Data Modeling and Databases Summary FS22

Danny Camenisch

June, 2022

Contents

Data Modeling

1. Introduction

A `database` is a collection of data, for example information about bank accounts or data on your favorite online service etc.

A `database management system (DBMS)` is software designed to assist in maintaining and utilizing large collections of data.

1.1 Database Management System

We have several “whishes” for 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
- **Transactional Access:** as if there is only a single users using a system that does not fail
- **Generic Abstraction:** Users do not need to worry about all the above issues for each new query

What’s the **potential downside** of using a DBMS?

- **Worklaod Mismatch:** maybe your specialized application is not what a certain DBMS is designed for
- **Data Model Mismatch:** maybe your application cannot be naturally modeled by a given DBMS

Historically, there have been different models for DBMS. In this course we will focus on the **relational model**. But this should give a small overview of the other two models.

Hierarchical Model

The database schema follows a strictly hierarchical structure. It is basically structured like a tree and a query is made by traversing the tree, one record at a time.

Limitations:

- A hierarchical model might be too restrictive and forcing a non-hierarchical app into a tree-based model might cause problems such as data redundancy.
- Does not provide data independency, requires application changes when the physical data representation is changed.
- Requires manual query optimization.

Network Model

The database schema is a network of nodes, where each node represents one record. A query is made by traversing the network, one record at a time.

Limitations:

- Does not provide data independency, requires application changes when the physical data representation is changed.
- Requires manual query optimization.

2. Relational Model

The relational model focuses on representing knowledge as a collection of facts in the form of tabluar data, where the columns are our attributes. Inference is done using mathematical logic.

2.1 Schema

A `database schema` is a set of relation schema, where a `relation schema` is defined by a name and a set of attributes/fields/columns. A `field` or `attribute` is defined by a name and a `domain`, e.g. Integer, String, etc.

For example:

```
Students(sid:string, name:string, login:string, age:int, gpa:float)
```

2.2 Instances

For a relation $R(f_1 : D_1, ..., f_n : D_n)$, an `instance` I_R is a set of tuples: $I_R \subseteq D_1 \times \cdots \times D_n$. Inutitively, an instance is the "content" of a relation. It is important to remember that a relation instance is a **set**, this means we cannot have duplicated tuples and that the order of tuples doesn't matter.

2.3 Keys

There are many constraints that we can introduce on our database schemas. However, the key constraint is the most important one.

A `candiate key` is the minimal set of fields that identify each tuple uniquely (different candidate keys can have different amount of attributes). A `primary key` is one candidate key, marked in a schema by underlining. Every relation must have a key.

3. Query Languages

A `query` is a function that takes as input a DB instance and outputs a relation. A `query language` is a set of functions that you can express in that language.

3.1 Relational Algebra

We can query a DB in an `imperative` way. Relation instances are sets, so we query them using set operations.

Union: $\cup$

$$x \in R_1 \cup R_2 \Leftrightarrow x \in R_1 \vee x \in R_2$$

Difference: $-$

$$x \in R_1 - R_2 \Leftrightarrow x \in R_1 \wedge \neg(x \in R_2)$$

Intersection: $\cap$

$$R_1 \cap R_2 = R_1 - (R_1 - R_2)$$

Selection: σ_c - Return tuples which satisfy a given condition c.

$$x \in \sigma_c(R) \Leftrightarrow x \in R \wedge c(x) = True$$

Projection: $\Pi_{A_1,...,A_n}(R)$ - Only keep a subset of columns.

Cartesian Product: $\times$

$$(x, y) \in R_1 \times R_2 \Leftrightarrow x \in R_1 \wedge y \in R_2$$

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

Natural join: $\bowtie$

$$R_1(A,B) \bowtie R_2(B,C) = \Pi_{A,B,C}(\sigma_{R_1.B=R_2.B}(R_1 \times R_2))$$

If there are `no shared attributes` in a natural join, e.g. $R(A,B,C)$ and $S(D,E)$, then $R \bowtie S = R \times S$. If two relations `share all attributes`, then $R \bowtie S = R \cap S$.

Theta Join: $\bowtie_\theta$

$$R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$$

Equi-Join: $\bowtie_{A=B}$

$$R_1 \bowtie_{A=B} R_2 = \sigma_{A=B}(R_1 \times R_2)$$

Semi-Join: $\ltimes_c$

$$R_1 \ltimes_c R_2 = \Pi_{A_1,...,A_n}(R_1 \bowtie_c R_2)$$

Division: $\div$

$$R \div S = \Pi_{R-S}R - \Pi_{R-S}((\Pi_{R-S}R) \times S - R)$$

This last operation produces a relation consistent of tuples from R that match the combination of **every** tuple in S.

It is important to note that in real-world databases relational algebra uses `bag semantics` instead of set semantics:

- Each relation is a bag of tuples
- You can have duplicated tuples in the same relation
- i.e. set: {1, 2, 3}, bag: {1, 2, 3, 1, 2, 1}

It is furthermore important to remember that `bag operator semantics` are different to set operator semantics:

- *Bag Union:* $\{1,\ 2,\ 1\} \cup \{1,\ 2,\ 3\} = \{1,\ 1,\ 1,\ 2,\ 2,\ 3\}$
- *Bag Difference:* $\{1,\ 2,\ 1\} - \{1,\ 2,\ 3,\ 3\} = \{1\}$

3.2 Relational Calculus

Instead of using set operations, we can see relations as facts and use logic to query them.

We now have a query language that queries data in a `declarative way` - it tells the system *what* we want, instad of *how* to get it.

Formal Definition for Relational Calculus

We introduce the following formal definitions:

- 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: $dom = \cup_i D_i$
- Instance of Relation: $I_R \subseteq dom^n$
- Instance of DB: $\mathbb{I}$ a function that maps R_i to and instance of R_i, i.e. $\mathbb{I}(R_i)$

We can then define the syntax as follows: - Let ϕ be a first-order logic formula with free variables $x_1, ..., x_k$, then $Q_\phi = \{(x_1, ..., x_k) \mid \phi\}$ is a `domain relational calculus` query.

And we define the semantic as follows:

- Each *relation* R corresponds to a predicate R in ϕ
- Each *instance* I corresponds to a first-order interpretation $\mathbb{I}$
- An *assignment* is a mapping $\alpha : var \to dom$

Therefore the `answer of` Q `over` $\mathbb{I}$ is:

$$Q(\mathbb{I}) = \{(\alpha(x_1), ..., \alpha(x_k)) \mid \mathbb{I},\ \alpha \vDash \phi\}$$

Safe and Unsafe Queries

Let Q_ϕ be a relational calculus query. Then we say Q_ϕ is `safe`, if $Q_\phi(\mathbb{I})$ is finite for all instances $\mathbb{I}$.

