Abstract Data Modelling and Databases

Roman Böhringer, June 2019

Table of contents

Functionality

1 Introduction

Multiple challenges arise when one uses custom-built scripts for data management: The data needs to have a life-cycle that is longer than a single query and it might be too large for the memory of the computer. If one decides to use the file system, there are other challenges like performance, concurrent access, crash recovery and the code needs to change once you decide to change the file format. The advantages of a DBMS (database management system) are:

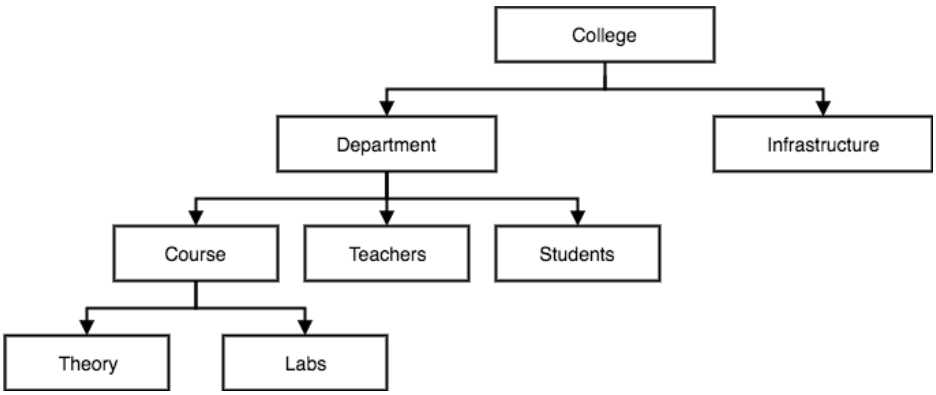
- **Data Independence:** The application should not know how the data is stored
- **Efficient Data Access:** The system should be able to stare and retrieve data efficiently, without users worrying about it (too much)
- **Transactional Access:** As if there is only a single user using a system that does not fail.
- **Generic Abstraction:** Users do not need to worry about all of the above issues for each new query.

But there are also possible reasons for not using a database like a workload mismatch (specialized application might not be what a certain DBMS is designed for) or a data model mismatch (application cannot be naturally modeled by a given DBMS, because it uses for instance graphs).

Historically, there have been (and still are) multiple different database models:

- **Hierarchical Model:**

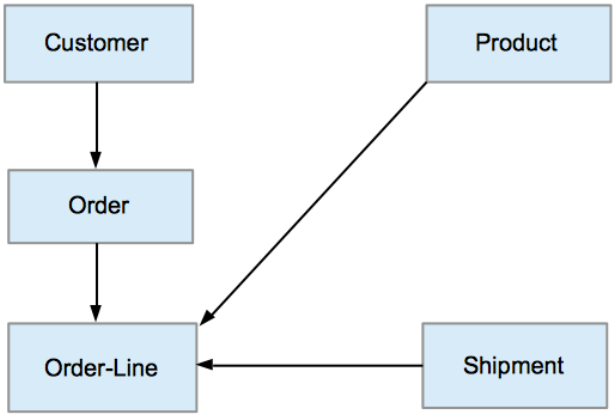
Data is organized into a tree-like structure. The data is stored as records which are connected to one another through links. It is mandated that each child record has only one parent, whereas each parent record can have one or more child records. So a possible tree looks like this:



The query language queries one record at a time and essentially travers the tree. Certain queries are very hard to express in this language, like getting all parents of a child with a certain attribute (e.g. the department of all teachers that start with "A"). So if the data is not really a tree, one ends up storing redundant information. Furthermore, data independence is not really achieved (the application / queries have to be changed when the physical data representation changes) and the query language is not declarative, which forces the user to do manual query optimization.

- **Network Model:**

The network model allows (in contrast to the hierarchical model) each record to have multiple parent and child records, forming a generalized graph structure, e.g.:



Data Independence is still not achieved, and the query language isn't declarative.

- **Relational Model:**

Relations are set of tuples of the same type. Instead of querying one record at a time, one set at a time is queried. Data independence is achieved because physical data representation doesn't change the query and a declarative language (SQL) enables automatic query optimization.

2 ER-Diagram

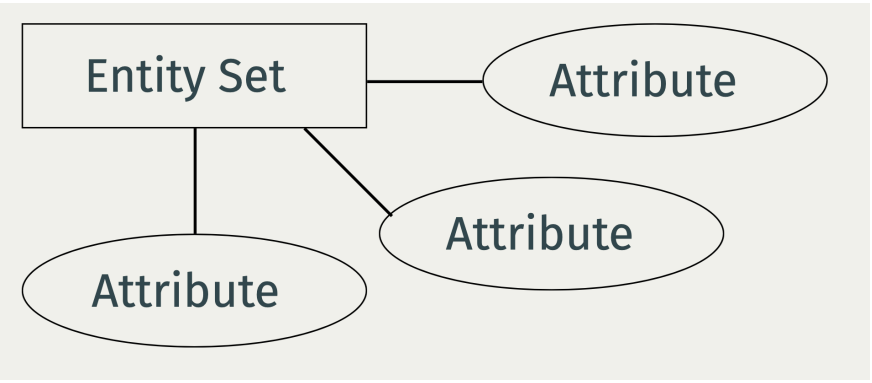
Modelling consists of conceptual modeling (capturing the domain to be represented), logical modeling (mapping the concepts to a concrete logical representation) and physical modeling (implementation in concrete hardware). An ER-diagram / model (entity relationship diagram / model) deals with conceptual modeling.

An ER-diagram is a graphical representation containing three element types:

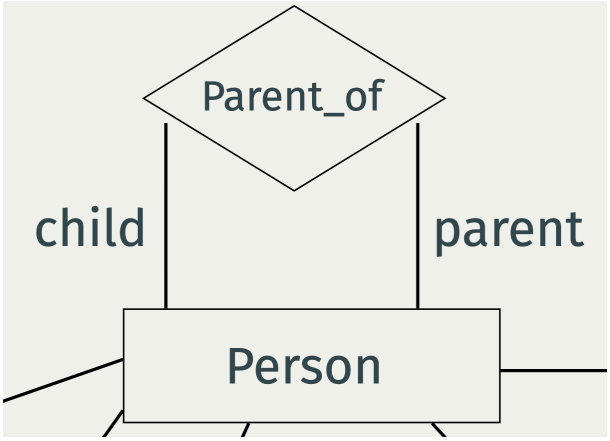
- Entity sets
- Attributes
- Relationships

An entity is an abstract object of some sort / an object in the real world that is distinguishable from other objects (depending on the definition). An Entity Set is a set of similar entities (similar in the sense that entities in the same entity set share the same attributes).

Relationships are connections among two or more entity sets (e.g. the relationship "enrolled in"). An ER-Diagram is a graphical way of representing entities and the relationship among them. Entity sets, relationships and attributes are visualized like this:



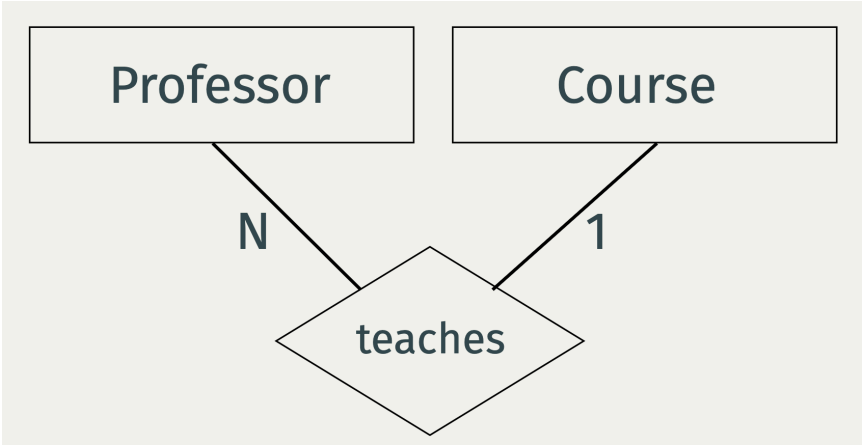
Roles can be modelled by relationships, e.g. "parent_of":



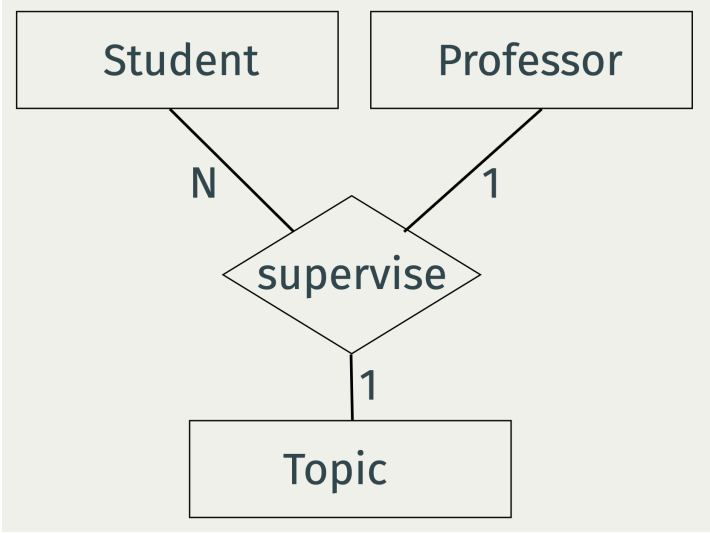
We have the following definitions for ER-Diagrams:

- **Key:** A *minimal* set of attributes whose values uniquely identify an entity in the set.
- **Candidate Key:** Any key is a candidate key, there might be multiple of them.
- **Primary Key:** Only one candidate key can be the primary key of the entity set.

