to decide whether $\{R, FD\}$ satisfies a given normal form.
- Given $\{R, FD\}$ that satisfies a given normal form, what are the set of properties that it will have.
- Given a relational schema R, and a set of functional dependencies FD, how to generate a new schema R' such that $\{R', FD\}$ satisfies a given normal form.

9.1 First Normal From (1NF)

A database in first normal form contains only atomic domains. This means we do not allow non-atomic domains like arrays. This makes the concept of *key* well defined (or, easier to define).

9.2 Second Normal From (2NF)

The second normal form tries to remove data redundancy. $\{R, FD\}$ is in second normal form if and only if every non-key attribute is minimally dependent on every key. Minimally dependent means that no attribute depends on part of the key. Relations that are not in second normal form experience the following problems:

- Insert Anomalies
- Update Anomalies
- Delete Anomalies

All of these can lead to redundant/inconsistent updates of the relations. In order to enforce the second normal form, we take the evil functional dependence and decompose the relation into multiple relations. This decomposition is lossless. The second normal form gets rid of insert anomalies but still experiences update and delete anomalies.

9.3 Third Normal From (3NF)

A relation R is in third normal form if and only if for all functional dependencies $\alpha \to B$, at least one condition holds:

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

- α is a superkey of R (superkey = superset of some candidate keys)

Intuitively, if $\alpha \to B$ does not satisfy any of these conditions, α is a concept in its own right. The third normal form tries to get rid of *transitive dependencies* (e.g. $A \to B, B \to C$). In 3NF, we still have update and delete anomalies.

9.4 Boyce-Codd Normal Form

A relation R is in BCNF if and only if for all $\alpha \to B$, at least one condition holds:

- $B \in \alpha$ ($\alpha \to B$ is a trivial FD)
- α is a superkey of R (superkey = superset of some candidate keys)

Intuitively, in each relation, you only store the same information once.

9.5 Decomposition Algorithm

Algorithm 2 Decomposition($\mathcal{R}$)

$Result = \{\mathcal{R}\}$
while $\exists R_i$ in Result such that R_i is not BCNF **do**
  Let $\alpha \to \beta$ be the evil FD
  $\mathcal{R}_i^1 = \alpha \cup \beta$
  $\mathcal{R}_i^2 = \mathcal{R}_i - \beta$
  $Result = (Result - \mathcal{R}_i) \cup \{\mathcal{R}_i^1, \mathcal{R}_i^2\}$
return $Result$

This algorithm takes as input a schema $\mathcal{R}$ and outputs a lossless decomposition $\mathcal{R}_1, ..., \mathcal{R}_n$ such that they will perfectly recover the information in $\mathcal{R}$ and $\mathcal{R}_1, ..., \mathcal{R}_n$ are in BCNF. BCNF does not preserve all functional dependencies, and it does not get rid of all data redundancies, only the ones that are caused by functional dependencies.

9.6 Synthesis Algorithm

Algorithm 3 Synthesis($\mathcal{R}$)

Compute the minimal basis F_c of F
For all $\alpha \to \beta \in F_c$, create $\mathcal{R}_{\alpha\cup\beta}(\alpha \cup \beta)$
If none of the above relations contains a superkey, add a relation with a key
Eliminate $\mathcal{R}_\alpha$ if there exists $\mathcal{R}_{\alpha'}$, such that $\alpha \subseteq \alpha'$

This algorithm takes as input a schema $\mathcal{R}$ and outputs a lossless decomposition $\mathcal{R}_1, ..., \mathcal{R}_n$ that is in 3NF. The synthesis algorithm preserves all functional dependencies, but since it is not in BCNF it is not free of redundancies caused by functional dependencies.

9.7 Multi-Value Dependency

We formally define MVD as follows:

$$\begin{aligned} \alpha \twoheadrightarrow \beta \text{ for } R(\alpha, \beta, \gamma) \text{ iff.} \\ \forall t_1, t_2 \in R,\ t_1.\alpha = t2.\alpha \implies \exists t_3, t_4 \in R : \\ t_3.\alpha = t_4.\alpha = t_1.\alpha = t_2.\alpha; \\ t_3.\beta = t_1.\beta; t_4.\beta = t_2.\beta; \\ t_3.\gamma = t_2.\gamma; t_4.\gamma = t_1.\gamma; \end{aligned}$$

An intuitive way to think about MVD is to think about it in terms of joins. If we have $(\alpha \twoheadrightarrow \beta)$ we can decompose R losslessly into $R_1 = \Pi_{\alpha,\beta}R$ and $R_2 = \Pi_{\alpha,\gamma}R$ such that $R = R_1 \bowtie R_2$. The following laws hold for multi-value dependencies:

- **Trivial MVDs**
  $\mathcal{R}(\alpha, \theta) : \ \alpha \twoheadrightarrow \alpha\theta. \ (\alpha \twoheadrightarrow \mathcal{R})$
  Similarly: $\mathcal{R}(\alpha, \theta) : \ \alpha \twoheadrightarrow \theta. \ (\alpha \twoheadrightarrow \mathcal{R} - \alpha)$
  $\beta \subseteq \alpha \implies \alpha \twoheadrightarrow \beta$
- **Promotion**
  $\alpha \to \beta \implies \alpha \twoheadrightarrow \beta$
  but $\alpha \twoheadrightarrow \beta \not\Longrightarrow \alpha \to \beta$
- **Complement**
  $\alpha \twoheadrightarrow \beta \implies \alpha \twoheadrightarrow \mathcal{R} - \alpha - \beta$
- **Multi-Value Augmentation**
  $\alpha \twoheadrightarrow \beta \wedge (\delta \subseteq \gamma) \implies \alpha\gamma \twoheadrightarrow \beta\delta$
- **Multi-Value Transitivity**
  $(\alpha \twoheadrightarrow \beta) \wedge (\beta \twoheadrightarrow \gamma) \implies \alpha \twoheadrightarrow \gamma$

9.8 Fourth Normal Form

A relation R is in 4NF if and onlly if for all $\alpha \twoheadrightarrow \beta$, at least one condition holds:

- $\alpha \twoheadrightarrow \beta$ is trivial
- α is a superkey of R (superkey = supereset of some candidate keys)

The fourth normal form implies BCNF. This algorithm takes a schema $\mathcal{R}$ and outputs a lossless

Algorithm 4 Decompositon($\mathcal{R}$)

```
Result = {R}
while ∃R_i in Result such that R_i is not in 4NF do
    Let α ↠ β be the evil MVD
    R_i^1 = α ∪ β
    R_i^2 = R_i − β
    Result = (Result − R_i) ∪ {R_i^1, R_i^2}
return Result
```

decomposition $\mathcal{R}_1, ..., \mathcal{R}_2$ that is in 4NF.

9.9 (De-)Normalization

Now that we have seen all these normal forms the question of whether higher normalization is better poses itself. The answer is; it depends. There is always a tradeoff between performance and space. A denormalized database could be faster to read while redundancy incurs a higher overhead to maintain consistency.
What we have seen is that our normal forms provide:

- Lossless decomposition up to 4NF.
- Preserving dependencies up to 3NF.

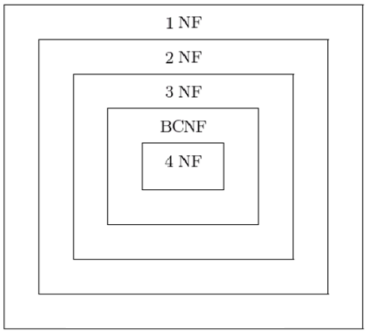


Figure 25: Hierarchy of Normal Forms

10 Database Systems

In the real world, a database management system can be quite complex. In this course we focus on relational databases with disk-oriented architecture, where all data is stored on disk, the disk is larger and slower than memory and favors different access patterns.

10.1 Overview

A database management system can be abstracted into several layers, where an SQL query is passed in at the top, processed through the layers and the result is returned. The different layers try to hide complexity from layers above, they provide an interface to the upper layers and require services from the lower levels.

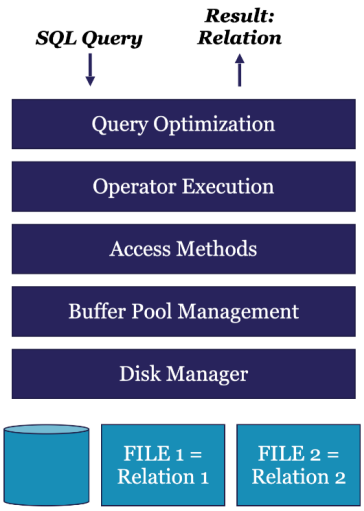


Figure 26: Layers of a DBMS

- **Disk Manager**
  The disk manager interacts with the disks, it allocates, deletes, and fetches pages. The idea is that no other layer has to interact with the disk directly.
- **Buffer Pool Manager**
  The buffer pool manager maintains an in-memory buffer such that all upper layers have the *illusion* that the entire data is in memory and not on disk. It provides the functionality to fetch and update pages.
- **Access Methods**
  It provides different ways of accessing data from a relation such as

- Sequential Scan
- B-Tree Index
- Hash Table
- Sort

It provides a higher-level abstraction to access information in a table without interacting with the buffer or disk.