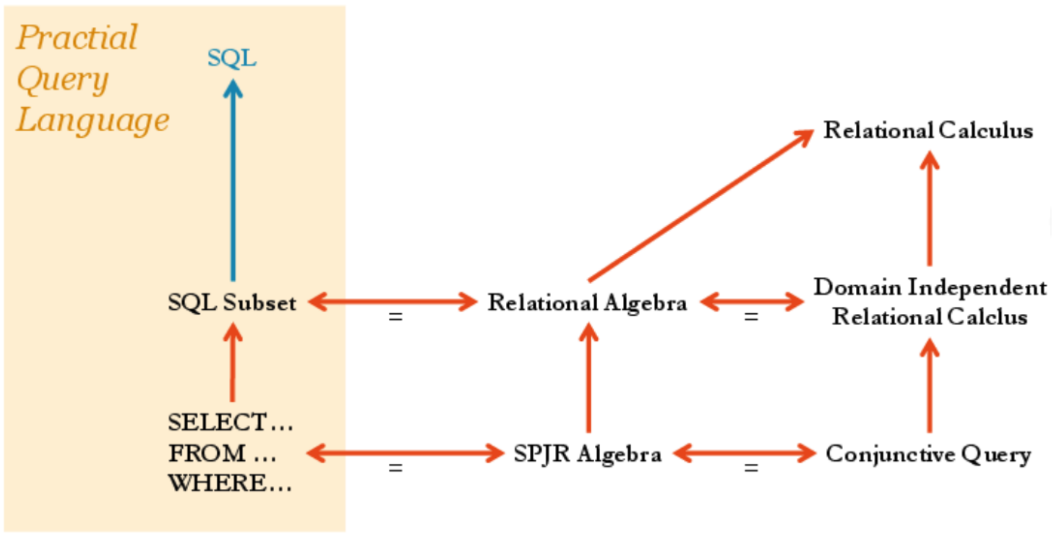
We do not want our DB to output infinite answers and as the problem whether a query is safe or not is undecidable, we introduce `domain independent relational calculus`.

A query where the answer is not only dependent on the DB instance, but also on the domain is called domain dependent. We try to turn a RC query Q_ϕ into something domain independent. For this we introduce active domain semantics:

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

where $adom(\phi, \mathbb{I})$ are all constans in $\mathbb{I}$ and Q_ϕ

2.3 RA & RC



From a theory side, we can see that the Relational Calculus is more powerful than the Relational Algebra, since every RA query can be turned into a RC query. This does not hold the other way around.

Conjunctive Query

Domain independent relational calculus is still tremendously difficult to analyze. The problems of equivalency and satisfiability are undecidable! We try to further restrict our query language such that checking these properties is easier.

Conjunctive Query:

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

CQ is as expressive as RA with only Selection, Projection, Join and Renaming (SPJR Algebra).

3. SQL (Structured Query Language)

`SQL` is a familiy of standards:

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

3.1 Data Definition Language

DDL provides statements to define the schema. In SQL, you need to provide a name, a set of columns, and their types. Example:

```sql
CREATE TABLE Professor(
    PersNR integer,
    Name varchar(30),
    Level character(2) default "AP",
    PRIMARY KEY (PersNR)
);
```

We `delete` a relation with the `DROP` keyword:

```sql
DROP TABLE Professor;
```

We `modify` a table with the `ALTER` keyword:

```sql
-- add a column
ALTER TABLE Professor ADD COLUMN (age integer);

-- delete a column
ALTER TABLE Professor DROP COLUMN age;
```

3.2 Data Manipulation Language

While we can manualy populate a DB, in reality it is often done automatically.

```sql
-- insert values
INSERT INTO Student (PersNr, Name)
VALUES (28121, 'Frey');

-- delete values
DELETE Student
WHERE Semester < 13;

-- update values
UPDATE Student
SET Semester = Semester + 1;
```

3.3 Query Language

Nearly all queries follow the form `SELECT ... FROM ... WHERE ...`. We can put this into relational algebra the following way:

SQL

```sql
SELECT PersNr, Name
FROM Professor
WHERE Level = 'FP';
```

Relational Algebra

$$\Pi_{PersNr,\,Name}(\sigma_{Level="FP"}Professor)$$

It is important to note, that **every RA expression can be written in SQL subset:**

	Relational Algebra	SQL
Union	$R_1 \cup R_2$	`(SQL1) UNION (SQL2)`
Difference	$R_1 - R_2$	`(SQL1) EXCEPT (SQL2)`
Selection	$\sigma_c(R)$	`SELECT * FROM (SQL1) WHERE c;`
Projection	$\Pi_{A_1,\ldots,A_n} R$	`SELECT A1,..., An FROM (SQL1)`
Cross Product	$R_1 \times R_2$	`SELECT * FROM (SQL1), (SQL2);`
Rename	$\rho_{a,b,c} R$	`SELECT A as a,..., C as c FROM (SQL1);`

Sorting

By default tables are not sorted, if a result needs to be sorted, the query explicitly needs to specify this.

```
SELECT PersNr, Name, Level
FROM Professor
ORDER BY Level DESC, Name DESC;
```

Grouping

Group table by different values for attributes with the `GROUP BY` keyword, then aggregation functions (`COUNT, SUM, AVG, ...`) can be used for each group.

```
SELECT Level, COUNT(*)
FROM Professor
GROUP BY Level;
```

When using `GROUP BY` we can't use `WHERE` afterwards. To achieve the same affect we can use `HAVING`.

3.4 Known Unknowns and Incomplete Information

One way to model incomplete information is to place the values that we don't know with a special state `NULL`. It is important to note, that NULL represents a *state*, not a value.

Operations Over `NULL`

Arithmetic:

```
(NULL + 1) -> NULL
(NULL * 0) -> NULL
```

Comparisons:

```
(NULL = NULL)  -> Unknown
(NULL < 13)    -> Unknown
(NULL > NULL)  -> Unknown
(NULL = NULL)  -> Unknown
(NULL IS NULL) -> True
```

When aggregating, all NULLs form a single group.

Where can `NULL` come from?

`NULL` mostly appears when inserting incomplete data or applying (right / left) outer joins.

3.5 Views

Views aim at raising the level of abstraction. In a DB, a view is the result set of stored query on the data.

```
CREATE VIEW <NAME_OF_VIEW> AS <SQL_QUERY>
```

After creating a view, it can be used like a table. But there are some key differences in using them, one of them is that views are logical, it's just an "alias" for a query. If data in the underlying table gets updated, the view will be updated as well.