Relationships can have different cardinality (1-to-1, 1-to-many, many-to-1, many-to-many) which is denoted by a 1 or N (many) on the corresponding edge. In the following example, one course can be taught by multiple professors, but one professor can only teach one course:

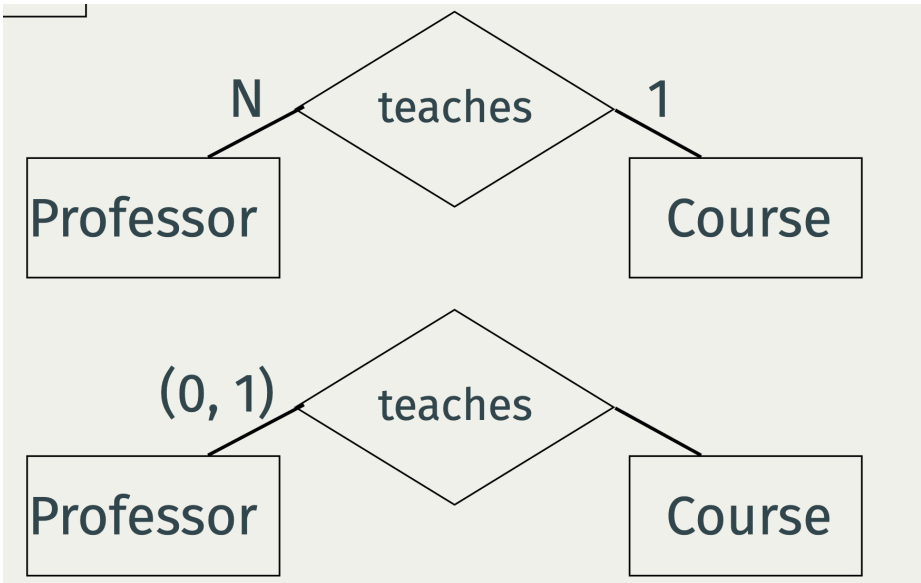


If we have a relationship to an entity set with a "1", the other entities of the relationship uniquely decide this entity set. For instance, in the following ER-Diagram:

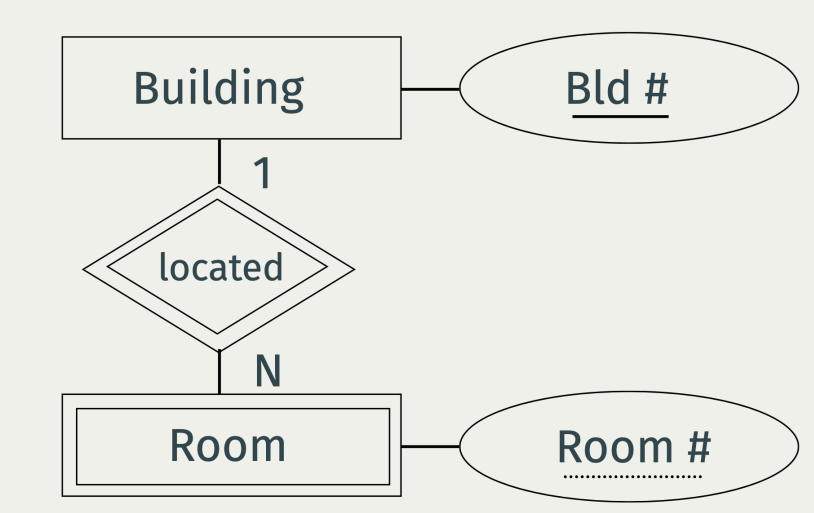


Given a student and a professor, we know the topic. Given a student and a topic, we know the professor. But given a topic and a professor, there can be multiple students.

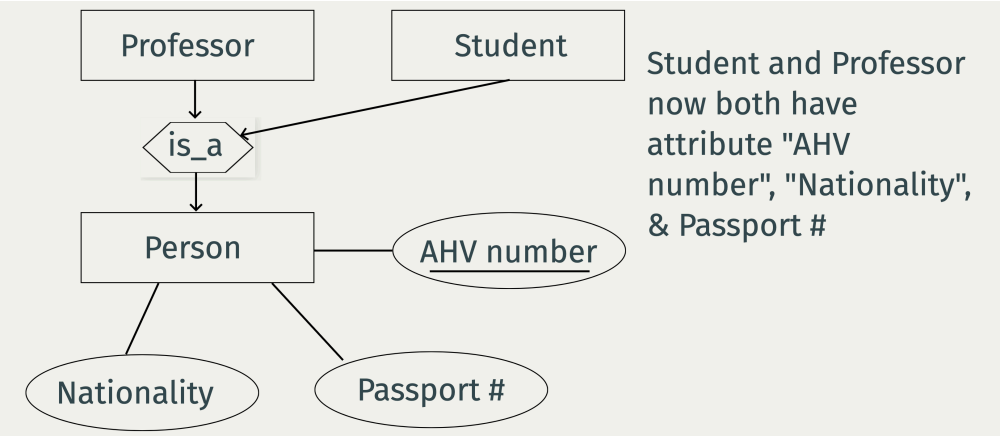
There's also the more generic (min, max) notation which denotes how often an entity can participate in a relation. So the following notations are equivalent:



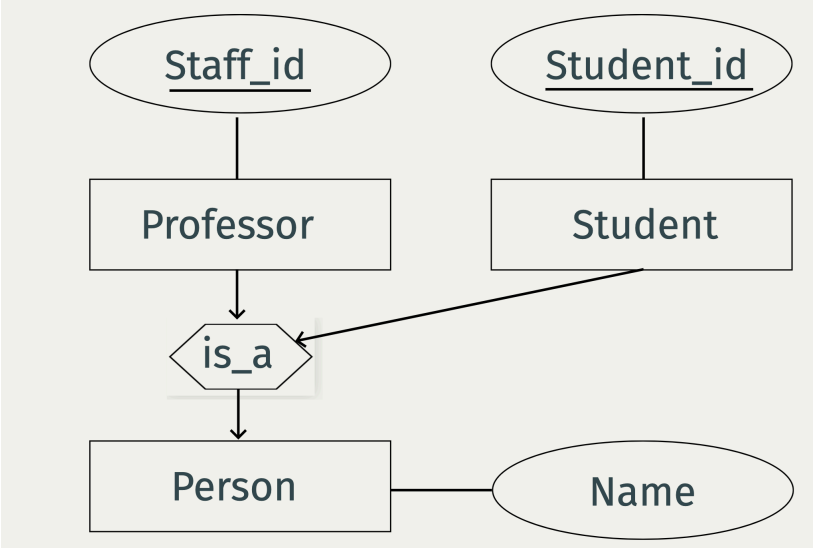
Some entity's existence relies on other entities. These are called weak entities. The key is the key of the entity + the key of the relied on other entity. In ER-Diagrams, weak entitles are modeled like this (in this example, the weak entity "Room" with the key (Bld #, Room #):



Generalization is used when some entity sets share some common attributes but are different entities (in the object under consideration), e.g. Professor / Student and Person:



But they can also have their own primary key (but in this model, no persons that aren't professor / students are allowed):



We assume that there is no NULL in an ER model.

3 Relational model

Relational models are logical models (i.e. map the concepts to concrete logical representation; in this case relations).

A database is a collection of one or more relations. Each relation has a name and a set of fields (also called attributes / columns). Each field has a name and a domain (e.g. int or string).

An instance of a relation is a set of tuples, also called records, in which each tuple has the same number of fields as the relation schema. Therefore, we have formally:

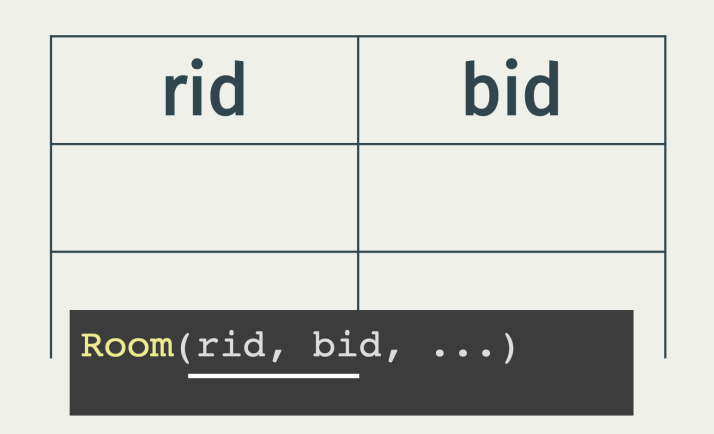
- Schema: $R(f_1 : D_1, \ldots, f_n : D_n)$
- Instance: $I_R \subseteq D_1 \times \ldots \times D_n$

In a relational model, the primary key is marked by underlining. Foreign keys are attributes that identify tuples in another relation.

When converting an ER-Model to a relational model, entity sets become tables and the attributes of the entity set become attributes of the table. When there are no cardinality constraints, relationships become a table, containing the keys of all participating relations. When there are

cardinality constraints, only the keys that are needed to identify the relation (i.e. the entity sets with the "N") become keys. Relationships with the same primary key are (often) merged.

Weak entities become a separate table with the key of the relied on other entity and the own key (part of the key) as primary key, i.e. we have in the previous example:



For generalization, there are two possibilities: We can have separate tables that contain only the attributes of the "specialized" entity (& the primary key of the "base" entity), e.g. Professor(PersNr, Level). Or the "specialized" entity can also contain all the attributes of the "base entity", e.g. Professor(PersNr, Level, Name).

We assume that there is no NULL in relational model.

4 Relational Algebra

Relational algebra is a family of algebras used for modelling the data stored in relational databases and defining queries on it.