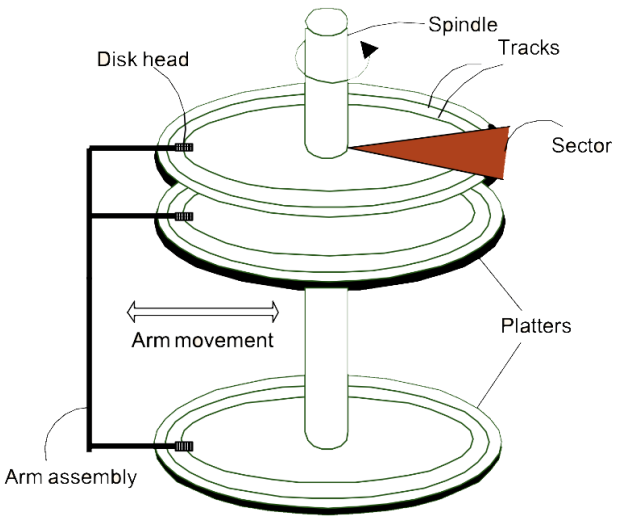
- **Operator Execution** This layer executes a relational algebra operator such as
  - Join
  - Projection
  - Select

  It uses the access methods from the lower level to implement these operators.
- **Query Optimization**
  Given an SQL query, this layer generates a good execution plan (a tree of relational algebra operators). It uses the provided relational algebra operators from the lower level.

10.2 Storage Hierarchy

In an ideal world we would have private, infinitely large and fast, non-volatile and inexpensive memory. However, this is not the case. Our memory is structured like a pyramid. The faster and closer to the processor our memory is, the smaller, volatile and expensive it is. One access to main memory is already around 60 CPU cycles. A random access on a hard drive is closer to 10 million cycles.
Because our architecture is focused around hard drives, it is important to know how they work. A hard drive consists of multiple spinning disks and arms. On each arm there is a read/write head, which can read or write whatever part of the disk is currently beneath it. Only one head can read or write at any one point in time. The disk is split into multiple circular tracks. To read a specific track, the head is moved over the right track by an arm assembly. A block on the disk is a multiple of the fixed sector size. Now we take a look at a simplified performance model for hard drives:

- Seek / Rotate: t_s, t_r
  - seek time (moving arms to position disk head on track)
  - rotational delay (waiting for block to rotate under head)
- Transfer: t_{tr}
  - transfer time (actually moving data to/from disk surface)
- Random access D times:
  - $D(t_s + t_r + t_{tr})$
- Sequential access D times:
  - $t_s + t_r + Dt_{tr}$



(a) Model of Hard Drive

In our model we focus on a simple hierarchy consisting of the hard drive, the DRAM and the CPU. Because the connection between DRAM and the CPU is dominatingly fast compared to

the connection from hard drive to DRAM, we will focus on how to optimize data movement on the latter connection.

10.3 Disk Manager

The disk manager is the lowest layer of the DBMS software stack and it manages the space on disk. Higher levels call upon this layer to:

- allocate/deallocate a page
- read/write a page

A DBMS stores a database as one or more files on disk. Sqlite for example uses a single file. PostgreSQL on the other hand uses a collection of files.
The disk manager organizes the files as a collection of pages. A page is a fixed-size block of data that with a unique identifier. A page can contain tuples, meta-data, indexes or log records. A relation is stored as a collection of pages, where each page is a collection of tuples. There are multiple different ways to manage the collection of pages.

10.3.1 Heap File

A heap file is an unordered collection of pages, where tuples are stored in random order. Each record has a **Record ID** that consists of a Page ID and a Slot ID to identify in which page and where on the page the record is stored. There are two different ways to implement a heap file.

Linked List
A header page stores two pointers. One pointer points to the first entry in a linked list of free pages. The other pointer points to the first entry in a linked list of data pages. If a page is filled with tuples, it is moved from the free list to the data list.

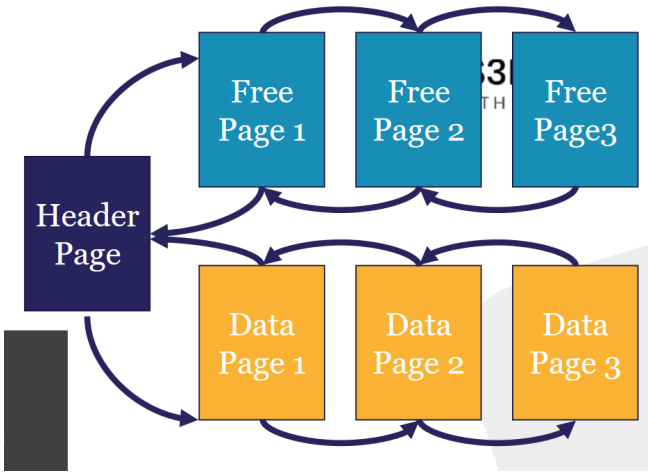


Figure 28: Linked List Implementation

Page Directory
In a page directory, there is a set of header and data pages. A header page contains pointer to data pages as well as the number of free slots in those pages. The pages are allocated sequentially.

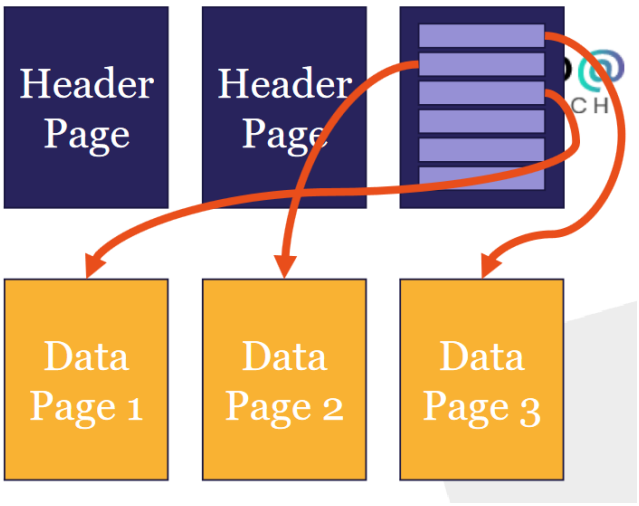


Figure 29: Page Directory Implementation

In the naive implementation using linked list, there will be free spaces on data pages here and there. There could also be free spaces when using page directory, but those spaces are easier to fill.

Performance Model
We assume the directory fits entirely into memory and is allocated. The number of pages we want to access is D.

Linked List
Pages are randomly allocated on disk (worst case)

- Insert
  $t_{s+r} + 2 \cdot t_{trans}$ (if page #1 has slot available)
- Find Record (by non-RID value)
  $\frac{D}{2}(t_{s+r} + t_{trans}$ (In expectation)
- Scan (all tuples)
  $D(t_{s+r} + t_{trans})$

Page Directory
Pages are sequentially allocated on disk (best case)

- Insert
  $t_{s+r} + 2 \cdot t_{trans}$ (if page #1 has slot available)
- Find Record (by non-RID value)
  $(t_{s+r} + \frac{D}{2} t_{trans}$ (In expectation)
- Scan (all tuples)
  $t_{s+r} + D \cdot t_{trans}$

10.3.2 Page Layout

Now we look at how data is stored inside a page. A page contains both a page header and actual data. The header contains meta-data like page size, DBMS version, compression/encryption information and checksum. In practice, often the header grows from the beginning of the page and the data from the end of the page. The data itself can be organized in different ways.

Naive Strategy
Each page contains fixed-length slots and in the header we keep track of the number of tuples in the page. With this implementation we run into trouble, when we encounter data that has a non-fixed length like strings.

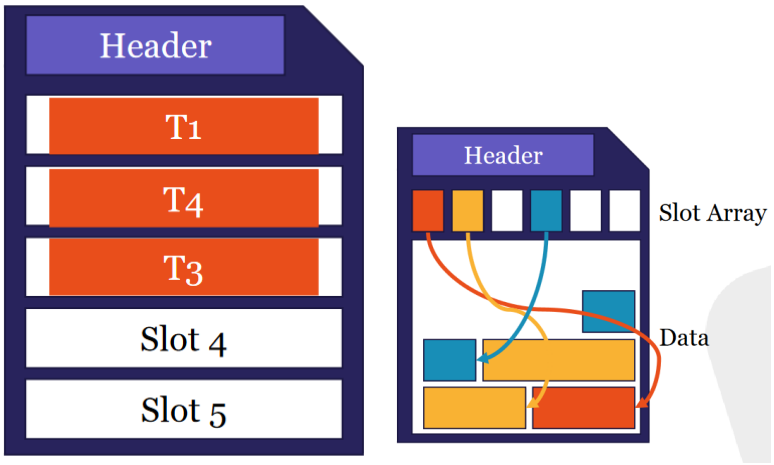


Figure 30: Implementation of Page Layout

Slotted Page
Another approach is to have slots of pointers. Each slot contains a pointer that then points to the start position of each data record. With this approach we can reorder the page without changing the Record ID.

10.3.3 Tuple Layout

- **Fixed Length Fields**
  - direct access to these fields
- **Variable Length Fields**
  - store (length, pointer) as part of a fixed-length field
  - store payload information in a variable-length field
  - access in two steps: retrieve pointer + chase pointer
- **NULL Values**
  - Bitmap: set 1 if value of a field is NULL

Another option to store variable length field would be to separate the fields by special delimiter symbols.

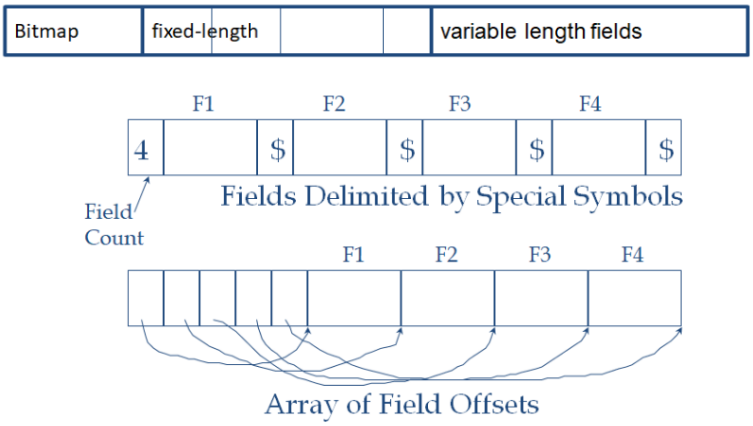


Figure 31: Tuple Implementation

10.3.4 Tuple Store vs Column Store

For now we have looked at ways to store entire tuples together. However, we could also store the data in a column layout, depending on the workload we expect.