We can update a view, but only if the following conditions are met:

- View involves only one base relation
- View involves the key of the base relation
- View does not involve aggregates, goupby or duplicate-elimination

This is needed to maintain a one-to-one mapping between the view and the base relation.

3.6 Recursion

Assume we want to answer a query of the following form: *Select all ancestors of person A*. For this we would need to execute the same query again and again until it converges. SQL provides an easy way to express such a query:

```
WITH RECURSIVE R AS (
    <BASE_QUERY>
    UNION
    <RECURSIVE_QUERY>
)
<QUERY_INVOLVING_R>
```

Example:

```
WITH RECURSIVE Ancestors(ancester) AS (
    SELECT parent
    FROM ParentOf
    WHERE child = A
    UNION
    SELECT p2.parent
    FROM Ancestors p1, ParentOf p2
    WHERE p1.ancester = p2.child
)
SELECT * FROM Ancestors;
```

Recursion can be tricky, as it could lead to non-termination.

3.7 Constraints

Sometimes we want to constrain the set of valid DB instances, we already have seen some constraints we can put in a DB.

```
CREATE TABLE <NAME_OF_TABLE> (
    <ATTR_NAME> <ATTR_TYPE> <ATTR_CONSTRAINT>,
    ...
    PRIMARY KEY (<ATTR_NAME>)
);
```

- `NOT NULL`: An attribute cannot be NULL.
- `UNIQUE`: An attribute must be unique.
- `CHECK`: An attribute must satisfy a certain condition.
- `FOREIGN KEY`: An attribute must be a key of a different table.
- `PRIMARY KEY`: An attribute must be a key of a table.
- `DEFAULT`: Sets a default value for an attribute.

Using `PRIMARY KEY (<COLUMN_NAME>)` we can also select multiple attributes to be the primary key.

A `FOREIGN KEY` refers to a tuple from a different relation. If a foreign key gets updated, there are multiple ways to deal with this:

- `CASCADE`: The update of the foreign key will be propagated to the other relation.
- `SET NULL`: The value of the foreign key will be set to NULL.
- `RESTRICT`: Prevents deletion of the primary key before trying to do the change, causes an error.
- `NO ACTION`: Prevents modification after attempting the change, causes an error.

4. Entity-Relationship Model

4.1 Conceptual Modeling, Logical Modeling, and Physical Modeling

The process of implementing a real-world application includes modeling a DB. Modeling a DB goes through the following stages:

1. *Conceptual Modeling:* Capture the domain to be represented

2. *Logical Modeling:* Mapping the concepts to a concrete logical representation
3. *Physical Modeling:* Implementation in a concrete hardware

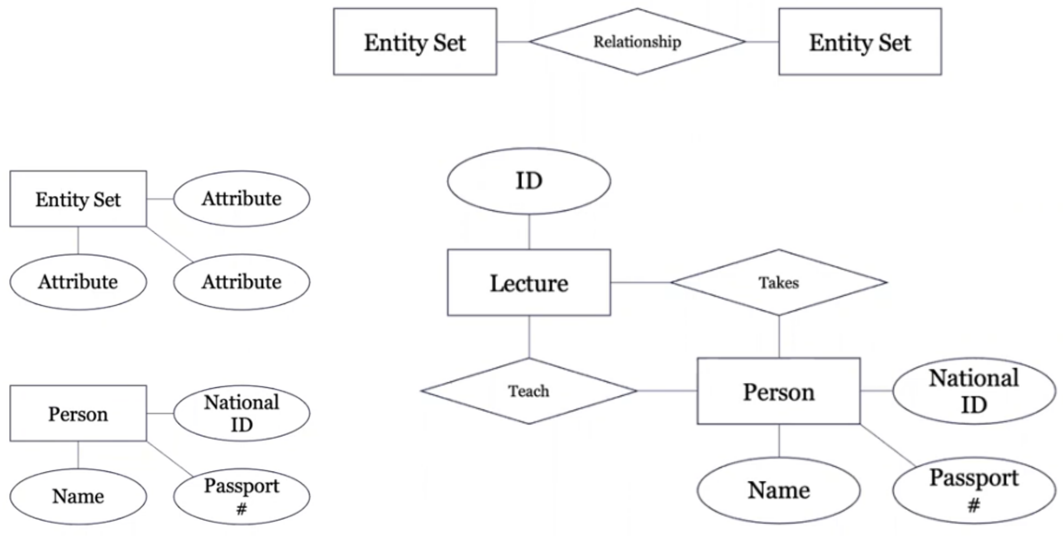
4.2 Conceptual Modeling using Entity-Relationship Model

Basic Concept

An `Entity-Relationship Model` models an application as a graph with the following three element types:

- `Entity sets` : A set of similar entities. "*Similar*" means that entities in the same entity set share the same attributes (e.g. "Professor" is an entity set, "ProfA" is an entity).
- `Attributes` : Properties of entities (e.g. ID and name of a professor).
- `Relationships` : Connections among two or more entity sets (e.g. relationship between professor and lecture).

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



`Primary keys` are underlined in an ER-Diagram.

Formal Semantics of ER-Diagram

An ER-Diagram is a constraint language, defining the set of *valid DB instances.* All the values the DB can take is given by $\mathcal{D} = \mathcal{B} \cup \Delta$, where: - $\mathcal{B}$: concrete values (Int, String, etc.)
- Δ : abstract values (corresponding to an entity)

We can then furthermore define:

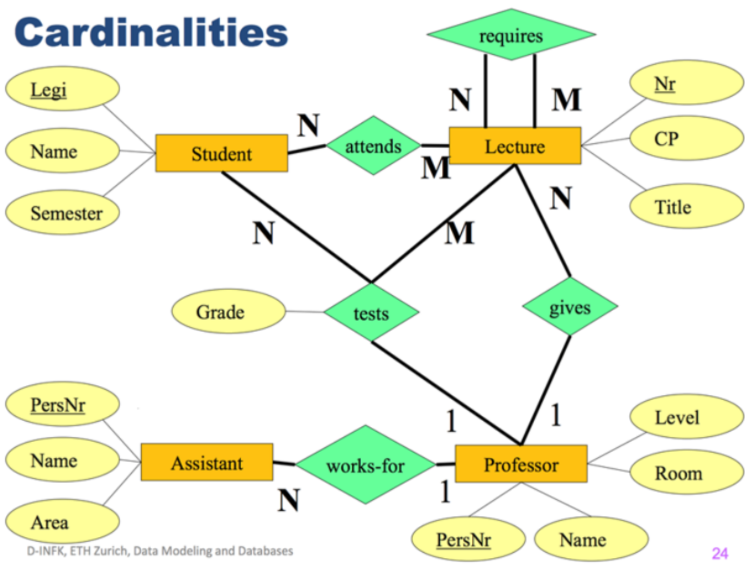
- Entity set E : 1-ary predicate $E(x)$, i.e. $E(x)$ is true if x is of entity type E
- Attribute A : binary predicate $A(x, y)$, i.e. $A(x, y)$ is true if x has attribute y
- n-ary relation R : n-ary predicate $R(x_1, ..., x_n)$, i.e. $R(x_1, ..., x_n)$ is true if $(x_1, ..., x_n)$ participate in R