We assume set semantics when dealing with relational algebra (one could also assume bag semantics where duplicates could occur). The basic operators of relational algebra are:

- **Union**: $\cup$

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

- **Difference**: $-$

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

  - *Derived: Intersection* $\cap$

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

- **Selection**: σ

$$x \in \sigma_c(R) \Leftrightarrow x \in R \wedge c(x) = \text{ true } \quad \text{e.g. } \sigma_{\text{salary}>40000}(\text{ Employee })$$

- **Projection**: Π

$$\Pi_{A_1,\ldots,A_n}(\mathrm{R}) \text{ eliminates columns e.g. } \Pi_{ssn,name}(\text{ Employee })$$

- **Cartesian Product:** $\times$ (normal product, i.e. each tuple with each tuple)
- **Renaming:** ρ (changes the schema, not the instance)

$$\rho_{B_1,\ldots,B_n}(R) \text{, e.g. } \rho_{A,B,C}(\text{ Employees })$$

- **Join**: $\bowtie$

$$R_1 \bowtie R_2 = \Pi_A (\sigma (R_1 \times R_2)) \text{ i.e.:}$$

  - Selection checks equality of all common attributes
  - Projection eliminates the duplicate common attributes

For instance, for R(A1...An, B1...Bn) and S(B1...Bn, C1...Ck):

$$R \bowtie S = \Pi_{A_1 \ldots A_n, B_1 \ldots B_n, C_1 \ldots C_k} (\sigma_{R.B_1 = S.B_1 \wedge \ldots \wedge R.B_n = S.B_n} (R \times S))$$

When R and S do not have common attributes, a natural join is equivalent to the cartesian product. When they have the same set of attributes, it is equivalent to the intersection

- **Theta Join:** $\bowtie_\theta$

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

  - *Eq-Join:* $\bowtie_{A=B}$

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

  - *Semi-Join:* $\ltimes_C$

$$R_1 \ltimes_C R_2 = \sigma_{A_1 \ldots A_n} (R_1 \bowtie_C R_2)$$

For instance, a query like this: $\text{Employee} \bowtie_{SSN=EmpSSN} (\sigma_{age>71}(\text{ Dependent }))$
could be semi-join reduced to (in a distributed database):
$(\text{Employee} \ltimes_{SSN=EmpSSN} (\Pi_{EmpSSN} \sigma_{age>71}(\text{ Dependent })) \bowtie_{SSN=EmpSSN} Dependent$

- **Outer Joins:**

- left outer join (natural join + unmatched tuples from L)

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟕

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Result				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-

- right outer join (natural join + unmatched tuples from R)

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟖

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
-	-	c_3	d_2	e_2

- (full) outer join

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟗

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-
-	-	c_3	d_2	e_2

- **Division:**

$$t \in R \div S \Leftrightarrow \forall s \in S, \exists r \in R, r.S = s.S, r.(R - S) = t$$

For instance, "find students who attended all lectures with 4 CP".

R	
M	V
m_1	v_1
m_1	v_2
m_1	v_3
m_2	v_2
m_2	v_3

÷

S
V
v_1
v_2

=

R ÷ S
M
m_1

As we could see in the Theta Join example, relational algebra isn't declarative (different queries lead to the same result but with different performance). Furthermore, there are certain limitations (transitive closure, e.g. finding all ancestors of a person, can't be done in relational algebra).

We assume that there is no NULL in relational algebra.

5 SQL

SQL (structured query language) is a family of standards consisting of:

- Data definition language (DDL) to define schemas
- Data manipulation language (DML) to update data
- Query language to query data

Different database engines might have an implementation that is slightly different from the SQL standard (or unique extensions).

5.1 Data definition

To create a relation, one needs to specify a name, set of columns and the type of each column:

```
CREATE TABLE Professor
    (PersNr integer not null,
     Name   varchar (30) not null
     Level  character (2) default "AP");
```

The most important data types (in PostgreSQL) are:

- CHAR(n) / CHARACTER(n): fixed-length character string
- CHARACTER VARYING(n) / VARCHAR(n): Variable-length character string

"SQL defines two primary character types: character varying(n) and character(n), where n is a positive integer. Both of these types can store strings up to n characters (not bytes) in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type character will be space-padded; values of type character varying will simply store the shorter string."[1]

- NUMERIC(p,s) / INTEGER: Numeric types, p is precision and s the scale. An INTEGER has a storage size of 4 bytes.
- BLOB / RAW: For large binaries
- DATE: For dates

A relation is deleted like this:

```
DROP TABLE Professor;
```

To modify the structure of a table, one uses:

```
ALTER TABLE Professor ADD COLUMN (age integer);
```

The syntax of the command (most important parts) looks like this[2]:

```
ALTER TABLE [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column TO new_column
ALTER TABLE name
    RENAME TO new_name
ALTER TABLE name
    SET SCHEMA new_schema

where action is one of:

    ADD [ COLUMN ] column data_type [ COLLATE collation ] [ column_constraint
[ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column [ SET DATA ] TYPE data_type [ COLLATE collation ]
[ USING expression ]
    ALTER [ COLUMN ] column SET DEFAULT expression
    ALTER [ COLUMN ] column DROP DEFAULT
    ALTER [ COLUMN ] column { SET | DROP } NOT NULL
ADD table_constraint [ NOT VALID ]
    ADD table_constraint_using_index
```

[1] From https://www.postgresql.org/docs/9.2/datatype-character.html
[2] From https://www.postgresql.org/docs/9.1/sql-altertable.html

```
DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
```

New columns are initially filled with whatever default value is given (NULL if no default value is specified). When a new column has a NOT NULL constraint and no DEFAULT value and the table already contains values, an error is thrown.

Indexes are managed like this:

```
CREATE INDEX myIndex ON Professor(name, age);
DROP INDEX myIndex;
```

5.2 Data manipulation

5.2.1 Insert, Update, Delete

To insert tuples, INSERT is used, e.g.:

```
INSERT INTO Student (Legi,  Name)
            VALUES (28121, `Frey`)
```

The syntax is:

```
INSERT INTO table_name [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] |
query }
```

«The target column names can be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order; or the first N column names, if there are only N columns supplied by the VALUES clause or query. The values supplied by the VALUES clause or query are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.»

Multiple tuples are separated by commas, e.g.:

```
INSERT INTO Student (Legi,  Name)
            VALUES (28121, `Frey`), (28122, `Meier`)
```

Tuples can be deleted with DELETE, e.g.:

```
DELETE FROM Student
WHERE  Semester > 13;
```

The syntax is:

```
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
```

Tuples can be updated with UPDATE, e.g.:

```
UPDATE Student
SET    Semester = Semester + 1;
```

The syntax is

```
UPDATE [ ONLY ] table [ * ] [ [ AS ] alias ]
    SET { column = { expression | DEFAULT } |
          ( column [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
```

An update including a join is done with "UPDATE … FROM", e.g.:

```
UPDATE vehicles_vehicle AS v
SET price = s.price_per_vehicle
FROM shipments_shipment AS s
WHERE v.shipment_id = s.id
```

ETL (extract, transform, load) tools are used to populate whole databases. In PostgreSQL for instance:

```
COPY Professor FROM '/profs.csv' WITH FORMAT csv;
```

5.2.2 Advanced queries

There are multiple ways to do joins. One way (for a natural join) is to to specify both tables (which would result in the cross product) and then filter based on the common attribute(s), e.g.:

```
SELECT Name
FROM   Professor P, Lecture L
WHERE  P.PersNr = L.PersNr
```

Common set operations can be expressed in SQL, e.g. for union, intersection and minus we have:

```
(SELECT Name FROM Assistant)
UNION
(SELECT Name FROM Professor);

(SELECT Name FROM Assistant)
UNION ALL
(SELECT Name FROM Professor);

(SELECT Name FROM Assistant)
INTERSECT
(SELECT Name FROM Professor);

(SELECT Name FROM Assistant)
EXCEPT
(SELECT Name FROM Professor);
```

UNION ALL doesn't remove duplicates, whereas UNION does.

DISTINCT is used to eliminate duplicates. Only the rows where all the selected columns are identical are eliminated.

In a SELECT, subqueries can also be used, for instance:

```
SELECT ...
FROM   R1, (SELECT ... FROM ... WHERE) I
WHERE R1.XX = I.XX;
```

For renaming in SQL, AS is used.

Tables are not sorted by default. But one can specify it explicitly using ORDER BY (it's also possible to specify multiple columns), e.g.:

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

For existential quantification, EXISTS / NOT EXISTS / IN / NOT IN can be used. If a subquery returns NULL, the result of EXISTS is true. The result of [NOT] IN (NULL) is always empty because normal comparison with NULL fails. The subqueries can be correlated (reference the base table by alias) or uncorrelated.

The ANY operator (SOME is a synonym for ANY) compares a value to a set of values returned by a subquery (that returns one column). The ANY operator must be preceded by =, <=, >, <, > or <>. It returns true if at least one comparison returns true, false otherwise. It therefore returns always false for an empty subquery result. NULL is returned if no comparison returns true but at least one returns NULL.

The ALL operator must be preceded by one of the above operations as well and be followed by a subquery. It returns true if ANY operator meets the condition, false otherwise. It therefore returns always true for an empty subquery result. It returns NULL if no comparison returns false but at least one returns NULL.

5.2.3 Aggregation

There are multiple aggregation functions like AVG, MAX, MIN, COUNT or SUM.

AVG / MAX / MIN / SUM ignores null values (for AVG they are ignored for the count as well, meaning AVG(1,2,NULL) = 1.5 whereas AVG(1,2,0) = 1.0). COUNT doesn't ignore null values when it's used as COUNT(*), i.e. if a = 1,2,NULL then COUNT(*) = 3. If it's used on a column, NULL values are ignored, i.e. if a = 1,2,NULL then COUNT(a) = 2.