- **On-Line Transaction Processing**
  Simple queries that read/update a small amount of data that is related to a single entity in the database.
- **On-Line Analytical Processing**
  Complex queries that read large portions of the database spanning multiple entities.

Our current layout always forces us to read the entire tuple even if we only need a few columns in a potentially huge table. This could waste a lot of I/O operations. One way of tackling this problem would be to store the values of the same column together.

- **Advantages**
  - Reduces the amount of wasted I/O because the DBMS only read the data it needs
  - Easier data compression, since we only have to store each value that appears in the column once.
- **Disadvantages**
  - Slow for point queries, inserts, updates and deletes since we need to search to tuple attributes one-by-one.

10.4 Buffer Manager

The buffer manager acts like the intermediate layer between the system and disk manager. It tries to provide the illusion that all the data is in DRAM. The core question of the buffer manager is how to optimally replace pages in the buffer. The optimal strategy depends on whether we know what is going to happen in the future.

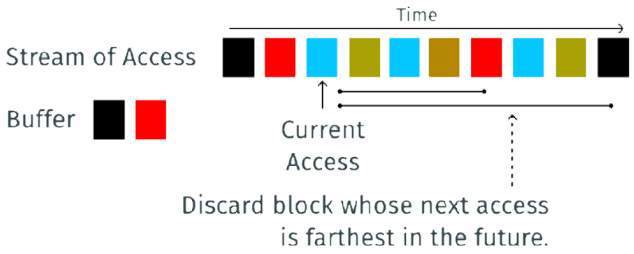


Figure 32: Optimal Strategy

If we know the future access pattern, we discard the block whose next access is the farthest in the future. This strategy is proven to be optimal and called **Bélády's Min**.

But how do we chose which page to evict in practice, where we do not always know the future. One way to do it is called **Least Recently Used**. For each page in the buffer pool, we keep track of the time it was last used. We then always evict the page, that has been used the furthest in the past. This method works well for repeated accesses to popular pages. We encounter a problem when the number of pages exceeds the number of buffer slots. If we then sequentially access all the data, we are going to miss on every request. This is called **sequential flooding**. But we know that LRU is at most twice as bad as the optimal strategy when given twice the memory.

Another common strategy is called **Most Recently Used**. For each page in the buffer pool, we keep track of the time it was last used. We then always evict the page that hast been used the most recently.

How well a replacement strategy performs depends on the access pattern. Such access patterns could be

- Sequential: Table Scans
- Hierarchical: Index Scans
- Random: Index Lookup
- Cyclic: Nested Loop

Different replacement strategies work better for different cases. In many cases, the DBMS knows the pattern and can tell the buffer manager.

10.5 Access Methods

This layer provides different ways of accessing data from a relation. To the upper layer it provides an abstraction to access information in a table without interacting with the buffer or disk. It conducts a sequence of invocations to the buffer manager to access information in a relation. There are different ways to conduct the accesses.

10.5.1 Sequential Scan

In the sequential scan we tell the buffer manager to sequentially bring all the pages of the relation and scan the tuples. It would look something like

- Bring me page 1, scan all the tuples and check
- Bring me page 2, scan all the tuples and check
- Bring me page 3, scan all the tuples and check

The cost of a sequential scan to read D pages:

- When pages are sequentially allocated
  $t_{s+r} + D(t_{trans}$
- WHen pages are randomly allocated
  $D(t_{s+r} + t_{trans})$

If we also update and write back the tuples, we need one more t_{trans}.

10.5.2 Index

To index a relation is to build a data structure f over the relation R. That given information about a tuple, we can find the Record ID.

$$f(\text{"}Age = 5\text{"}) \mapsto RID$$

Usually, evaluating f is cheaper than a sequential scan of the whole relation. There are different kinds of indexes such as B-Tree, Hash table, Learned index and many more.

10.5.3 B-Tree (B+ Tree)

A B-Tree is a self-balancing tree data structure that keeps data sorted. As usual in balanced trees we have $\mathcal{O}(\log n)$ asymptotic complexity for search, insertion and deletion. It is a generalization of a binary search tree in which each node contains more than two children. It is optimized for system that read and write large blocks of data (i.e. read in pages).
A B-Tree is a M-way search tree that is:

- Perfectly balanced

- Every inner node other than the root, is at least half-full
- Every inner node with k keys has $k+1$ non-null children.

Each inner node's keys act as separation values which divide its subtrees. For example if an inner node has 3 child nodes then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.
There are two different flavors of B-Trees. In a **clustered** B-Tree, the leaf nodes contain the actual tuple. It is only possible to have one such clustered B-Tree per relation. In an **unclustered** B-Tree, the leaf node contains the Record Id, pointing to the tuple.

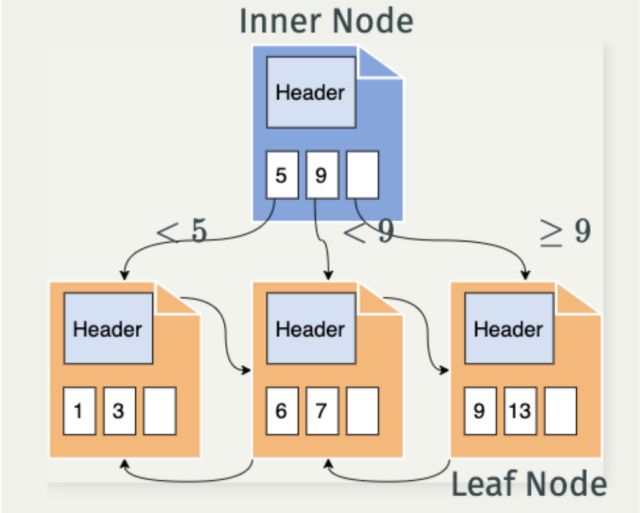


Figure 33: B-Tree

We assume a B-Tree with node size M and the total number of tuples N. The depth of the B-Tree is $\log_M N$. To find a single key:

- #I/O, Unclustered B-Tree:
  - $\log_M N + 1$
  - $\log_M N$: Read one pager per level
  - 1: Read the actual tuple
- #I/O, Clustered B-Tree:
  - $\log_M N$

If we want to find all the tuples in a range of keys:

- #I/O, Unclusterd B-Tree:
  - $\log_M N + \frac{\#tuples}{tuple\_per\_page} + \#tuples$ (worst case)
  - $\log_M N$: REad one page per level
  - $\frac{\#tuples}{tuple\_per\_page}$: #leaf pages one need to read to get all RIDs
  - $\#tuples$: Cost of reading all tuples unclustered, which is one I/O operation per result in the worst case.
- #I/O, Clustered B-Tree
  - $\log_M N + \frac{\#tuples}{tuple\_per\_page}$
  - $tuple\_per\_page$ is lower, since in a clustered B-Tree the entire tuple is stored.

To insert a tuple into a B-Tree, we find the right leaf node L and insert our tuple. If L has enough space we are done, otherwise we split L and insert the key into the parent of L. Splitting is done by finding the median key.
To delete a tuple, we find the right leaf node L and remove the entry. If L is at least half full we are done, otherwise we might have to merge two leaf nodes or borrow a tuple from a neighbor and update the parents.

To create a B-Tree in PostgreSQL we can use the following command:

```
CREATE INDEX name ON table USING btree (column);
```

Since B-Tree is the default index we can also omit the *USING* clause.

10.5.4 Hash Table

Hash tables are a different kind of index where we try to achieve constant access time for point queries. What we hope for in hash functions is that the probability of collision is very small. Still because the probability is non-zero we have to deal with potential collisions. The way hash tables work is that we feed the tuple (or parts of it) into the hash function which then returns the index of a slot. There are different ways to approach collision handling:

- **Closed Hashing**
  In closed hashing we know how many elements we are trying to hash. This means that each slot stores the tuple itself or at most a single Record ID. A way to deal with collisions is the **Linear Probe**, where we first try to insert the element at the slot given by the hash function. If this slot is already occupied, we sequentially go to the next slot until we find a free one. This creates some issues when deleting entries as we have to maintain all the sequential hops with special values. Linear Probe is very cache-efficient but also very sensitive to the chosen hash function. Given that the hash function is reasonably good, this method performs adequately well. There are also other methods to deal with collisions in closed hashing:
  - Double Hashing - Offset the search of a free slot by a second hash value
  - Quadratic Probing - Offset the search of a free slot by $1^2, 2^2, 3^2, ...$
- **Open Hashing**
  In open hashing we do not know how many elements we are dealing with. Potentially even more than our hash table has slots. So instead of a single Record ID, we store a linked list of tuples or Record IDs. This technique is called **chained hashing**. If a tuple falls into an already occupied slot, the tuple is appended to the linked list. To retrieve a value, we have to search the given linked list.

10.6 Operator Execution

This layer executes relational algebra operators. It uses the different access methods from the layer below to implement these operators. To the upper layer it provides the ability to execute those operators.

10.6.1 Select

To implement the select operator $\sigma_C(R)$ for a predicate C we have two different methods.