Cardinality in ER-Diagrams

For relationships we distinguish between different types, that could all be represented in FOL:

- 1-to-many
- 1-to-1
- many-to-many
- many-to-1

Example:

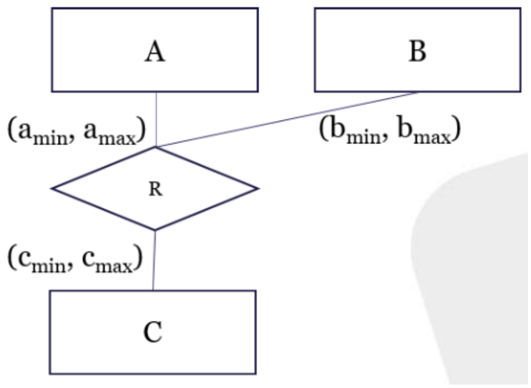


In this example the professor has a 1-to-many relationship with lectures. This means that a professor can have multiple lectures, but a lecture can only have one professor.

This can be expressed in FOL as:

$$\forall x_{Prof}, x_{Lect}.\ R(x_{Prof}, x_{Lect}) \Rightarrow \neg\exists x'_{Prof}.\ R(x'_{Prof}, x_{Lect}) \wedge (x_{Prof} \neq x'_{Prof})$$

We can also have a more expressive notation, called `(min, max)-notation`:



This specifies the following constraints:

- $\forall x_A . A(x_A) \Rightarrow \exists^{\geq a_{min}, \leq a_{max}} x'_B, x'_C . R(x_A, x'_B, x'_C)$
- etc.

Weak Entities in ER-Diagrams

Some entity's existence relies on other entities. e.g., both buildings CAB and HG have a room with number F 1. So how can we uniquely identify those two rooms?

We therefore say that *Room* is a `weak entity` relying on *Building*. The key of *Room* would be *(Bld#, Room#)*.

Design Principles of ER-Diagrams

When designing ER-diagrams, one should follow the following rules:

- Attribute vs. Entity
  - Entity if the concept has more than one relationship
  - Attribute if the concept has only 1:1 relationship
- Partitioning of ER-Models
  - Most realistic models are larger than a page
  - Partition by domains (library, research, finances, etc.)

- Good vs Bad models
  - Do not model redundancy or tricks to improve performance
  - Less entities is better
  - Remember the `C4 rule`: *concise, correct, complete, comprehensive*

4.3 Logical Modeling

ER-Diagram to Relational Model

- Entities become relations
- Relationships become relations
- Entity sets become tables
- Attributes of entity sets become attributes of the table
- Merge relations with the same key

Note that when there is no cardinality constraints, a `relationship becomes a table`, containing the keys of all participating entity sets.

5. Functional Dependency

5.1 Definition of Functional Dependency

Given a relation schema $\mathcal{R}(A : D_A, B : D_B, C : D_C, D : D_D)$ and instances $R \subseteq D_A \times D_B \times D_C \times D_D$. Now let $\alpha \subseteq \mathcal{R}$ and $\beta \subseteq \mathcal{R}$ (α is a subset of attributes, e.g. $\alpha = \{A, C\}$).

We now have a functional dependency $\alpha \to \beta$ iff, for any two tuples r, s in R, if they share the same values on attributes α then they share the same values on attributes β.

$$\forall r, s \in R.\ r.\alpha = s.\alpha \Rightarrow r.\beta = s.\beta$$

We write $R \vDash \alpha \to \beta$ if R satisfies $\alpha \to \beta$.

5.2 Defining Keys via Functional Dependency

α is a **superkey** iff $\alpha \to \mathcal{R}$.

A key α is **minimal** iff $\forall A \in \alpha.\ (\alpha - \{A\}) \not\to \beta$, we denote minimal functional dependencies as $\alpha \dot{\to} \beta$. If $\alpha \dot{\to} \mathcal{R}$ then α is a candidate key. Note that not all superkeys are minimal.

5.3 Infering Functional Dependencies

When given a set of FDs F, we want to find new FDs that are implied by F.

Armstong Axioms

- Reflexivity: $\alpha \subseteq \beta \Rightarrow \beta \to \alpha$
- Augmentation: $\alpha \to \beta \Rightarrow \alpha\gamma \to \beta\gamma$ where $\alpha\gamma := \alpha \cup \gamma$
- Transitivity: $\alpha \to \beta \wedge \beta \to \gamma \Rightarrow \alpha \to \gamma$

These three axioms are both complete and sound. All possible FDs can be implied from these axioms. We call a relation **trivial** if it holds for every instance of relation, e.g. $\alpha \to \alpha$.

Other Rules

- Union: $\alpha \to \beta \wedge \alpha \to \gamma \Rightarrow \alpha \to \beta\gamma$
- Decomposition: $\alpha \to \beta\gamma \Rightarrow \alpha \to \beta \wedge \alpha \to \gamma$
- Pseudo-Transitivity: $\alpha \to \beta \wedge \beta\gamma \to \theta \Rightarrow \alpha\gamma \to \theta$

5.4 Closures

Let F be a set of functional dependencies over $\mathcal{R}, \alpha \subseteq \mathcal{R}$ is a set of attributes of $\mathcal{R}$. The **closure of α with respect to F, a^+**, is the set of all attributes $\gamma \in \mathcal{R}$ such that $\alpha \to \gamma$ can be derived from F using Armstrong's axioms.

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

The following algorithm can be used to find the closure of α with respect to F:

```
a+ := a
do
    a+_old = a+
    for b -> c in F:
        if b subset a+
            a+ := a+ union c
while (a+_old != a+)
return a+
```

From this algorithm we can answer many questions like: How to check if $F \vdash \alpha \to \gamma$.

5.5 Minimal Basis / Minimal Cover

Given a set of FDs F, there might be redundant FDs that can be derived from others. We want to ask how to simplify F to remove such redundant FDs.

A set of FDs G is a **minimal cover** of F 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

We can achieve this with the following algorithm:

- Let G be the set of FDs obtained from F by decomposing the right hand side of FD to a single attribute.
- Remove FDs that are trivial.
- Remove all redundant attributes from the left hand side of FD in G.
- From the resulting set of FDs, remove all redundant FDs.