The GROUP BY statement divides the rows returned from the SELECT statement into groups. Aggregate functions can then be applied to each group. A group can also be formed by NULL, for instance we have:

```
CREATE TABLE groupby (
  a INT,
  b INT
  );
INSERT INTO groupby VALUES (1, 1), (1, 2), (2, 100), (2, NULL), (NULL, 100),
(NULL, 20);

SELECT a, SUM(b) FROM groupby
GROUP BY a;
```

We get (null, 120), (2, 100) and (1, 3) as a result.

With HAVING it's possible to subselect the groups returned by GROUP BY. So HAVING applies to summarized group records whereas WHERE applies to individual records.

5.2.4 NULL

The value NULL is used as "I don't know". NULL is a state, not a value. Because of that, one has to check with IS NULL / IS NOT NULL, = NULL will always return zero values.

We have NULL + 1 -> NULL, NULL * 0 -> NULL and NULL = NULL or NULL < 13 is unknown. SQL implements 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

5.2.5 Snapshot Semantics

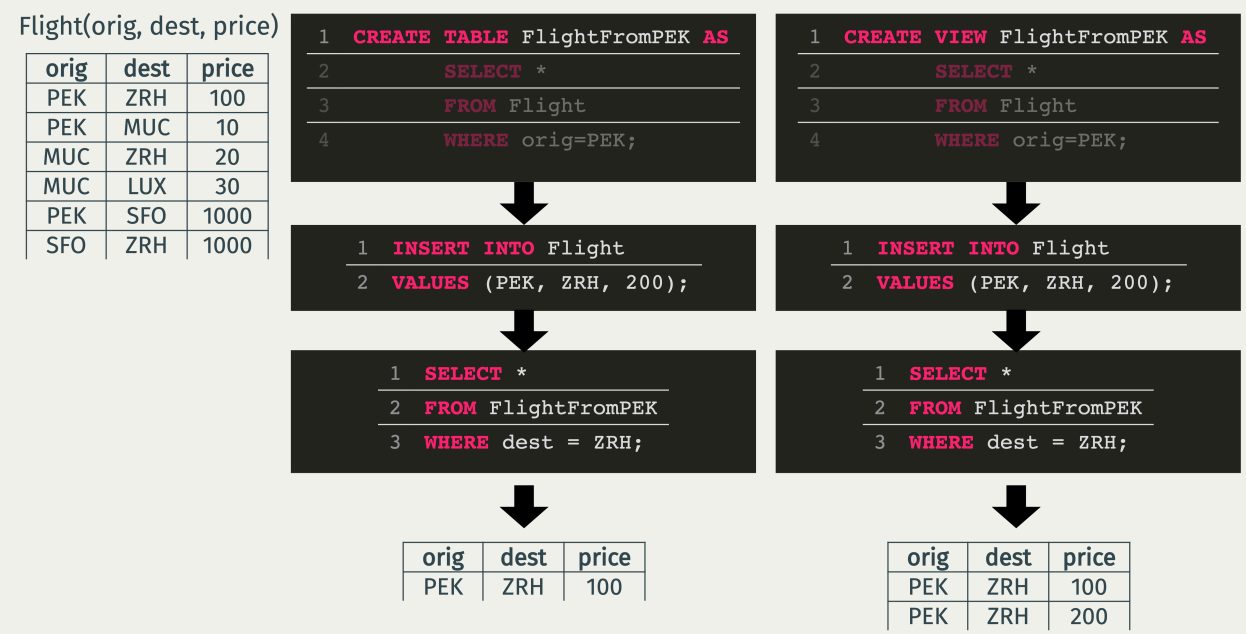
When doing changes (deletion, updates), in a first phase, all tuples which are affected by the update are marked. In a second phase, the updates are implemented on the marked tuples. This is done because otherwise, indeterministic execution of updates can happen.

5.2.6 Views

The goal of a view is to provide logical data independence (whereas relations provide physical data independence). It is a result set of a stored query on the data, which can be queried just as other database collection objects. But the table never gets materialized. The syntax to create a view is:

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

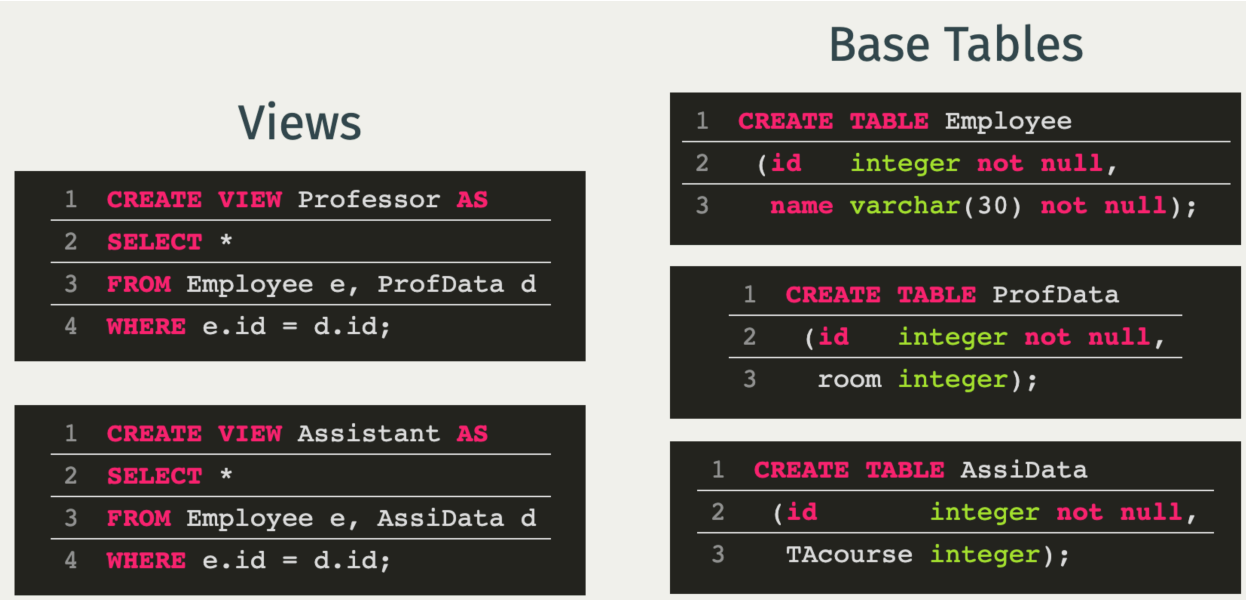
The main difference between a table that was created with CREATE TABLE AS and a view is that the contents of the table don't change when new tuples are inserted in the "base" tables whereas the view gives the updated results.



One can also think of a view as an "alias" for a query. There are multiple applications for using a view:

- Privacy: It's possible to use it for access control and preserving privacy.
- Usability: Views can be used to make writing certain queries easier.

- Generalization: Views can be used to implement generalization:



When a query with a view is evaluated, it gets rewritten. The DBMS inserts the view SQL (with an appropriate alias) and transforms the conditions (and further optimizes the query), e.g.:

```
SELECT *
FROM FlightFromPEK
WHERE dest = ZRH;
```

```
SELECT *
FROM
      (SELECT *
       FROM Flight
       WHERE orig=PEK) T
WHERE T.dest = ZRH;
```

```
1  SELECT *
2  FROM Flight T
3  WHERE T.dest = ZRH
4    AND T.orig = PEK;
```

Note: View updates NOT relevant for the exam.

It's not always possible to update a view (i.e. run an update query on a view). SQL tries to avoid indeterminism and is conservative with respect to view updates. A SQL view is updatable iff:

- 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

It's also possible to create views that involve other views. But one has to take care when dropping this view. When one specifies CASCADE while dropping, the objects that depend on the view (e.g. other views) get dropped as well, when one specifies RESTRICT (default) an error is thrown when there are objects that depend on the view.

5.2.7 Recursion

Note: NOT relevant for the exam.

Without the SQL described so far, a query like "return all ancestors of D" cannot be answered because we don't know how many generations a DB contains. The functionality that is needed to do this is to execute the same query over and over again until it converges.

The syntax of WITH looks like this:

```
WITH RECURSIVE R[(parameters,…)] AS
   (base query
     UNION [ALL]
     recursive query)
<Query involving R and other tables)
```

First, the base query is executed. In the case of UNION, the duplicates are eliminated. All remaining rows are included in the result of the recursive query and also placed in a temporary working table. As long as the temporary table isn't empty, repeat:

- Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION, discard duplicate rows / rows that duplicate any previous rows. Include all remaining rows in the result of the recursive query and place them in a temporary intermediate table.
- Replace the contents of the temporary table with the intermediate table and empty the intermediate table.

Some examples are:

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

```
WITH RECURSIVE
```

```
     R(r) AS
          (SELECT 1
              UNION
           SELECT r FROM R)
SELECT * FROM R;

WITH RECURSIVE
     R(r) AS
          (SELECT 1
              UNION
           SELECT r+1 FROM R)
SELECT * FROM R LIMIT 10;
```

5.2.8 Constraints

Besides a schema (defines the domain) and types, there are also integrity constraints. This is the way that the database makes sure changes are consistent and do not cause troubles later on or controls the content of the data and its consistency as part of the schema.