- **Sequential Scan over Heap File**
  We bring the pages of the heap file into DRAM one-by-one. For each page, we scan all of its tuples and check for the predicate C. If the predicate evaluates to true, we output the tuple.
  To evaluate the I/O cost we assume that the relation R has $|R|$ tuples on $B(R)$ pages. The

buffer size of the system is M pages. The cost to read all pages is $B(R)$. The cost to write the tuples is $\alpha(C, R) \cdot B(R)$, where $\alpha(C, R)$ is a key constant called **selectivity**, which is the relative number of tuples in R that satisfy the condition C divided.
The total cost of a sequential scan is: $B(R) + \alpha(C, R) \cdot B(R)$.

- **Index Scan**
  When the predicate C is a check for equivalence or greater/less we can use the B-Tree index to check the condition.
  In an unclustered B-Tree we search for the right leaf node, scan the index and then fetch the corresponding tuple in the heap file.
  In a clustered B-Tree we search for the right leaf node and return the tuples along the way.
  To evaluate the I/O cost we assume the same metrics as above.

  For unclustered B-Tree:
  The cost to find the leaf is $\mathcal{O}(\log |R|)$. The cost fetch the tuples is $\mathcal{O}(\alpha(C, R) \cdot |R|)$ and to write $\alpha(C, R) \cdot B(R)$.
  The total cost is $\mathcal{O}(\log |R| + \alpha(C, R) \cdot |R| + \alpha(C, R) \cdot B(R))$.

  For clustered B-Tree:
  The cost to find the leaf is $\mathcal{O}(\log |R|)$. The cost to fetch the tuples is $\mathcal{O}(\alpha(C, R) \cdot B(R))$ and to write $\alpha(C, R) \cdot B(R)$.
  The total cost is $\mathcal{O}(\log |R| + 2 \cdot \alpha(C, R) \cdot B(R))$.

Whether we should use sequential scan or index scan depends on a lot of factors such as the relative speed of sequential to random access as well as the selectivity and whether the index is clustered or not. To factor this in and make an estimate on which mode is the fastest is the job of the DBMS.

10.6.2 Sort

To implement the select operator that sorts a relation R on an attribute A we have several options again. If we have a clustered B-Tree on A is is very straight forward. We simply scan the B-Tree leaf nodes and return the trivially sorted result.
If we have an unclustered B-Tree on A the job is equally simple but a lot slower. Since we still index the tuples in an ordered fashion we only have to read the tuple references in order. However, since the B-Tree is unclustered and for each tuple we have to chase a reference we have one random access per tuple in the worst case.

Sorting Tuples on Disk
In case we do not have a B-Tree index or want to sort the actual data on disk we might think about QuickSort. However, QuickSort is not designed to be I/O efficient and therefore doesn't have very good locality. Since we are dealing with block story, we need to worry more about locality. Whenever we read a page, we better take advantage of every tuple in that page.
This can be done with the **external sort** algorithm.
We assume to have N pages on disk to sort and a B-page buffer in memory. In the case that $N < B$, we simply load all the data into the buffer and sort it using QuickSort. In the more interesting case where $N > B$ we apply an algorithm that is similar to MergeSort.

- Sort each page individually
- Combine pages

The single pages can easily be sorted with QuickSort. With a 3-page buffer we can sort 2 sorted files. We simultaneously load the first elements of both files into the buffer, choose the smaller one between the too and output it to the third buffer. Then we take the next element from the file we chose the first tuple and repeat the process until both files are merged together.
The total cost of sorting is $\mathcal{O}((\log_{B-1}(\frac{N}{B}) + 1)N)$.

10.6.3 Join

- **Nested Loop Join**
  This is the most naive way to perform a join operation. We simply iterate through each

Algorithm 5 Nested Loop Join ($R \bowtie_\theta S$)

```
for each tuple r in R do
    for each tuple s in S do
        if θ(r, s) then
            append (r, s) to result
```

tuple combination with two for-Loops and check the condition. The cost of this operation depends on the join order and buffer size. For a buffer size of 3 the cost, disregarding the operations to write the result out is $B(R) + |R| \cdot B(S)$. Usually we want the smaller relation to be in the outer loop, however, if we have a large buffer ($M \geq B(S) + 2$ we can cache the smaller relation and want it in the inner loop.

- **Block Nested Loop Join**
  This version of the nested loop join performs loop unrolling to partition the relations into blocks and iterate over those instead of the entire relations. This increases locality and therefore performance by reducing I/O operations by up to 2 orders of magnitude. The

Algorithm 6 Block Nested Loop Join($R \bowtie_\theta S$)

```
for each block BR in R do
    for each block BS in S do
        for each tuple r in BR do
            for each tuple s in S do
                if θ(r, s) then
                    append (r, s) to result
```

cost of a join using blocked loops is with a buffer size of 3 is $B(R) + B(R) \cdot B)S)$. If we have a larger buffer of $M > 3$ we can partition S into $B(s)/(M-2)$ chunks which reduces our cost to $B(S) + B(R) \cdot (B(S)/(M-2))$.

- **Index Nested Loop Join**
  This version leverages the search index on a relation S.

Algorithm 7 Index Nested Loop Join

```
for each tuple r in R do
    for each tuple s in IndexScan(S, r, θ) do
        append (r, s) to result
```

The cost model of this is very simple, it is $B(R) + |R| \cdot C$, where C is the cost of a lookup in the index. This can be very efficient. Because you usually have the first few levels of a B-Tree in the buffer, you can retrieve a lot of tuples with very few I/O operations. However, this kind of join is restricted to operations like $=, >, <$.

- **Sort Merge Join**
  To conduct equivalence joins or joins with $<, >$ we can first sort both relations. Once we have sorted both relations we compare them similarly to external sort. We load the first page from both relations into the buffer, since both relations are sorted it is easy to find matches and discard tuples without matches.
  The cost of a sort merge join is $B(R) + B(S) + Sort(R) + Sort(S)$.

- **Hash Join**
  There is an even faster way to conduct equivalence joins using hash tables.

Algorithm 8 Hash Join $R \bowtie_\theta S$)

build hash table HT for R
for each tuple s in S **do**
  if $h(s)$ in HT **then**
    append tuples (r, s) where $\theta(r, s)$ to result

This algorithm is incredibly efficient and only passes both relations once, if the hash table fits into memory that is. The cost is $B(R) + B(S)$.

- **Grace Hash Join**
  If the hash tale does not fit into memory completely we can perform a grace hash join. We also create a hash table for R, that sadly is too large for the memory. Therefore, we partition the hash table HT as well as the relation S into 3 (or n) partitions using another hash function. With these partitions we then can perform the hash join. The cost of this is $3(B(R) + B(S))$ (or n when doing more partitions).

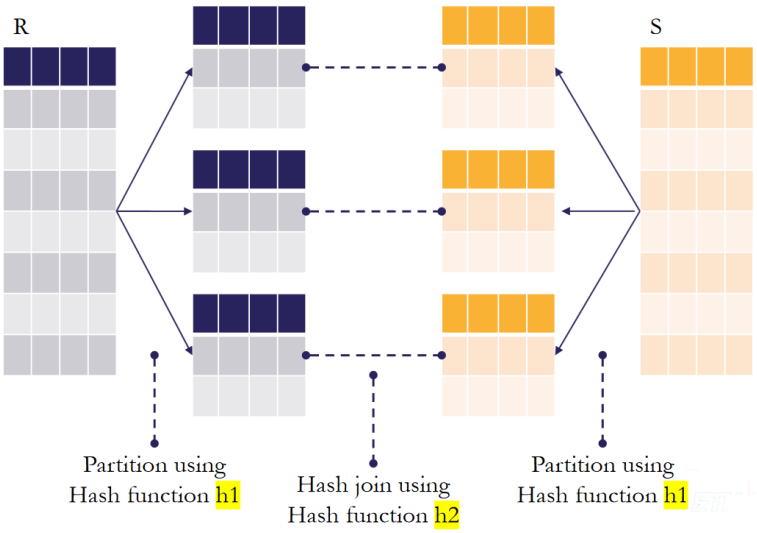


Figure 34: Grace Hash Join

As an overview again the different types of joins and on which predicates they can join.

Join Type	Join Predicates
• Nested Loop Join	• $\theta(r, s)$
• Block Nested Loop Join	• $\theta(r, s)$
• Index Nested Loop Join	• B-Tree: $r.a = s.b$, $r.a < s.b$, $r.a > s.b$; Hash Index: $r.a = s.b$
• Sort Merge Join	• $r.a = s.b$, $r.a < s.b$, $r.a > s.b$
• Hash Join	• $r.a = s.b$

10.7 Query Optimization

The query optimization layer should, given an SQL query, generate a good execution plan (a tree of realtional algebra ooperatiors). The fundamental questions of this layer are; How to run a physical plan, given that we know how to run each operator? How to search for the best physical plan?

10.7.1 Execution Model

The execution model is the way in which we put the operators together.

Iterator Model
In the iterator model, each operator is an iterator - it takes as input a set of streams of tuples and outputs a stream of tuples. A stream provides a *next()* interface. In this model the query plan is a tree of iterators. To get the result, we call *root.next*() over and over again. This model mimics a data flow from the bottom to the top.

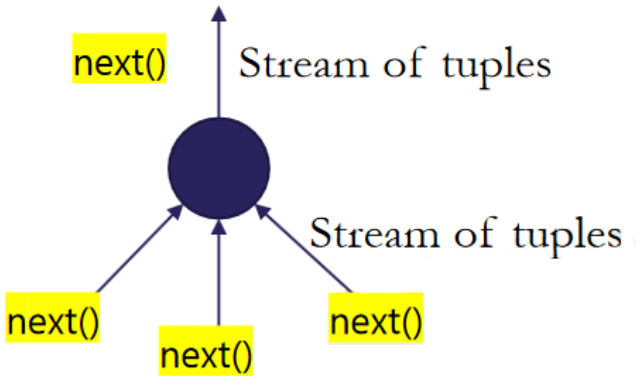


Figure 35: Iterator Model

This model is a very generic interface that hides the information very well and is easy to implement. There are is no overhead in terms of main memory and it supports pipelining, parallelism and distribution. However, it produces a high overhead of method calls and has poor instruction cache locality.