5.6 Normal Form

When we try to put to much information into a single relation, we encounter problems called `anomalies`. There are different types:

- **Redundancy:** Information may be repeated unnecessarily.
- **Update Anomalies:** We may change information in one tuple but leave the same information unchanged in another.
- **Deletion Anomalies:** If a set of values becomes empty, we may lose other information as a side effect.

Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms with the goal to reduce anomalies and improve data integrity.

For each normal form, we consider the following problems:

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

1NF - First Normal Form

The first normal form only has atomic domains. This makes it easier to define the concept of a key, there are no further restrictions. Example:

Father	Mother	Child
A	B	C1

Father	Mother	Child
A	B	C2

2NF - Second Normal Form

2NF tries to remove data redundancy caused by similar cases. $\{R, FD\}$ is in 2NF iff every non-key attribute is minimally dependent on every key. 2NF can be enforced by taking the bad FDs and decomposing the relation into multiple relations.

But there are still problems with the 2NF, including update and delete anomalies.

3NF - Third Normal Form

R is in 3NF iff for all $\alpha \rightarrow B$, at least one condition holds:

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

The intuition behind this is, that if it does not satisfy any of these conditions, then α is a concept in its own right. 3NF tries to get rid of transitive dependencies.

But 3NF still suffers from the same problems as 2NF.

BCNF - Boyce-Codd Normal Form

R is in BCNF iff for all $\alpha \rightarrow B$, at least one condition holds:

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

This is mostly the same as 3NF, with one condition removed. Intuitively, in each relation, you only store the same information once.

Now the questions arise: How to turn a DB schema into 3NF / BCNF and why do we need 3NF if we can have BCNF?

Decomposition Algorithm

```
Result = {R}
while (exists R_i in Result such that R_i is not in BCNF)
    Let a -> b be the evil FD
    R_i1 = a union b
    R_i2 = R_1 - b
    Result = (Result - R_i) union {R_i1, R_i2}
return Result
```

The result is guaranteed to be in BCNF and the output is a lossless decomposition of the original schema.

BCNF does not preserve all FDs, but 3NF does. It also does not get rid of all data redundancies, only the ones caused by functional dependencies.

Synthesis Algorithm

- Compute the minimal basis Fc of F
- For all $\alpha \rightarrow \beta \in Fc$, create $R_{\alpha \cup \beta}(\alpha \cup \beta)$
- If none of the above relations contains a superkey, add a relation with a key
- Eliminate R_α if there exists $R_{\alpha'}$ such that $\alpha \subseteq \alpha'$

This time the result is guaranteed to be in 3NF and the output is a lossless decomposition of the original schema.

Again the result is not free of any redundancies, since it is not in BCNF.

Multi-Value Dependency

Given a relation with attributes a, b, c. Intuitively, the value of b does not have an impact on the value of c; and b, c can take multiple values for the same a. We formally define:

$$a \to\to b \text{ for } R(a, b, c) \text{ iff:}$$

$$\forall t_1, t_2 \in R.\ t_1.a = t_2.a \Rightarrow \exists t_3, t_4 \in R:$$
$$t_3.a = t_4.a = t_1.a = t_2.a;$$
$$t_3.b = t_1.b;\ t_4.b = t_2.b;$$
$$t_3.c = t_2.c;\ t_4.c = t_1.c;$$

One way to think about MVD is to think about it in terms of joins. If we have $a \to\to b$ then we can decompose R, losslessly into $R_1 = \Pi_{a,b}R$ and $R_2 = \Pi_{a,c}R$, where $R = R_1 \bowtie R_2$.

Laws of MVDs

- Trivial MVDs: $\mathcal{R}(\alpha, \theta):\ \alpha \to\to \alpha\theta.(\alpha \to\to \mathcal{R})$
- Trivial MVDs: $\beta \subseteq \alpha \Rightarrow \alpha \to\to \beta$
- Promotion: $\alpha \to \beta \Rightarrow \alpha \to\to \beta$
- Complement: $\alpha \to\to \beta \Rightarrow \alpha \to\to \mathcal{R} - \alpha - \beta$
- Multi-Value Augmentation: $\alpha \to\to \beta \wedge (\delta \subseteq \gamma) \Rightarrow \alpha\gamma \to\to \beta\delta$
- Multi-Value Transitivity: $(\alpha \to\to \beta) \wedge (\beta \to\to \gamma) \Rightarrow \alpha \to\to \gamma$

4NF - Fourth Normal Form

R is in 4NF iff for all $\alpha \to\to \beta$, at least one condition holds:

- $\alpha \to\to \beta$ (trivial FD)
- α is a superkey of R

We can conclude that 4NF $\Rightarrow$ BCNF.

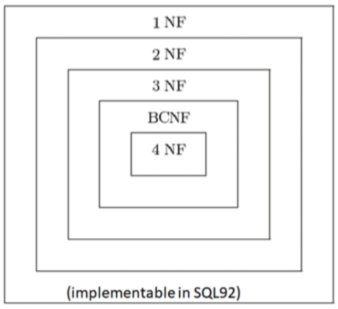
Decomposition Algorithm

```
Result = {R}
while (exists R_i in Result such that R_i is not in 4NF)
    Let a -> -> b be the evil MVD
    R_i1 = a union b
    R_i2 = R_i - b
    Result = (Result - R_i) union {R_i1, R_i2}
return Result
```

Normalization - Higher the better?

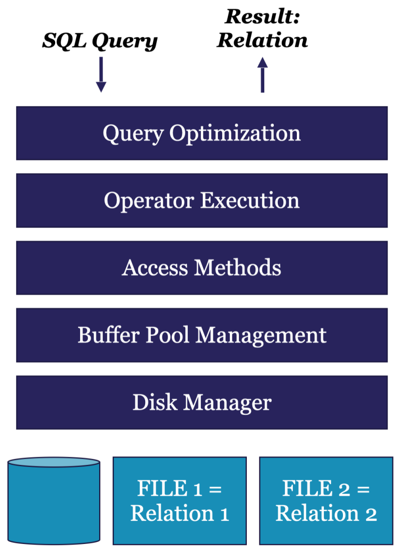
We ask ourself the question if higher normalization is better than lower normalization. In practice this is not always the case, as denormalized databases can be faster to read, while normalized databases are less redundant and therefore it is easier to maintain consistency.

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



Database Systems

Database management systems can be quite complex. In this course, we focus on relational DB with disk-oriented architecture, meaning all data is stored on disk. We can represent a DBMS as different layers.