- NOT NULL constraint: Value of an attribute cannot be null, e.g. `GPA real NOT NULL`
- UNIQUE constraint: Value needs to be unique, but (an arbitrary number of) NULL values are allowed.
- KEY constraint: A certain attribute needs to be unique and NOT NULL. There can only be one primary key per relation. To define a primary key over multiple columns, `PRIMARY KEY (sID, sName)` is used. The same syntax is also used for UNIQUE over multiple columns.
- CHECK constraint: Local checking of the value of attributes, NULL & TRUE is accepted and FALSE rejected. No sub-queries are allowed in CHECK. For example:

```
CREATE TABLE Student
    (sID int PRIMARY KEY
     sName text,
     GPA real
       check(GPU <= 4.0 and GPA > 0.0),
     semester int
       check(semester <20 and semester > 0)
);
```

- REFERENTIAL constraint: Refer to tuple from a different relation. For every foreign key, one of the following two conditions must hold:
  - The value of the foreign key is NULL
  - The referenced tuple must exist

  The syntax is:

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

CREATE TABLE T
  (...,
   kb varchar(30) reference R(b));

FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
```

  In absence of a column list, the primary key of the referenced table is used. The referenced columns must either be a primary key or form a unique constraint.

There are different actions that can be specified on a delete / update of the referenced value:

- 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 to a default value

There can be a difference between restrict / no action for deferrable constraints. No cation allows the check to be deferred until after the transaction, whereas restrict doesn't. The default behavior (if nothing is specified is No action).

To enforce a 1:1 relationship, an intermediate table has to be created, i.e.:

```
CREATE TABLE Employee
   (emp_id VARCHAR(20) NOT NULL UNIQUE);

CREATE TABLE Office
   (room varchar(20) NOT NULL UNIQUE);

CREATE TABLE HasOffice
   (emp_id NOT NULL UNIQUE
         REFERENCES Employee (emp_id),
    room NOT NULL UNIQUE
         REFERENCES Office (room)
);
```

5.3 Relational Design Theory

The goal of relational design theory is to assess (with regard to redundancy and integrity constraints) and / or improve the quality of a schema and to give rules for constructing "high-quality" schemas.

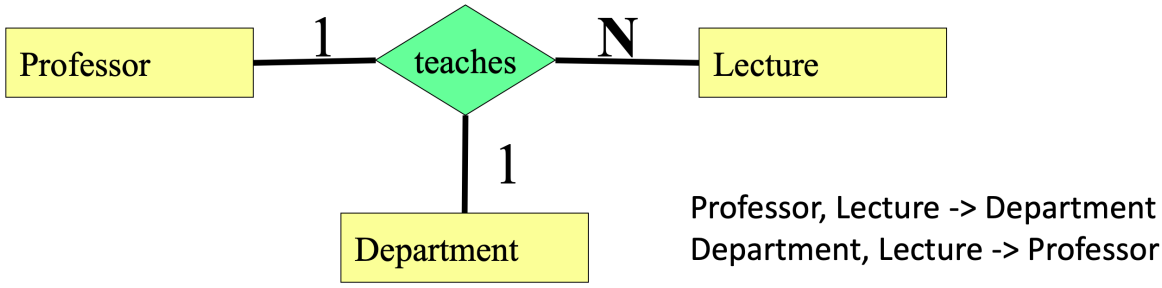
Problems with redundancy are a waste of storage space, additional work to keep multiple copies of data consistent and additional code to keep multiple copies of data consistent.

Bad Schemas cause Update-, Insert- or Delete-Anomalies. But there are some movements to Multi-version (NoSQL) databases (no delete; just set flag to "deleted", no update in place; just create a new version). Insert anomalies still exist but aren't a big problem.

5.3.1 Functional Dependencies

Let R be a schema and $\alpha \subseteq \mathcal{R}, \quad \beta \subseteq \mathcal{R}$. We say that there exists a functional dependency $\alpha \to \beta$ iff $\forall r, s \in R : r.\alpha = s.\alpha \Rightarrow r.\beta = s.\beta$. For example in a family tree, one could have {Child} -> {Father, Mother} or {Child, Grandma} -> {Grandpa}. $\alpha \subseteq \mathcal{R}$ is a superkey iff $\alpha \to R$. $\alpha \to \beta$ is minimal (noted as $\alpha \to^{\cdot} \beta$) iff $\forall A \in \alpha : \neg((\alpha - \{A\}) \to \beta)$. $\alpha \subseteq \mathcal{R}$ is a key (or candidate key) iff $\alpha \to^{\cdot} \mathcal{R}$.

Some functional dependencies are derived from the cardinality information, e.g.:



For deriving functional dependencies, there are the Armstrong axioms:

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

Other (derived) rules are:

- Union of FDs: $\alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma \Rightarrow \alpha \rightarrow \beta\gamma$
- Decomposition: $\alpha \rightarrow \beta\gamma \Rightarrow \alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma$
- Pseudo transitivity: $\alpha \rightarrow \beta \wedge \gamma\beta \rightarrow \delta \Rightarrow \alpha\gamma \rightarrow \delta$

The following algorithm calculates the closure of attributes. Meaning, given a set of FDs and a set α of attributes, $\alpha+$ is calculated such that $\alpha \rightarrow \alpha+$