Materialization Model
In the materialization model, each operator processes its input all at once and emits its output all at once. This model is good when the intermediate results are not too much larger than the final result. OLTP is a good example workload while OLAP might not be best suited for this model.

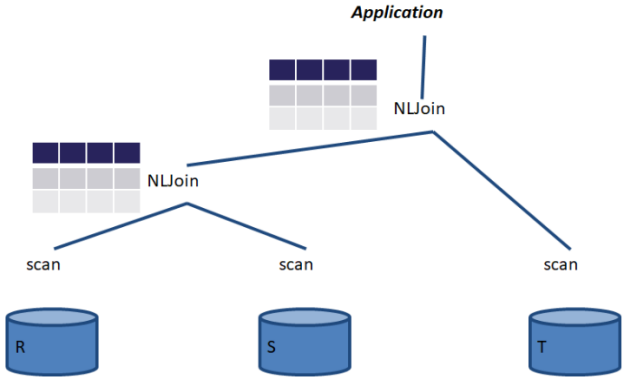


Figure 36: Materialization Model

Vectorization Model
The vectorization model operates similarly to the iterator model, but instead of returning a single tuple on invocation, the operator returns a batch of tuples. This greatly reduces the number of invocations per operator and allows for operators to use vectorized (SIMD, AVX) instructions to process batches of tuples, which makes this mode good for OLAP workloads.

10.7.2 Search Space

Given a logical plan as input, there are different ways that one can construct a physical plan. For example we could change the join order or execute selection at different times. We now define a set of transformations called **query rewriting rules** that will take as input a query plan and output an equivalent query plan.

- $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
  Conjunctive selection operations can be deconstructed into a sequence of individual selections.

- $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
  Selection operations are commutative.

- $\Pi_{t_1}(\Pi_{t_2}(E)) = \Pi_{t_1}(E)$
  Only the last in a sequence of projection operations is needed, the others can be omitted.

- $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
  $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
  Selections can be combined with Cartesian products and theta joins.

- $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
  Theta-join operations (and natural joins) are commutative.

- $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
  Natural join operations are associative.

- $\sigma_\theta(E_1 \bowtie E_2) = \sigma_\theta(E_1) \bowtie (E_2)$
  Selection can be pushed down, if θ only involves attributes in E_1.

- $\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$
  The projection operation distributes over the theta join operation if θ only involves attributes from $L_1 \cup L_2$.

- $E_1 \cup E_2 = E_2 \cup E_1,\ E_1 \cap E_2 = E_2 \cap E_1$
  $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3),\ (E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
  The set operations union and intersection are commutative and associative.

- $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$
  $\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$
  $\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$
  The selection operator distributes over $\cup, \cap$, and $-$.

- $\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$
  The projection operation distributes over union.

All these rules give us a way to generate physical plans.

10.7.3 Cost Model

The cost model provides a way to estimate the performance of a given physical query plan without actually running the query. We have already seen cost estimation for each operator. While some constants like number of pages or number of tuples are query-independent, other factors like selectivity are query-dependent. The core concept is **cardinality estimation**. We need to figure out how many tuples a query would return.
One way to estimate the number of tuples is by creating histograms over attribute. The cost is ammortized since we can use the histogram over many queries. However, this only works if the values are distinct. For continuous values the histogram needs to use bins. Another difficulty in estimation is in correlated queries. There the attributes are assumed to be independent which can give inaccurate answers. Another option is to build multi-dimensional histograms, but this is very expensive.

10.7.4 Search Algorithm

Now that we are able to generate semantically equivalent plans, the first idea is to enumerate all possible executions and choose the cheapest one. However, since queries can be reasonably complex, the search space is incredibly huge. There are already 4^n ways to join n relations.
A first approach could be to constrain the search space, i.e. limit the possible execution plans we look at. One way is to only consider left-deep join trees, because left-deep trees allow to generate fully pipelined plans where intermediate results don't have to be written to temporary files. The best plan in the limited search space can then be decided by dynamic programming but it is still very slow.
Another approach could be heuristic-based. Since cost-based optimization is very expensive, systems may use heuristics to reduce the number of choices that must be made in a cost-base fashion. Common heuristics are:

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

11 Transactions & ACID

What motivates the notion of transactions are concurrent database accesses and resilience to system failures. If we imagine a DBMS with a single thread that has to process two queries at the same time. The system could interleave the two executions which can lead to a variety of inconsistencies. Inconsistencies can be experienced on attribute-, tuple- or table-level or occur from multiple statements. One way to interleave the instructions is called a **schedule**.
The second motivation is the dealing with system failure. When the system fails during changes to the database, there are three possible states:

- None of the changes are in
- All of the changes are in
- Some of the changes are in

In database systems we want to avoid the last state. The way to avoid this are transactions. A transaction is a sequence of one or more SQL operations treated as a unit. Concurrent transactions appear to run in isolation (i.e. sequentially). If the system fails, each transaction's changes are reflected either entirely or not at all. A transaction is initiated with the keyword *BEGIN* and terminated with the keyword *COMMIT* or *ABORT*. If the transaction

commits, the changes of the transaction have been made persistent. If the transaction aborts, the database rollbacks and changes are undone.

```
BEGIN;
SQL
SQL
COMMIT;
```

```
BEGIN;
SQL
SQL
ABORT;
```

11.1 ACID

ACID stands for the desired properties of transactions

A Atomicity - a transaction is executed in its entirety or not at all

C Consistency - a transaction executed in its entirety over a consistent DB produces a consistent DB

I Isolation - a transaction executes as if it were alone in the system

D Durability - committed changes of a transaction are never lost - can be recovered

A naive way to implement transactions would be to execute the transactions one-by-one. Before *BEGIN*, we dump the whole database to a new file and before *COMMIT* we overwrite the original file. However, this is extremely slow and incurs huge overhead to copy the whole database every time. Luckily, this is not how current database systems work.

11.2 Atomicity

Atomicity requires transactions to be all-or-nothing, no transaction is left half done. The effects are either all in the database or not in the database at all.

11.3 Consistency

Consistency requires transactions to keep consistent states. Given all integrity constraints hold when the transaction begins, all constraints must hold when the transaction ends. During the transaction some constraints may be violated. However, serializability (isolation) implies that constraints always hold, if each transaction is correct.

11.4 Isolation

Isolation allows execution to be interleaved but the execution must be **equivalent to some sequential order** of all transactions. An execution that is equivalent to a serial execution is called **serializable**. It is possible that different serial executions lead to different results but that is acceptable from the DBMS perspective. If we need one possibility over another, this should be enforced by the application and not the DBMS.
Achieving isolation as defined above might be expensive and in some cases a weaker notion of isolation would suffice. Therefore, databases provide different isolation levels. Isolation levels are a property of each individual transaction. The isolation levels are defined by anomalous behaviors that are allowed in each level.

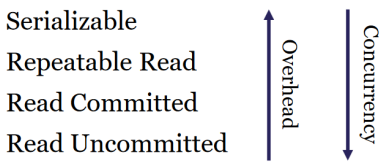


Figure 37: Isolation Levels

11.4.1 Read Uncommitted

The *read uncommitted* level allows a transaction to perform dirty reads. A read is dirty if it was written by a different uncommitted transaction. The dirty read might be a value that never exists in the database when the other transaction aborts.
e

11.4.2 Read Committed

The *read committed* level does not allow dirty reads, it is stronger than *read uncommitted* but still doesn't guarantee serializability. Another transaction could update and commit in the meantime and change values. The execution is therefore, not equal to any sequential execution.

Figure 38: Read Committed but not Serializable

11.4.3 Repeatable Read

The *repeatable read* level does not allow dirty reads and additionally, an item read multiple times cannot change value. This notion is stronger than *read committed* but still does not guarantee serializability. Another transaction could insert new tuples or change different attributes such.

Figure 39: Repeatable Read but not Serializable

11.5 Durability

Durability requires changes of transactions to be persistent in the database. If the system crashes after a transaction committed, all effects must remain in the database.

11.6 Formal Definition

A database is a fixed set of named data objects (e.g. A, B, C...). In practice, transactions can create new objects but here we assume that they cannot do that. A transaction is a sequence of read and write operations:

- $a \leftarrow R(A)$: read object A into variable a
- $W(B, b)$: write the value of variable b into object b
  We assume that the moment we write, the changes are reflected in the database.

A new transaction starts with *BEGIN* and stops with either *COMMIT* or *ABORT*. If the transaction commits, all changes are saved. If the transaction aborts, all changes are undone so ti is like the transaction never happened.