1. Disk Manager

The disk manager is responsible for interacting with the disks, meaning allocation, deleting and fetching pages.

For this course we will focus on a system with a simple storage hierarchy: HDD -> DRAM -> CPU.

The disk manager organizes files as a collection of `pages`, where a page is a fixed-size block of data with a unique identifier (page id). One page again consists of a collection of `tuples`.

1.1 Heap File

A heap file is an unordered collection of pages where tuples are stored in random order. The `Record ID` consists of the Page ID and the Slot ID. To support record level operations, we must keep track of the pages in a file, keep track of free space on pages and keep track of the records on a page. There are two ways to implement a Heap File.

Linked List

A header page stores two pointers: one pointer to the free page list and the other pointer to the data page list. From there we simply have two linked lists.

Page Directory

The page directory consists of a set of header pages, each of which contain pointer to data pages. These header pages also track the number of free slots on each page. Overall the page directory should be faster than the linked list approach.

1.2 Page Layout

Each pages consists of a header and a data part. The header contains meta-data about the page, such as the page size, DBMS version, compression info and checksum. For the data layout there are multiple approaches.

Naive Strategy

We have a fixed length for data slots in the page and the header keeps track of the number of tuples.

Slotted Page

Each record ID consists of a page ID and a slot number. At the top of the page we have a slot array that contains pointers to the start position of each record. One advantage is, that we can move records on a page without changing their record ID.

Tuple Layout vs Column Layout

We can either store all attributes of a tuple together in a single slot. This makes it easy to access all attributes of a tuple, but also makes it harder to access one attribute for all tuples. The column layout stores values of the same column together. This reduces the amount of wasted I/O and makes data easier to compress, but point queries, inserts, updates and deletes get slower.

2. Buffer Pool Management

The buffer pool is responsible for managing the pages in memory. In the end all upper layers have the illusion that the data is in memory.

The key question here is when to evict pages from memory. If we know all future accesses, we can simply discard the block whose next access is farthest in the future.

In practice this is often not the case. There we use either LRU (least recently used) or MRU (most recently used) replacement strategies. Both have their own advantages and it depends on the data access pattern which to prefere. The DBMS should often know these patterns and be able to tell the buffer manager.

3. Access Methods

The access methods provide different ways of accessing data from a relation (Sequential Scan, B-Tree index, Hash table, Sort).

An access method is a sequence of invocations to the buffer manager to access information in a relation. We will look at three different access patterns.

3.1 Sequential Scan

The goal of a sequential scan is to find a tuple whose attribute X equals to Y. An example would be:

```
SELECT * FROM table WHERE X = y;
```

It works like this:

- get page 1, scan all tuples and check
- get page 2, scan all tuples and check
- ...

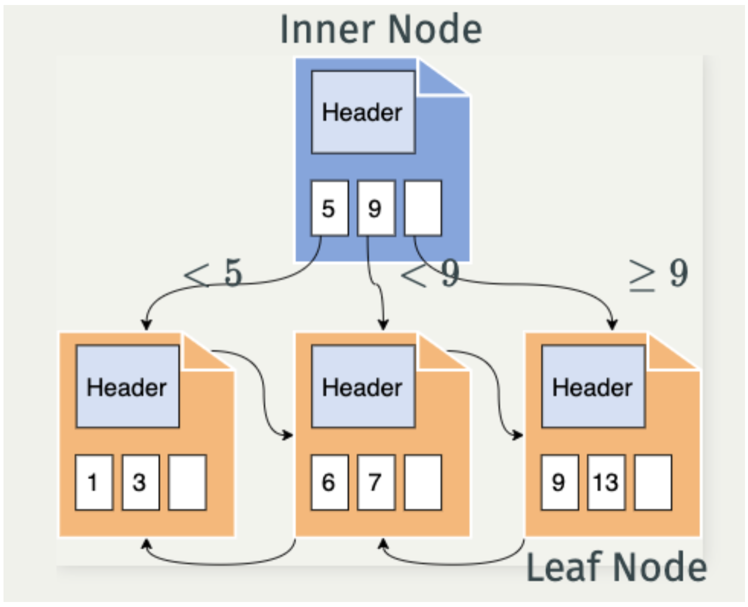
This is not very effective and we ask ourself if we can do better?

3.2 B-Tree Index

A `B+ Tree` is a self-balancing tree data structure that keeps data sorted. It is a generalization of a binary search tree in which each node contains more than two children. It has the following properties:

- The tree 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.

We differentiate between unclustered B+ Tree and clustered B+ Tree. In a unclustered B+ Tree, the leaf nodes contain the record ID, pointing to the relation, while in a clustered B+ Tree, the leaf nodes contain the actual tuple.



This tree structure allows us to performe search, insertion and deletion in $\mathcal{O}(\log_M N)$, where M is the maximum number of children per node.

3.3 Hash Table

Hash Tables are even better for point queries, as they run in $\mathcal{O}(1)$. We can use a hash function to create a mapping from a key to a bucket. When we encounter a collision, we can either try to find the next empty slot or we can use a linked list to store multiple values for the same bucket.

4. Operator Execution

The operator execution layer is responsible for executing the relational algebra operators (Join, Projection, Select, . . .).

4.1 Select

Given a predicate C we want to find the tuples that satisfy C.

```
SELECT * FROM table WHERE C;
```

We can either use sequential scan or B+ Tree index. If the predicate is of the form $=, <, >, <=, >=$ we can use B+ Tree index. For other predicates sequential scan is the way to go.

4.2 Sort

When sorting it is best to use a clustered B+ Tree, since we simply have to scan the leaf nodes. If the B+ Tree is unclustered, we have one random access per tuple, which is going to be a lot slower.

Sorting Tuples on Disk

We might want to sort the actual data on the disk. First we notice the quick sort is not designed to be I/O efficient and therefore might not be ideal to use. Therefore we use external sort. Assuming we have N pages on disk to sort and a B page buffer. If $N < B$ we can simply load all data to the buffer and perform quick sort. If $N > B$ we proceed as follows:

- Partition a large file into small chunks, each of which fits into the buffer.
- Sort each small chunk.
- Combine all these sorted chunks into a single large chunk.

This is really similar to merge sort.

4.3 Join

Again there are multiple ways to performe a join operation.

Nested Loop Join

```
for tuple1 in R:
    for tuple2 in S:
        if tuple1.x == tuple2.x:
            result.append(tuple1, tuple2)
```

Such a loop is very inefficient. We want to have to smaller relation in the outer loop and the larger relation in the inner loop, for it to be at least somewhat efficient.

Block Nested Loop Join

Blocking already gains us a large performance improvement of two orders of magnitude.

```
for block1 in R:
    for block2 in S:
        for tuple1 in block1:
            for tuple2 in block2:
                if tuple1.x == tuple2.x:
                    result.append(tuple1, tuple2)
```

Index Nested Loop Join

Performing an index scan, as seen before, is another approach to speed up the process.

```
for tuple1 in R:
    for tuple2 in IndexScan(S, r, x):
        result.append(tuple1, tuple2)
```

Using a hash index, this approach can join arbitrary data types.

Sort Merge Join (equi-join)

We assume that both relations are already sorted. Now we can scan both relations, comparing the head of each relation. If the head of the first relation is smaller than the head of the second relation, we can simply skip the head of the first relation. If the head of the first relation is larger than the head of the second relation, we can simply skip the head of the second relation.

Hash Join (equi-join)

First we build a hash table for the first relation. Then we scan the second relation and calculate its hash value. If the hash value is in the hash table, we can join the two tuples, after actually checking if the tuples are equal (to avoid false positives).

One advance approach to this is `Grace Hash Join`. Together with Sort Merge Join, Grace Hash Join is the best way to perform an equi-join. When choosing between these two approaches, we should take into account if the data is already sorted.

5. Query Optimization

The query optimization layer is responsible for generating a good execution plan for a given SQL query.

We divide the query optimization into four parts:

- Execution Model: How different operators are put together
- Search Space: What are the logically equivalent set of physical plans
- Cost Model: How to estimate the cost of each physical plan
- Search Algorithm: How can we search the best physical plan

Execution Model

There are different ways to put operators together.

Iterator Model

Each operator is an iterator - it takes as input a set of streams of tuples an provides a `next()` interface. A query plan is a tree of iterators and to get the result we call `next()` on the root iterator again and again.

This model is great at hiding information and abstraction, it supports pipelining and parallelism. But on the other hand it also creates a large overhead of method calls and poor instruction cache locality.

Vectorization Model

Similar to the iterator model, but each operator returns a batch of tuples instead of a single tuple.

Materialization Model

Each operator processes its input all at once and returns all the results at one. This is good when the indermediate result is not too much larger than the final result.

Search Space

Given an input logical plan, there are different ways to construct the physical plan. We now define a set of transformation rules, which are used to generate equivalent query plans.

- $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- $\sigma_{\theta_2}(\sigma_{\theta_1}(E)) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- $\Pi_{t_1}(\Pi_{t_2}(E)) = \Pi_{t_1}(E)$
- $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
- $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_2 \wedge \theta_1} E_2$
- $E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$
- and many more . . .

Cost Model

Given a physical plan we want to estimate the performance without actually running the query.

Cardinality estimation is an approach that tries to estimate the number of resulting tuples by creating a histogram (if continuous values use bins) of the input tuples and then estimating the number of resulting tuples under the assumption that the inputs are independent.

If we really care about correlation we might use multi-dimensional histograms.

Search Algorithm

It is not possible to generate and look at all equivalent plans. We need some restrictions to be able to perform a good search.

The first approach might be to constrain the search space. For example we may only consider left-deep join trees, as the allow us to generate all fully pipelined plans.

Another approach is to use heursistic based optimization. For example we try to perform selection and projection as early as possible (pushdown).

6. Transactions & ACID

Assuming a very simple DB with only a single thread running. If the user submits two SQL queries at the same time, the DB can run parts of each instruction interleaved. This can lead to bad behavior. We want these instructuins to have the same effect as if they were executed sequentially. Still our goal is to enable concurrency whenever safe to do so.

We also want to avoid bad changes when our system fails in the middle of executing some statements.

A `transaction` is a sequence of one or more SQL operations treated as a unit, where concurrent transaction appear to run in isolation and if the system fails, each transaction's changes are reflected either entirely or not at all.

In SQL we start a transaction with `BEGIN` and end it with either `COMMIT` or `ABORT`.

6.1 ACID

We want to have the following properties for our system:

- **Atomicity:** A transaction is executed in its entirety or not at all.
- **Consistency:** A transaction executes in its entirety over consistent DB produces a consistent DB.
- **Isolation:** A transaction executes as if it were alone in the system.
- **Durability:** Committed changes of a transaction are never lost, they can be recovered.

6.2 Formal Definition

Transactions are a sequence of read and write operations:

- $a < -R(A)$: read object A into variable a
- $W(B, b)$: write variable b into object B

A transaction has to start with `BEGIN` and end with `COMMIT` (all changes are save) or `ABORT` (all changes are undone).

A schedule is a way of mixing instructions between different transactions.

6.3 Serializability

This notion of isolation is strong and we might not need this, therefore DBs provide different levels of isolation.

Read Uncommitted

This allows for a transaction to perfom dirty reads. Meaning that we can read a uncommitted value from another transaction.

Read Committed

A transaction may not perform dirty reads. This is stronger than read uncommitted, but still not as strong as other levels of isolation.

Repeatable Read

A transaction may not perform dirty reads and an item read multiple times cannot change its value.

Serializable

A valid execution of transactions needs to be equivalent to one possible serial execution.

Note that not each schedule is serializable.

Conflict Serializability

The I/O pattern alone does not decide serializability. `Conflict Serializability` is a stronger notion of serializability that only depends on the I/O pattern.

There are different types of conflicts:

- **Read-Write Conflict:** leads to unrepeatable reads
- **Write-Read Conflict:** leads to dirty reads
- **Write-Write Conflict** leads to overwriting uncommitted data

Conflict equivalent means that two schedules involve the same actions of the same transactions and every pair of conflicting actions is ordered in the same way.

Conflict serializability is defined as a schedule being conflict equivalent to some serial schedule, i.e. we can translate the schedule into a serial schedule with a sequence of nonconflicting swaps of adjacent actions.

How to decide Conflict Serializability?

We can either use the naive solution and do the swaps, or we can create a dependency graph:

- Each transaction is a node
- There is an edge from T_i to T_j if the operator o_i is in conflict with o_j and o_i is executed before o_j

A schedule is conflict serializable if there is no cycle in the dependency graph.

6.4 Enforcing Isolation

There are two ways of dealing with non-serializable schedules. Either we say they happen all the time and proactively try to prevent them using locks or we say that they are rare and we deal with them whenever they happen (snapshot isolation).

Locking

We want to lock the data object before accessing it and unlock it only when it is safe to do so.

We differentiate between different types of locks.

- **Shared Lock:** locks the data object for reading
- **Exclusive Lock:** locks the data object for writing

There are also different types of locking.

2PL: Two Phase Locking

- **First Phase:** Transaction requests the lock from the DBMS's lock manager.
- **Second Phase:** Transaction is allowed to release locks that it acquired, but it can't acquire new locks.