11.7 Conflict Serializability

The serializability of a schedule is not decided by solely it's I/O pattern. **Conflict serializability** is a stronger notion of serializability that only depends on the I/O pattern and is thus easier to handle for the DBMS.
We call a schedule conflict serializable if the schedule is conflict equivalent to some serial schedule. Two schedules are conflict equivalent if and only if

- They involve the same actions of the same transactions
- Every pair of conflicting actions is ordered in the same way

There are different types of actions that conflict:

- Trivial Conflicts - two different operations from the same transaction are always conflicting, i.e. we cannot reorder within transactions
- Read-Write Conflict - leads to unrepeatable reads
- Write-Read Conflict - leads to dirty reads
- Write-Write Conflict - leads to overwriting uncommitted data

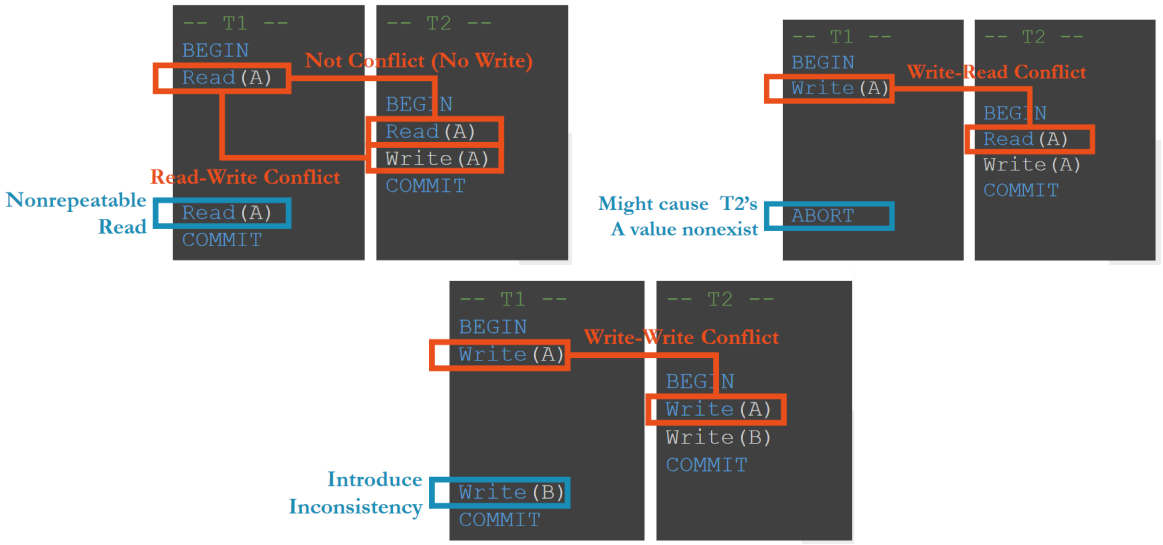


Figure 40: Different Types of Conflicts

Conflict serializable schedules are a subset of serializable schedules and it is enforced by most DBMS.

11.7.1 Decide Conflict Serializability

A naive way to check if a schedule is conflict serializable is to go from the definition and perform the swaps, however, there is a better way. We can construct a dependency graph where

- Each transaction is a Node in the graph.
- There is an edge from T_i to T_j if an operator o_i in T_i is in conflict with an operation o_j in T_j and o_j appears earlier than o_j.

A schedule is conflict serializable if and only if its dependency graph is acyclic.

11.8 Locking

A pessimistic approach to enforcing isolation is to proactively prevent non-serializable schedules from happening via locking. The idea is that before the system accesses the data object X, it locks X such that other transactions cannot access it at the same time and the system only releases the lock when it is safe to do so.
There are also different types of locks. For example, we could have shared and exclusive locks that are required for read and write operations respectively. Intuitively, multiple read operations can happen at the same time while writing needs exclusivity.

Current Lock

Request	No Lock	S	X
S	Grant	Grant	Deny
X	Grant	Deny	Deny

Figure 41: Lock Types

It is important to realize that not all locking strategies guarantee conflict serializable schedules. The following is not serializable, even though all Read/Write operations are protected by locks. This means we need to be careful about when to acquire and release locks.

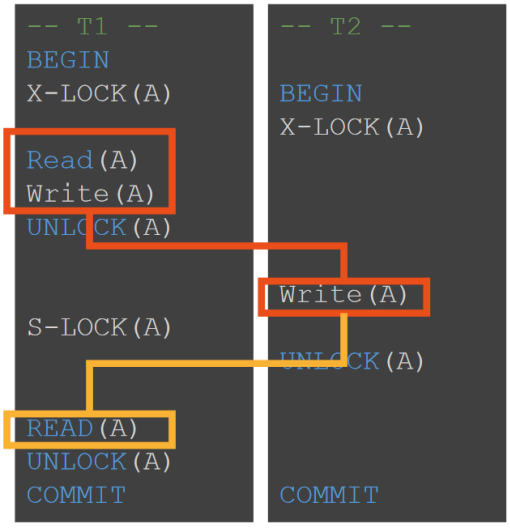


Figure 42: No Conflict Serializability

11.8.1 Two Phase Locking (2PL)

Two phase locking divides the locking strategy into two phases:

- **Growing Phase**
  - Each transaction requests the lock that it needs from the DBMS's lock manager.
  - It cannot release locks in the growing phase.
- **Shrinking Phase**
  - The transaction is only allowed to release locks that it previously required. It cannot require new locks in this phase.

The 2PL locking strategy guarantees conflict serializability, so it only generates schedules whose dependency graph is acyclic. The core problem of this strategy are cascading aborts. If one transaction aborts, it is possible the system has to abort other transactions as well because they have access to inconsistent state. This gets even worse because those other transactions could already be committed. The system would have to rollback an already committed transaction, which violates the durability property.

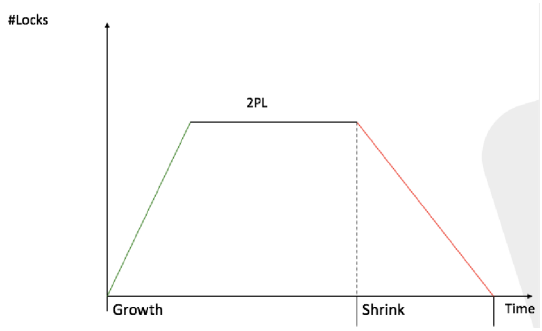


Figure 43: Two Phase Locking Strategy

11.8.2 Strict Two Phase Locking

The strict two phase locking strategy is very similar to the normal two phase locking. It is divided into the same phases, a growing and a shrinking phase. However, instead of releasing locks whenever they are not needed anymore, all the locks are kept acquired until the very end of the transaction, i.e. the commit or abort. This method solves the aforementioned problem with cascading aborts and ensures the durability property.
But there is a hitch, deadlocks can occur. To solve this problem the DBMS has to implement deadlock detection. The system may construct a wait-for graph where

- Each node is a transaction
- There is an edge from T_i to T_j if T_i is waiting for a lock currently held by T_j

If this graph has a cycle, the system deadlocked. The system then has to kill transactions. The process of choosing which transaction to kill may depend on the age, progress, number of locked items, number of rollbacks, number of times it has already been aborted and or how many deadlocks it can break.

11.8.3 Granularity of Locks

When using locks we have to ask ourselves at which granularity we want to do the locking. Does a query lock the entire table or does it hold a lock per tuple. In order to achieve correctness and reasonable performance, we need locks at multiple granularities to accommodate the hierarchical structure of the database. A naive strategy makes every database object (table, tuple, attribute, database) lockable. But such a strategy would hand out locks for a table if tuples in that table are already locked. To address this problem it would have to check the entire table whether there are locks, which is too slow and not even necessarily correct (locks granted while checking). Instead we introduce three new types of locks

- **IS** - Intention Shared
  - Some lower nodes are in shared
- **IX** - Intention Exclusive
  - Some lower nodes are in exclusive

- **SIX** - Shared and Intention Exclusive
  - Root is locked in shared
  - Some lower nodes are in exclusive

When the system wants to lock Tuple1 in S it starts from the root:

- Acquire IS on Database
- Acquire IS on Table1
- Acquire S on Tuple1

With this we can implement the different isolation levels.

- Serializable - Strict 2PL + Index Lock
- Repeatable Read - Strict 2PL
- Read Committed - S locks are released immediately
- Read Uncommitted - No shared lock

11.9 Isolation beyond Locking

11.9.1 Timestamps

A more optimistic approach to guarantee isolation is to assume non-serializable behavior is reasonably rare so we just hope for the best and rollback if anything goes wrong. The idea is to pre-define a serial order and roll back if transactions do not adhere to it. We assign a timestamp $TS(T_i)$ to each transaction T_i. If $TS(T_i) < TS(T_j)$ then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where T_i appears before T_j.
Each database object is also assigned with a timestamp:

- $RT(X)$: the read time of X - highest timestamp of a transaction that has read X
- $WT(X)$: the write time of X - highest timestamp of a transaction that has written X

The protocol is very simple. Each transaction reads or writes objects without locks. Each object is then tagged with the timestamp of the last transaction that read or wrote it. Now if a transaction T_i tries to access an object that has a timestamp higher than $TS(T_i)$, i.e. has been written by a transaction that should execute after T_i we abort and restart.

Schedules generated with this protocol are conflict serializable and deadlock free. However, performance could suffer if we are optimistic when we ought not to. Long transactions could experience starvation and cascading aborts could occur.

11.9.2 Snapshot Isolation

In the snapshot isolation, each transaction T receives a timestamp $TS(T)$ when it starts. All reads are carried out with with the DB state at the time $TS(T)$ and all writes are carried out in a separate buffer. When a transaction commites, the DBMS checks for conflicts. If there exists a transaction T_2 such that T_2 committed after the start of T_1 at time $TS(T_1)$ and before T_1 commits, and T_1 and T_2 update the same object, the transaction T_1 is aborted.
The good thing about this is that writers and readers do not block each other. However, it also introduces problems. The overhead it introduces is manageable since we only need to keep a write-set per transaction and aborts can be implemented very efficiently. Besides that, keeping all versions of an object is often useful anyway.

One problem that occurs is write skew. The checking of integrity constraints also happens in the snapshot and it can happen that two concurrent transactions update different objects, both of which conform to the constraints, but the combination of the two does not. This also implies that snapshot isolation is not fully serializable. Also, there may not be any deadlocks but it introduces unnecessary rollbacks.