This guarantees conflict serializability, but there is a problem. We can encounter cascading aborts, that is if a transaction aborts, we might have to roll back other transactions that already committed.

Strict 2PL

The first phase is the same as 2PL, but the second phase is different, as all locks have to be kept until the end of the transaction. This fixes the previous problem, but it introduces deadlocks.

To avoid deadlocks breaking the system, we might introduce a so called wait-for graph (edge from t_i to t_j if t_i waits for a lock from t_j) and periodically check if there is a deadlock (cycle).

Granularity of Locks

DBs have a hierarchical structure, we need to support locks at multiple granularity, both for correctness and performance.

For locks to work over multiple levels of the hierarchy, we introduce new types of locks:

- **IS - Intention Share:** some lower nodes are in shared
- **IX - Intention Exclusive:** some lower nodes are in exclusive
- **SIX - Shared and Intention Exclusive:** the root is locked in share and some lower nodes are in exclusive

Now to aquire a lock we go through the hierarchy and lock each node. For example if we want to lock a tuple in S, we start from the root:

- Aquire IS on Database
- Aquire IS on Table
- Aquire S on Tuple

Implementing Isolation

We can now apply these locks to enforce the isolations levels we talked about:

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

6.5 Isolation Beyond Locking

Now we want to look at the case where we want to enforce serializability but we don't want to use locks. We do this in the case where non-serializable schedules are rare.

The idea is to assign each transaction a timestamp and then the DBMS must ensure that transactions with an older timestamp have to appear in the serial schedule before transactions with a newer timestamp.

Further each database object is associated with a read and write timestamp, containing the highest timestamp of a transaction that has read or written the object.

Now if T_i tries to access an object with a higher timestamp than T_j, the transaction will be aborted and restarted.

This gives a conflict serializable schedule that is deadlock free, but it can introduce starvation (a large transaction could starve to death) and cascading abort becomes a problem again.

Snapshot Isolation

This protocol implements the idea from above. All writes are carried out in a separate buffer and when the transaction T_1 commits, the DBMS first checks for conflicts. If there exists another transaction T_2 such that T_2 commits after the timestamp of T_1 and before T_1 commits, and both update the same object, the transaction T_1 is aborted.

This prevents writers and readers from blocking each other. On the other hand it introduces overhead and unnecessary rollbacks.

6.5 Recovery

Schedules have different levels of recoverability properties:

- **Recoverable (RC):** if T_i reads from T_j and $c_j > c_i$
- **Avoids Cascading Aborts (ACA):** if T_i reads X from T_j and $c_j < r_i[X]$
- **Strict (ST):** if T_i reads from or overwrites a value written from T_j, then ($c_j < r_i[X]$ AND $c_j < w_i[X]$) or ($a_j < r_i[X]$ AND $a_j < w_i[X]$)

We hope to only allow schedules that are conflict serializable and strict. Strict 2PL actually guarantees these properties.

Write-Ahead Log

We assume that disk storage is safe. To actual implement recovery, we write a log file, that first lives in memory and gets flushed to disk. We can append records to the log file (START, COMMIT, ABORT, UPDATE) and flush the log.

Undo Logging

If T modifies X, write $< T, X, \text{ old value } >$ to the disk, before X gets changed. If a transaction commits, the COMMIT record must be written to disk only after all other changes are written to disk.

For recovery we simply search for uncommited transactions, if it is only one, we undo the changes and write ABORT at the end of log and flush it. If there are multiple uncommited transactions, we scan from the end, undo uncommited changes, skip those updates made by commited transactions and write ABORT at the end of log and flush it.

Redo Logging

If T modifies X, write $< T, X, \text{ new value } >$. Before modification on the disk, we flush the log and COMMIT to disk.

For recovery we go through the log and redo all changes of commited transactions, if a transaction is incomplete we write ABORT.

Undo/Redo Logging

Before modifying any database element X on disk, we write $< T, X, \text{ old value}, \text{ new value} >$ and flush the log before actual changes are made on disk.

Now if COMMIT is not in the log, it is incomplete and we undo it. If COMMIT is in the log, the transaction is complete and we redo it.

7. Distributed Transactions

7.1 Distributed Commit

Considering a system with multiple, distributed nodes. We want to commit a transaction only if all of the nodes have committed it.

Two Phase Commit

- Coordinator requires if all nodes are ready and willing to commit (coordinator writes PREPARE)
  - Workers can say NO, but if they don't, they can't in the next stage
  - If worker says OK, it writes PREPARE locally
- If all workers say OK, coordinator asks all nodes to commit (coordinator writes COMMIT)
  - If all workers say OK, commit (worker write COMMIT locally)
  - If one worker is not OK, abort (worker write ABORT locally)

The downside of the protocol is, that a lot of messages have to be exchanged.

Linear Two Phase Commit

- Coordinator asks Worker 1 if it is OK, Worker 1 asks Worker 2, etc.
  - if all Workers say OK, last Worker sends COMMIT back to Coordinator
  - if one Worker says NO, it forwards NO and sends NO back

This protocol minimizes the number of messages exchanged, but introduces greater latency.

7.2 Distributed Query Processing

Distributed databases can have shared memory, shared disk or share nothing. We focus on the share nothing architecture. To optimize query performance we need to know how our data is partitioned across different nodes.

Naive Table Partitioning

Each relation is on one single node. This allows to process two relations concurrently, but if one relation does not fit on one node we have a problem.

Horizontal Partitioning

Relations are split across multiple nodes. For small results this can be really fast as each node can process the query independently. Some operations might need all data from a relation, this would create a communication bottleneck.

Distributed QO

One relation might be replicated at every node, while the others are horizontaly split. This is good for queries that need to access all data from one relation, e.g. join operations.

Another approach here would be to partition the relations horizontally, based on the join attribute.

7.3 Distributed Key-Value Store

Key-Value store is a different approach to storing and processing data, there is no SQL, ACID, etc. It is designed to scale better and to ensure high availability across multiple nodes.

The data model is a list of key-value pairs (or key-pointer), that can be indexed by a key. The data is distributed horizontally across multiple nodes. For replication quorums (simple majority) or asynchronous replication is used.

These systems for example used for user profiles, retrieving web pages, etc. The downside is that it is not easy to support anything else than point queries and data is often inconsistent between nodes.

For a single node we can use a simple hash table. For a distributed system we need to use a more sophisticated data structure. We need consistend hashing. To achieve this we hash both node and data, so each node deals with the data points in the clockwise direction, before the next machine (range of hash values). If a node leaves or a new one joins the system, the data is moved between machines. For the case that a node crashes, we might replicate it's data to the r next nodes along the direction.