11.10 Recovery Theory

There are several scenarios in which we have to recover the state of the database. Each of them requires different actions.

- Abort of a single transaction
  - Recovery: Undo a single transaction
- System crash: lose main memory, keep disk
  - Recovery: Redo committed transactions
  - Recovery: Undo active transactions
- System crash with loss of disk
  - Recovery: Read backup of DB from tapes

What we are interested in are the first two cases.

11.10.1 Recoverability of Schedules

Given a schedule, we say T_1 reads from T_2 if T_1 reads a value written by T_2 at a time when T_2 was not aborted. There are different families of schedules, each of which has different recoverability properties.

- **Recoverable Schedule (RC)**
  - If T_i reads from T_j and commits, then $c_j < c_i$, where c_i is the commit time of T_i
  - No need to undo a committed transaction
- **Avoid Cascading Abort Schedule (ACA)**
  - If T_i reads X from T_j, then $c_j < r_i[X]$, where $r_i[X]$ is the time T_i reads X
  - Aborting a transaction does not cause aborting others
- **Strict Schedule (ST)**
  - If T_i reads from or overwrites a value written by T_j, then $(c_j < r_i[X]\ AND\ c_j < w_i[X])$ or $(a_j < r_i[X]\ AND\ a_j < w_i[X])$, where a_j is the abort time of T_j
  - Undoing a transaction does not undo the changes of other transactions
  - Strict 2PL ensures both **serializability** and **strict recoverability**

Recovery is important because it ensures that the state of the DB is correct. Problems with recovery are very expensive.

- **Not RC:** I run a program, get some data, finish the program and issue a report. The data read was later removed because another program aborted.
- **Not ACA:** thrashing behavior when transactions keep aborting each other (one program makes no progress because of other programs).
- **Not ST:** recovery after a failure becomes very complex (or impossible).

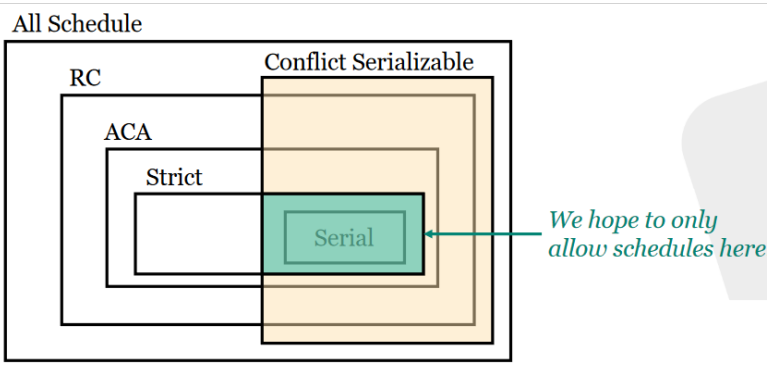


Figure 44: Allowed Schedules

11.10.2 Write-Ahead Log

Now we look at how to implement recovering transactions. We do this by creating a log file that is open for appending only. This log is in memory and has to be flushed to disk.

- Disk (Secondary Storage) is safe.
- $Write(A, v)$ - does not reflect on disk, it only changes the version in the buffer
- $OUTPUT(A)$ - reflects the changes in the buffer on disk

The operations we can do on a log file are the following:

- Append Record (only changes in-memory part)
  - $START\ T$
  - $COMMIT\ T$
  - $ABORT\ T$
  - $Update\ < T, X, v >$
  - These are only log entries, they do not change anything in the actual database.
- Flush Log (will be reflected on disk)
  - $FLUSH$ - flush log to disk

We will now look at different types of logging strategies.

Undo Logging

Undo logging is a simple strategy where before the transaction T modifies the database element X, it writes $< T, X, \text{old value} >$ to the log on disk before the change of X is written to disk. And if the transaction commits, a $COMMIT$ entry is written to disk after **all other changes are written to disk.**

All committed transactions have a $COMMIT$ record in the log file on disk. This entry is only written after all changes have been written to disk, which makes this transaction completely reflected on disk.

Uncommitted transactions do not have a $COMMIT$ entry in the log file on disk. This means we find a $START\ T$ in the log without a $COMMIT\ T$. If there is only a single transaction we undo the changes with the log and write $ABORT\ T$ at the end of the log and flush it. If there are multiple uncommitted transactions, we scan from the end and undo the changes while skipping those updates made by committed transactions. We then write $ABORT\ T$ for the uncommitted transactions and flush the log. Since we enforce a strict schedule (ST), we do not have to worry about updates of uncommitted transactions being overwritten by committed

transaction.

A problem with undo logging is that it incurs a lot of I/O operations before the commit. Before we can commit, we need to flush out all changes to the database and all undo logs. This can be very expensive.

Redo Logging
The idea of redo logging is not to undo uncommitted transactions but to redo committed transactions in case of failure. If the transaction T modifies the database element X, it writes $< T, X, \text{new value} >$ to the log. Before any modification on disk happens, it appends a $COMMIT\ T$ and flushes the log to disk.
If there is no $COMMIT\ T$ in the log, this means that no changes of T appear on disk. We write $ABORT\ T$ to the log.
If there is a $COMMIT\ T$ in the log, this does not mean its changes are reflected on disk, but we can redo those changes using the log. We scan the log forward from the beginning and for each update record we write the change to disk.

The problem with redo logging is that the log we need to keep can become very large, since all changes have to be reflected and a $COMMIT$ entry does not guarantee the changes are reflected in the database. A potential solution to this is to take checkpoints and redo changes from this checkpoint.
Another central problem of redo logging occurs because it cannot make changes to the disk before the commit record is flushed out. But what if:

- T_1 makes changes on A
- T_2 makes changes on B
- A and B are on the same page P
- T_1 commits

If we now flush out the page P, both changes on A and B will be reflected on disk, but changes to B cannot be reflected on disk until B commits.

Undo/Redo Logging
Undo/Redo logging tries to tries to address the above issue. Before a transaction T modifies any database element X on disk, we write $< T, X, \text{old value}, \text{new value} >$ into the log file. The log has to be flushed before actual changes are made on disk.
If there is no $COMMIT\ T$ in the log, the transaction is incomplete and it is undone with the log.
If the commit record is in the log, the transaction is complete and we redo it.

12 Distributed Databases

12.1 Distributed Commit

If we want to build an application where we have to commit into multiple distributed databases we experience a problem. Single node atomicity does not imply distributed atomicity. This means even if every node ensures atomicity itself, the atomicity of the global transaction is not guaranteed. We need to design a protocol for distributed commits.

12.1.1 Two Phase Commit

The commit process is split into two phases.

- **Voting Phase** - The coordinator ask all nodes to be ready and willing to commit.
  - Workers an say NO. But if they don't say NO, they cannot say NO in the next stage, i.e. they have to be able to commit.
- **Decision Phase** - The coordinator tells all nodes to commit or abort
  - If all workers are OK, commit
  - If one worker is not OK, abort

There are three types of failure that can occur:

- All machines fail
- Workers fail
- Coordinator fails

This protocol ensures global atomicity of the tansaction. The different types of failures are handled with log entries, timeouts and notifications that a machine was down.

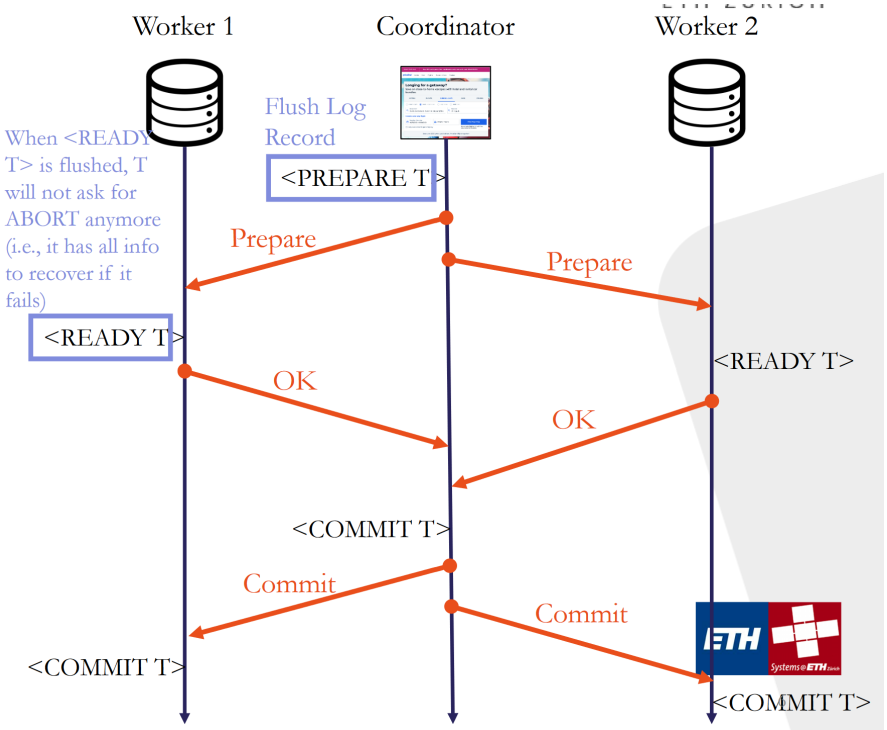


Figure 45: Two Phase Commit

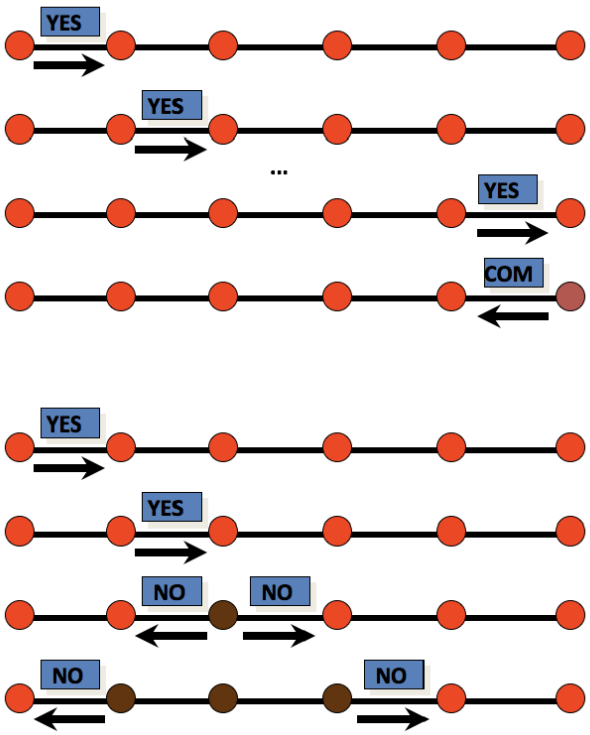
There are also different flavors to the two phase commit to change the amount of messages the coordinator has to send and receive. One option is to do a **linear two phase commit** where the messages are passed from one worker to another, this reduces the total amount of messages but increases the latency.

Two Phase Commit

- Given 1 coordinator and N workers
- Total messages - $3N$
- Total latency - $3t$, if t is the latency to pass one message.

Linear Two Phase Commit

- Given 1 coordinator and N workers
- Total messages - $3N$
- Total latency - $2Nt$



(a) Linear Two Phase Commit

12.2 Distributed Query Processing

There are several reasons why we might want to have a distributed database. The main ones are speed of computation and size of storage. There are different ways to distribute a database:

- Shared Memory
- Shared Disk
- Shared Nothing

In this lecture we focus on the shared nothing architecture, where the master receives the queries and coordinates the optimization, planning, concurrency control, logging and recovery. Each worker is a single-machine database and uses everything we have covered so far to answer queries. The basic idea is to partition the database to each worker and each worker only works on its partition. There are several ways to partition the database:

- **Naive Table Partitioning** - Each worker receives an entire relation. In this scheme the workers can process the relations concurrently and if the output relation is small so is the communication between workers. However, the amount of queries that can be parallelized in this way is actually quite limited under this model and we run into problems if one table does not fit into a single machine.
- **Horizontal Partitioning** - Horizontal partitioning splits up relations some or all relations to multiple machines. The partitioning can be done in different ways.
  - Example 1: One table is replicated at every node while the other table is partitioned to the workers. Each node then joins its local data and sends their results to a coordinating node.
  - Example 2: Both tables are partitioned on the join attribute. Each node then performs the join on the local data and sends to a coordination node.

- Example 3: Both tables are partitioned on different attributes. The node then broadcasts the table to all nodes.

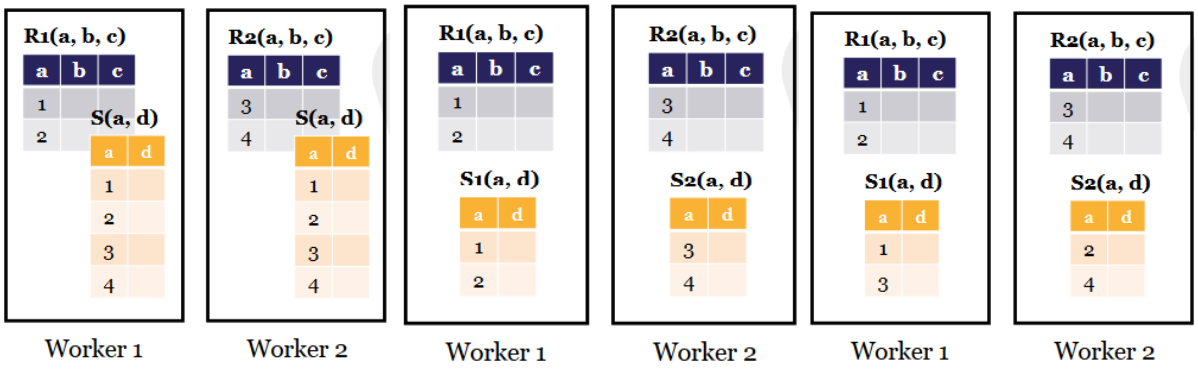
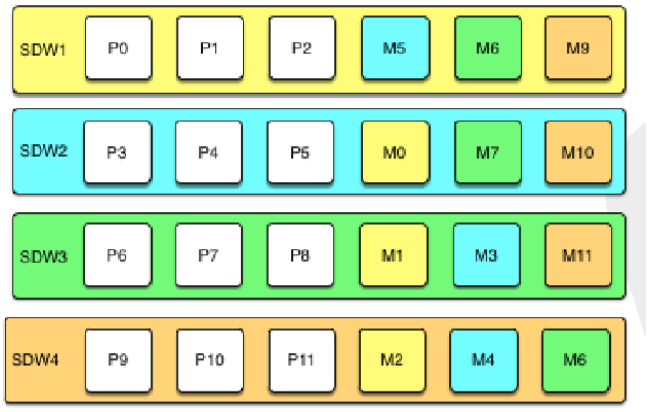


Figure 47: Partition Examples 1, 2 & 3

If we have multiple machines we have to deal with eventual crashes. One idea is to implement redundancy and store relations on multiple machines.
One way to do this is **group mirroring** where groups of partitions are copied to other machines as well. Since entire groups are mirrored, the failure of one machine will double the load on another, also if a combination of machines fail you are going to lose data.
Another option is to do **spread mirroring**. Groups are not mirrored as a whole but in segments. This means that a failure will only lead to a $1/N$ increase of load on all the other machines, which is more efficient than group mirroring. However, this is very sensitive to scenarios where more than two machines fail.

(a) Group Mirroring



(b) Spread Mirroring

12.3 Distributed Key-value Store

A key-value store is an approach to achieve a faster and more scalable architecture by supporting less. There are only two operations, get and set. The data model is very simple, we store a key and data (or pointer to data) and have an index on the key. Since we only have one store, we can partition it horizontally and replicate the partition. The replication strategy quorum (simple majority) with asynchronous replication across all copies (eventual consistency).
The key-value store provides very fast lookups and is easy to scale to multiple machines. However, it is not easy to support anything beyond point queries and often data is inconsistent across different replicas.

12.3.1 Consistent Hashing

If we want to partition our key-value store to different machines we implement consistent hashing. We consider the image space of our hash function to be a ring. We then hash all the data as well as a machine identifier onto this ring. The data is stored on the machine who's hashed identifer is in clockwise direction of the data hash, i.e. the nearest machine with a higher identifier hash.
One caveat of this method occurs when a machine leaves (not crashes) the distributed store. In this case we have to move the data to the next machine on the ring. However, this means that one machine has to take all the data that has previously been on another machine.
An approach to balance the data when a machine leaves is to assign multiple *virtual identifiers* to a machine. Those are then all hashed an put on the ring. Data distribution still works the same, but because there are now multiple positions on the ring where data is mapped to the machine, when it leaves we expect (statistically) the data to be distributed among the other machines.

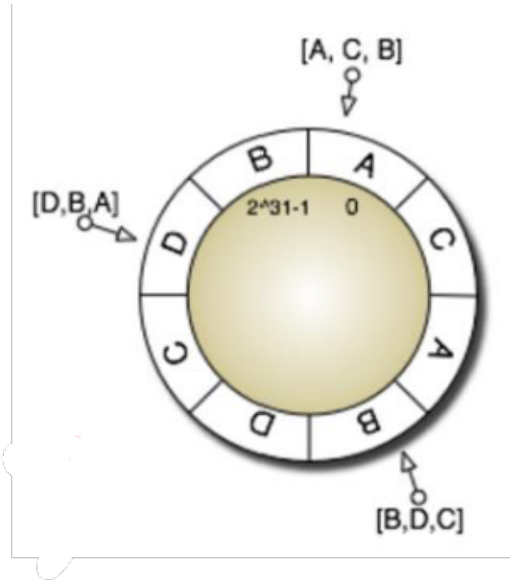


Figure 49: Consistent Hashing

This only works if a machine leaves, but what happens if it crashes. To avoid data loss we have to establish some kind of redundancy. One way to do this is to store the data on the next N machines in the clockwise direction. This does not completely guarantee consistency though. Assume the master copy (first machine) is replicated to the other machines ever t minutes. If the master node goes down, it is possible for a transaction to read a stale copy of the data.
To tackle this problem and guarantee consistency we implement the quorum. We define the following:

- N: The number of replicas for the data.
- R: The number of machines required for a read operation.
- W: The number of machines required for a write operation.

We can guarantee consistency if we ensure $R + W > N$. If we want to write data, we have to do so on at least W machines, this means after the write W machines have the latest version of the data, while the other $N - W$ machines might not have the newest version yet. If we then want to read this data, we have to contact at least N machines that have to agree on the value of the data. The pigeonhole principle guarantees, that among those N machines there is at least one of the W machines that have the newest version, which makes it impossible to read stale values.
In write heavy systems we can set $W = 1$ and $R = N$, which processes writes very fast, while blocking reads. In read heavy systems with strong consistency requirements we can set $W = N$ and $R = 1$, which processes read requests very fast and blocks writes. Such a system guarantees retains data as long as less than W machines fail. It is possible, if W machines fail that the data has not been replicated to any other nodes yet and the data is lost.