```
Closure(F, α)
    result := α                                  // Reflexivity
    while (result has changed) do
        foreach FD: β → γ in F do                // Transitivity
            if β ⊆ result then result := result ∪ γ
    output(result)
```

Fc is a minimal basis of F (a set of functional dependencies) iff:

1. $\mathrm{Fc} \equiv \mathrm{F}$ (the closure of all attribute set is the same in Fc and F)
2. All functional dependencies in Fc are minimal:

$\forall A \in \alpha : (\mathrm{Fc} - (\alpha \rightarrow \beta) \cup ((\alpha - \{\mathrm{A}\}) \rightarrow \beta)) \not\equiv \mathrm{Fc} \quad \forall B \in \beta : (\mathrm{Fc} - (\alpha \rightarrow \beta) \cup (\alpha \rightarrow (\beta - \{\mathrm{B}\}))) \not\equiv \mathrm{Fc}$

3. In Fc, there are no two functional dependencies with the same left side (which can be achieved by applying the Union rule)

The minimal basis is computed like this:

1. Reduction of left sides of FDs. Let $\alpha \rightarrow \beta \in F$, $A \in \alpha$:
   if $\beta \subseteq$ Closure(F, $\alpha - A$)
   then replace $\alpha \rightarrow \beta$ **with** $(\alpha - A) \rightarrow \beta$ in F

2. Reduction of right sides of FDs. Let $\alpha \rightarrow \beta \in F$, $B \in \beta$:
   if $B \in$ Closure$(F - (\alpha \rightarrow \beta) \cup (\alpha \rightarrow (\beta - B)), \alpha)$
   then replace $\alpha \rightarrow \beta$ **with** $\alpha \rightarrow (\beta - B)$ in F

3. Remove FDs: $\alpha \rightarrow \varnothing$ (clean-up of Step 2)

4. Apply Union rule to FDs with the same left side.

Bad relations combine several concepts so they are decomposed so that each concept is in one relation: $R \rightarrow R_1, \ldots R_n$ A decomposition is lossless if $\mathcal{R} = \mathcal{R}_1 \bowtie \mathcal{R}_2 \bowtie \ldots \bowtie R_n$. Decompositions preserve dependencies if $\mathrm{FD}(\mathcal{R})+ = (\mathrm{FD}(\mathcal{R}_1) \cup \ldots \cup \mathrm{FD}(\mathcal{R}_n))+$.

Let $\mathcal{R} = R1 \cup R2$ (i.e. $\mathrm{R1} := \Pi_{R1}(\mathrm{R})$ and $\mathrm{R2} := \Pi_{R2}(\mathrm{R})$). A decomposition is lossless if $(\mathcal{R}1 \cap R2) \rightarrow R1$ or $(R1 \cap R2) \rightarrow R2$

5.3.2 Normal Forms

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. For each normal form, one can decide (given a relational schema R and a set of functional dependencies FD) whether (R, FD) satisfies a given normal form. If (R, FD) satisfies a given normal form, one can define a set of properties that it will have. And given a relational schema R and a set of functional dependencies FD, one can generate a new schema R' such that (R', FD) satisfies a given normal form.

The intuition behind 1NF, 2NF, 3NF and BCNF is "don't store the same fact twice" whereas the intuition behind 4NF is don't store unrelated information in the same relation.

5.3.2.1 1NF: First Normal Form

A relation is in first normal form if and only if it contains only atomic domains. SQL3 / XML allows Non-First Normal Form by introducing the array type, so a relation doesn't have to be in first normal form.

5.3.2.2 2NF: Second Normal Form

R is in 2NF iff every non-key attribute is minimally dependent on every key (meaning no attribute depends on part of a key).

5.3.2.3 3NF: Third Normal Form

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

- $B \in \alpha$ (i.e. $\alpha \rightarrow B$ is trivial)
- B is an attribute of at least one key
- α is a superkey of R (meaning some candidate key is a subset of it)

The following synthesis algorithm is used for 3NF:

> **Algorithm 26:** Synthesis of Third-Normal-Form Relations With a Lossless Join and Dependency Preservation.
>
> **INPUT**: A relation R and a set F of functional dependencies that hold for R.
>
> **OUTPUT**: A decomposition of R into a collection of relations, each of which is in 3NF. The decomposition has the lossless-join and dependency-preservation properties.
>
> **METHOD**: Perform the following steps:
>
> 1. Find a minimal basis for F, say G.
> 2. For each functional dependency $X \rightarrow A$ in G, use XA as the schema of one of the relations in the decomposition.
> 3. If none of the relation schemas from Step 2 is a superkey for R, add another relation whose schema is a key for R.

The relations where all columns are a subset of another relation can be dropped in the end. The third step means to check if there exist a relation that contains all columns of a key. If not, one has to be added.

This synthesis algorithm preserves functional dependencies.

When we transform an ER model into a relational model, we automatically generate a model in 3NF (because the only FDs are caused by cardinality constraints).

5.3.2.4 BCNF: Boyce-Codd Normal Form

R is in BCNF iff $^{\forall \alpha \rightarrow B}$, at least one condition holds:

- $B \in \alpha$ (i.e. $\alpha \rightarrow B$ is trivial)
- α is a superkey of R (meaning some candidate key is a subset of it)

It is always possible to turn a schema into BCNF by the decomposition algorithm:

1. `result` = $\{\mathcal{R}\}$
2. while $(\exists \mathcal{R}_i \in$ `result` : $\mathcal{R}_i$ *is not* $BCNF)$
3. Let $\alpha \rightarrow \beta$ be the evil FD
4. $\mathcal{R}_i^{(1)} = \alpha \cup \beta$
5. $\mathcal{R}_i^{(2)} = \mathcal{R}_i - \beta$
6. `result` $\leftarrow$ (`result` $- \mathcal{R}_i) \cup \{\mathcal{R}_i^{(1)}, \mathcal{R}_i^{(2)}\}$
7. Output `result`

One needs to check the whole closure of FDs, not only the set of FDs F. The decomposition does not preserve all functional dependencies.

BCNF has no redundancy (in contrast to 3NF), but doesn't model all functional dependencies.

5.3.2.5 4NF: Fourth Normal Form

Multi value dependencies A -> -> B and A -> -> C mean intuitively that the value of B doesn't have impact on the value of C and B / C can take multiple values for the same a.

Formally, $\alpha \twoheadrightarrow \beta$ for $R(\alpha, \beta, \gamma)$, iff:

- $\forall t_1, t_2 \in R, t_1.\alpha = t_2.\alpha \implies \exists t_3, t_4 \in R$:
  - $t_3.\alpha = t_4.\alpha = t_1.\alpha = t_2.\alpha$
  - $t_3.\beta = t_1.\beta, t_4.\beta = t_2.\beta$
  - $t_3.\gamma = t_2.\gamma, t_4.\gamma = t_1.\gamma$

For example {PersNr} -> -> {Language} and {PersNr} -> -> {Programming}:

Skills		
PersNr	Language	Programming
3002	Greek	C
3002	Latin	Pascal
3002	Greek	Pascal
3002	Latin	C
3005	German	Ada

Let $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$, $R_1 = \Pi_{\mathcal{R}_1}(R)$ and $R_2 = \Pi_{\mathcal{R}_2}(R)$. Decomposing R into {R1, R2} is lossless iff:

$\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_1$ or $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_2$.

The laws of MVDs are:

- Trivial MVDs: $\alpha \twoheadrightarrow \mathcal{R}$, $\alpha \twoheadrightarrow \mathcal{R} - \alpha$ and $\beta \subseteq \alpha \implies \alpha \twoheadrightarrow \beta$
- Promotion: $\alpha \to \beta \implies \alpha \twoheadrightarrow \beta$ (but not in the other direction)
- Complement: $\alpha \twoheadrightarrow \beta \implies \alpha \twoheadrightarrow \mathcal{R} - \beta - \alpha$
- Multi-value augmentation: $\alpha \twoheadrightarrow \beta \wedge (\delta \subseteq \gamma) \implies \alpha\gamma \to \beta\delta$
- Multi-value transitivity: $\alpha \twoheadrightarrow \beta \wedge \beta \twoheadrightarrow \gamma \implies \alpha \twoheadrightarrow \gamma$

R is in 4NF iff $\forall \alpha \twoheadrightarrow \beta$, at least one condition holds:

- $\alpha \twoheadrightarrow \beta$ is trivial
- α is a superkey of R.

When R is in 4NF, it is also in BCNF.

The decomposition algorithm for 4NF works like this:

- Input: $\mathcal{R}$
- Output: $\mathcal{R}_1 ... \mathcal{R}_n$ such that
  - $\mathcal{R}_1 ... \mathcal{R}_n$ is a lossless decomposition of $\mathcal{R}$
  - $\mathcal{R}_1 ... \mathcal{R}_n$ are in 4NF

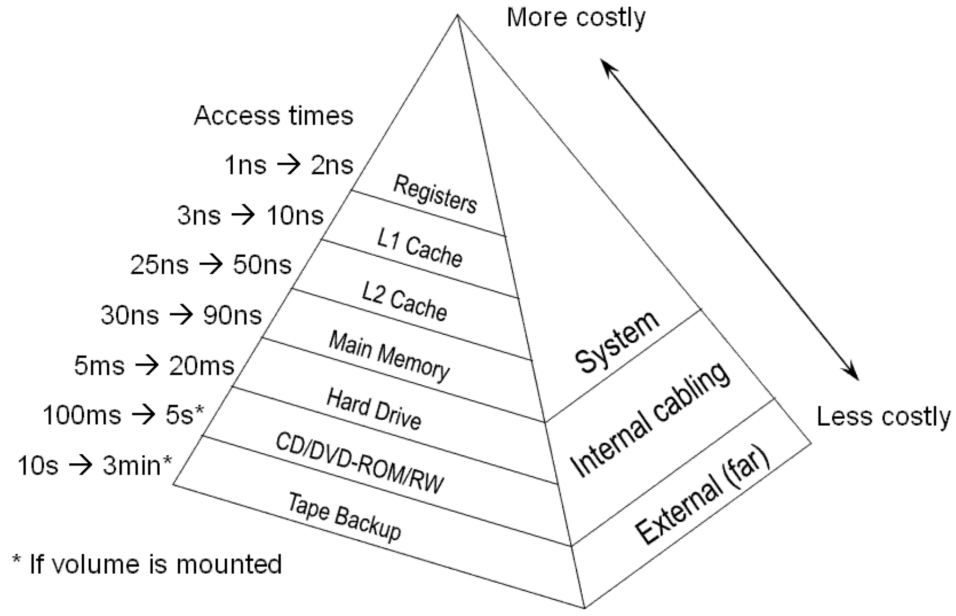
1. `result` = $\{\mathcal{R}\}$
2. while ($\exists \mathcal{R}_i \in$ `result` : $\mathcal{R}_i$ *is not* $4NF$)
3. Let $\alpha \rightarrow\rightarrow \beta$ be the evil MVD
4. $\mathcal{R}_i^{(1)} = \alpha \cup \beta$
5. $\mathcal{R}_i^{(2)} = \mathcal{R}_i - \beta$
6. `result` $\leftarrow$ (`result` $- \mathcal{R}_i$) $\cup \{\mathcal{R}_i^{(1)}, \mathcal{R}_i^{(2)}\}$
7. Output `result`

5.3.3 System Implementation

6 System overview

A database gets an SQL statements as input and outputs tuples. It translates the SQL query into a set of get / put requests to the backend storage and then extracts, processes and transforms tuples from blocks. There are tons of optimization potential in this process. The focus will be on a relational database with a disk-oriented architecture which implies that the disk is larger but slower than memory and the disk favors a different access pattern than memory.

The storage of a modern computer is structured hierarchically:



A simplified performance model for a hard disk (with seek / rotate time t_s, t_r and transfer time t_{tr}): D random accesses take $D(t_s + t_r + t_{tr})$ whereas D sequential accesses take $t_s + t_r + Dt_{tr}$

Because of this, often times algorithms that have a sequential access pattern are preferred.

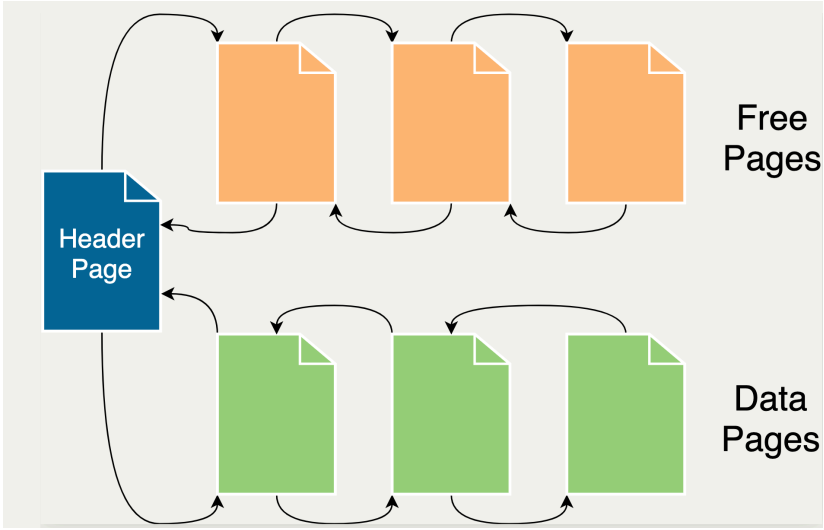
7 Disk Manager

A DBMS stores a database as one or more files on disk. Some DBs use a single file (e.g. sqlite) whereas other use a collection of files (e.g. PostgreSQL). Nowadays, most DBs use the filesystem provided by the OS (which wasn't always like that).

The storage / disk manager is responsible for maintaining a database's files. It organizes the files as a collection of pages where a page is a fixed-size block of data. It contains tuples, metadata, indexes, log records, … Each page has a unique identifier (the page id). Tuples can have a fixed or a variable size. There are multiple storage structures:

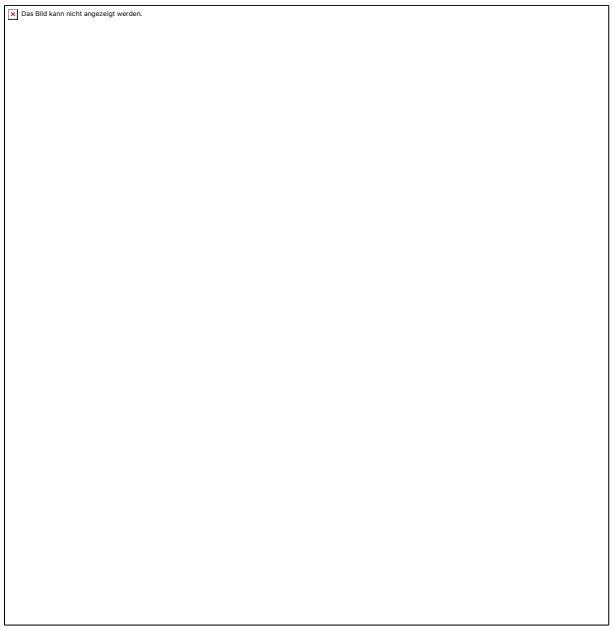
7.1 Heap file

A heap file is an unordered collection of pages where tuples are stored in random order. They can be implemented with a linked list or a page directory. If they are implemented with a linked list, they look like this:



The header page stores two pointers, one to the free page list and one to the data page list.

If they are implemented with a page directory, they look like this:



The directory is a set of header pages that contain pointer to data pages (and also the number of free slots).

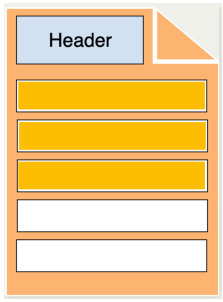
A disadvantage of using linked lists is that there will be free spaces on data pages here and there (in the naïve implementation).

For a heap file with D pages (where we assume that the directory is already in memory), the insertion takes $t_{s+r} + 2 \times t_{trans}$ (seeking / rotation for the data page, reading the data page and writing the data page back). For finding a record (going through all data pages), we have in expectation a time of $\frac{D}{2}(t_{s+r} + t_{trans})$ if the pages are randomly allocated and $t_{s+r} + \frac{D}{2} t_{trans}$ if they are sequentially allocated. A whole scan takes $D(t_{s+r} + t_{trans})$ if the pages are randomly allocated and $t_{s+r} + D t_{trans}$ if they are sequentially allocated on disk.

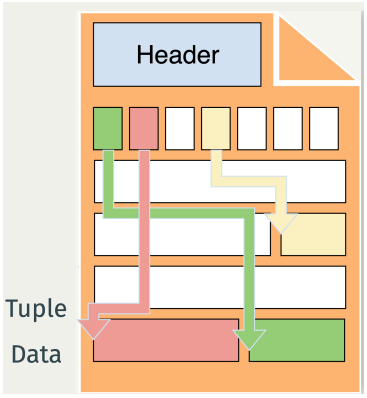
Each page contains a header of meta-data like the page size, DBMS version, compression / encryption info or a checksum. For actually storing the tuples, there are different ways (page layouts):

7.2 Page Layout

The naïve strategy is to have fixed-length slot and keep track of the number of tuples in the header:



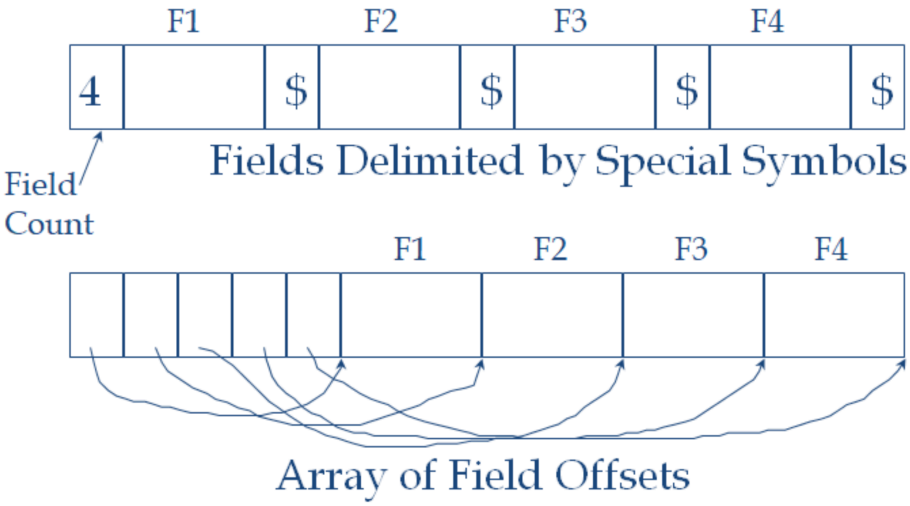
But not all tuples are of the same length (e.g. VARCHAR's). The solution for this is to use slotted pages. At the beginning of the page, there is a slot array that is used as a lookup table (contains pointers / relative locations to the different entries). The record identifier (that is used by indexes to ref. records) is then often the combination of page number and slot number so that the pointer / offset in the slot array can be accessed directly. Because of that, indexes don't need to get updated when data grows / shrinks / is moved in the page, only the entry in the slot array.



7.3 Tuple Layout

For fixed length field, a direct access is possible. For variable length fields, many systems store the length (pointer) as part of a fixed-length field and the payload is stored in a variable-length field. So the access is in two steps: Retrieve the pointer and chase the pointer. NULL values are usually indicated by a 1 / flag in the bitmap:

So for fixed length fields, one doesn't need to scan the record to get the address of the i-th entry (can be calculated statically). Multiple variable length fields can be delimited by special symbols or an array of field offsets can be used (offers direct access to the i-th field):



7.4 Column Store

Besides storing entries per row, a DBMS can also use a column store and store values per column. This can reduce the amount of wasted I/O because the DBMS only reads the data that it needs and facilitates data compression (similar values / data types per column). But it's slow for point queries (reading all information / attributes of a single row), inserts, updates and deletes.

8 Buffer Pool Management

The goal of a buffer manager is to provide the "illusion" that all data are in DRAM. But because RAM space is limited, pages have to be loaded / flushed out (if dirty) or evicted. The key design decision is how to choose which page to evict. There are multiple strategies:

- **LRU (Least Recently Used):**
  - For each page in the buffer pool, time of the last unpin action (page is pinned if it contains a requested record, unpinned when the request is fulfilled) is recorded.
  - Replace the frame which has the oldest (earliest) time.
- **MRU (Most Recently Used):**
  - Time of the last unpin action is recorded as well, but the frame with the newst (latest) time is replaced.

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

A more advanced buffer manager is DBMin. It makes use of the fact that each query is composed of a set of operators (like scan, fetch, index, …). For each operator of each query, memory is allocated, and the replacement policy is adjusted according to the access pattern. This allows to do load control and further optimizations.

Meta-data of databases is stored in tables and accessed internally. The meta-data includes the schema, table spaces (files used to store the database), histograms, parameters (e.g. for query optimization like cost of I/O), compiled queries (used for prepared statements / ODBC), configuration, users and statistics.

Many DBMS use OS's file system, which has its own buffers. This can cause problems because of higher latency, cost (more I/O) and utilization (same page held twice in main memory).

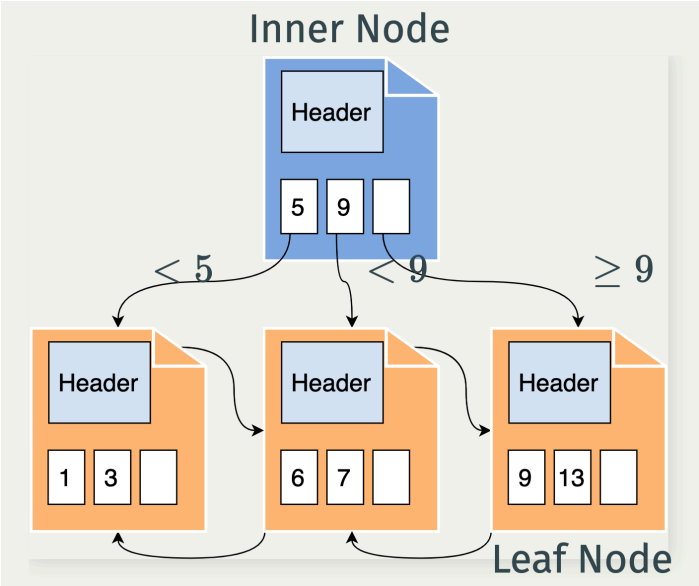
9 Access Methods

An index is a function that takes as input the value of one attribute (x) and some data structure and gives the record id, i.e. $f(x, M) \mapsto rid(\text{location})$. There are different types of data structures for M:

9.1 B-Tree (B+ Tree)

A B+ Tree is a self-balancing tree data structure that keeps data sorted. It has a runtime of $O(\log n)$ for search, insertion and deletion. It is a generalization of a binary search tree in which each node contains more than two children and is optimized for systems that read and write large blocks of data.

More concretely, it is a perfectly balanced M-way search tree where every inner node other than the root is at least half-full and each inner node with k keys has k+1 non-null children. Each node is a page.



The value can be the rid or the actual tuple data (but there can only be one such an index).

If we have nodes with a size of M and N tuples, finding a single key (point query) takes $\log_M N + 1$ and a range query $\log_M N + \#\ \text{tuples}\ /\ \text{pagesize}$ I/O.

To insert, the right leaf L is searched, and data is inserted in L. If it hasn't enough space, it is split and the key is inserted to the parent of L. When removing, one needs to take care if L is still at least half-full. If not, pages need to be merged or keys borrowed from neighbors.

B+ trees are very powerful, a typical fill-factor is 67%. When one node has 200 keys, a height 4 B+ tree has a capacity of 312M entries.

But B-trees are not better for all types of queries. For point queries / queries with a very low selectivity, they perform usually very well. But for queries with a very high selectivity, a scan of the heap file can be better.

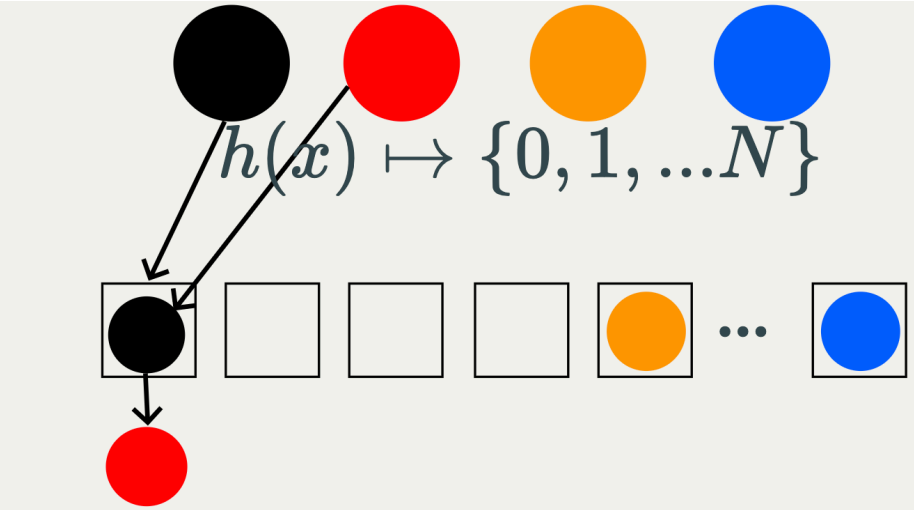
For bulk building of a B-tree, one can insert tuples one by one (slow) or sort the whole relation and then insert the appropriate keys (build the tree bottom-up).

9.2 Hash Table

A hash function maps a value x to a value 0,1,…N. The entry can then be fetched from the table at position h(x) in O(1). One wants to achieve x =! y => h(x) != h(y) but this can never be achieved. So we need to deal with hash collisions. One approach for closed hashing (where all

keys are stored in the hash table itself) is linear probing. On a collision, the value is just inserted into the next space. When deleting, a placeholder needs to be inserted. So in theory, linear probing doesn't guarantee O(1) (but the expected time is for truly random hash functions).

An approach for open hashing (which is used when we don't know how many tuples we need to support) is chained hashing. In each table entry, a linked list is stored and entries are just appended to this list. So on a collision, the list just gets longer:



10 Operator Execution

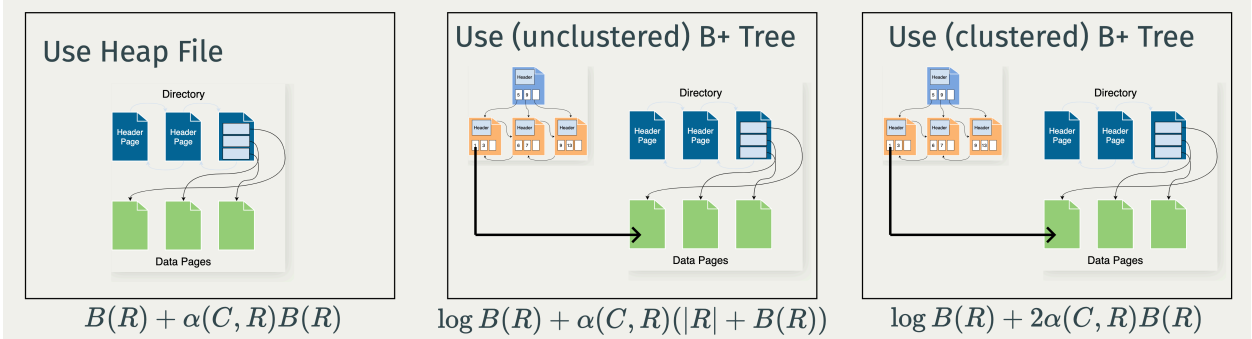
We assume that a relation R has $|R|$ tuples and B(R) pages and that the buffer size is M pages. $\alpha(C,R)$ is a key constant called selectivity i.e. the number of tuples in R that satisfies C divided by $|R|$.

10.1 Select

For a select, the heap file is scanned and for each tuple, the condition C is checked. I/O cost to read is B(R) and to write (the result into the "new" pages for output) $\alpha(C,R)B(R)$, in total therefore $B(R)+\alpha(C,R)B(R)$.

When C is =, < or >; an index scan can be used. For an unclustered B+ tree, the right leaf node has to be found (I/O cost $\log B(R)$), the tuples have to be fetched in the heap file (IO of $\alpha(C,R)|R|$) and the result has be written ($\alpha(C,R)B(R)$). The total I/O cost is therefore $\log B(R)+\alpha(C,R)(|R|+B(R))$. For a clustered B+ tree, the cost to fetch tuples is reduced to $\alpha(C,R)B(R)$, so we have in total $\log B(R)+2\alpha(C,R)B(R)$

As we can see, it's all about selectivity and scan is not always worse than B-tree:



10.2 Sorting

Quick sort isn't designed to be I/O efficient, so one needs to use other variations for sorting on disk. One approach is external sort. There is a B-page buffer and N pages that need to be sorted (where N >> B is assumed, N < B is trivial). In a first phase, small chunks of data that fit in memory are sorted. In the second phase, these sorted sub-files are combined (B-1 at a time) into a single larger file. In total, there are $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$ passes over the data (1 for the sorting, the rest for the merging).

10.3 Join

There are multiple execution strategies for joins (the I/O to write out is ignored for all join algorithms):

10.3.1 Nested Loop Join

This strategy simply loops over each tuples in the different relations in a nested loop, i.e.:

```
foreach tuple r in R:
   for each tuple s in S:
      check theta(r, s), if true, output
```

The cost (in general) is B(R) + |R| * B(S) and the buffer needs to have a size of (at least) 2. Because of this cost, the join order matters and the smaller relation should be in the outer loop.

When the buffer size is larger or equal than B(S) + 1, the cost is B(R) + B(S) (because the whole relation S can be kept in the buffer) and in this case, we want the larger relation to be in the outer loop.

A variation is the block nested loop join which partitions the relations into blocks and acts on them:

```
foreach block BR in R:
   foreach block BS in S:
      foreach tuple r in BR:
         foreach tuple s in BS:
            check theta(r, s), if true, output
```

Here, the cost (for a buffer size of 2) is B(R) + B(R) * B(S). For a buffer size of M, if we partition S into $\lceil B(S)/M-2 \rceil$ partitions, the cost is $B(S) + B(R) \times \lceil B(S)/(M-2) \rceil$ (so B(S) + B(R) for M=B(S) + 2).

Another option is to use an index in the loop, i.e.:

```
foreach tuple r in R:
   foreach tuple s in IndexScan(r, S):
      output
```

The cost is B(R) + |R| * C where c is the lookup cost in the index.

10.3.2 Sort Merge Join

When both relations are sorted, both relations can be scanned "in parallel" (i.e. traversing / advancing a pointer). The cost is B(S) + B(R) + sort(S) + sort(R).

Sort Merge is useful when data is already sorted or other operators need the result to be sorted anyway.

10.3.3 Hash Join

First, a hash table is built and then probed while iterating, i.e.:

```
build Hash Table HT for S
foreach tuple r in R:
```

```
output, if h(r) in HT
```

To solve the issue of the hash table getting too big, Grace Hash Join can be used. This algorithm partitions both S and R via a hash function. It then loads pairs of partitions into memory, builds a hash table for the smaller relation and probes the other relation for matches with the current hash table. Because the partitions were formed by hashing on the join key, it must be the case that any join output tuples must belong to the same partition. The cost is 3(B(S) + B(R))

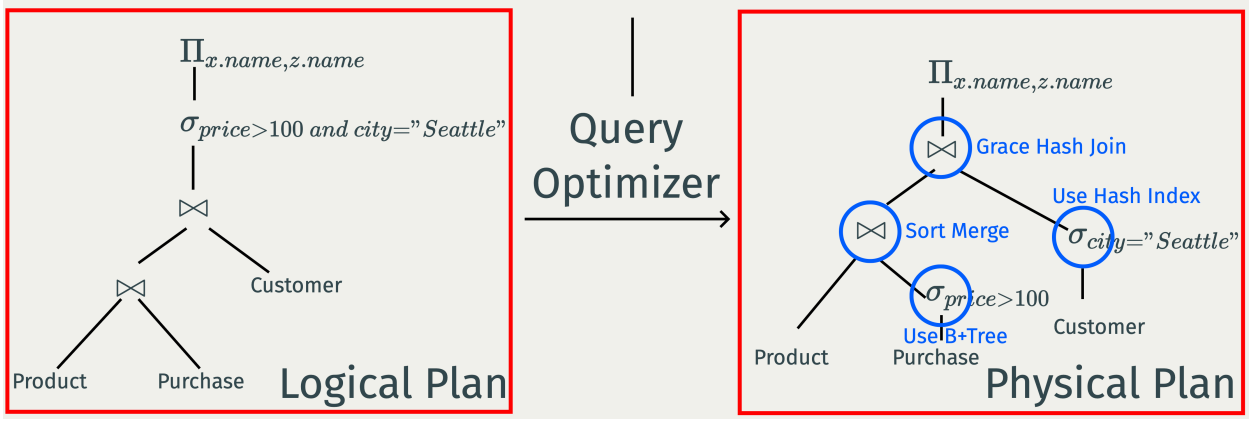
10.3.4 Overview

- $B(R) = 500, |R| = 40,000$
- $B(S) = 1,000, |S| = 100,000$
- $M = 100$

Nested Loop Join: $B(R) + \|R\| \times B(S)$	40,000,500 I/Os	4000 secs
Block Nested Loop Join: $B(S) + B(R) \times \lceil B(S)/(M-2) \rceil$	6,102 I/Os	0.61 secs
Index Nested Loop Join: $B(R) + \|R\| \times C \quad (assume\ C = 1)$	40,500 I/Os	4.05 secs
Sort Merge Join: $B(S) + B(R) + sort(S) + sort(R)$	5,849 I/Os	0.58 secs
Hash Join (Enough Memory): $3(B(S) + B(R))$	4,500 I/Os	0.45 secs
Again, DBMS will choose for you (user)	1 I/O ~ 0.1ms	

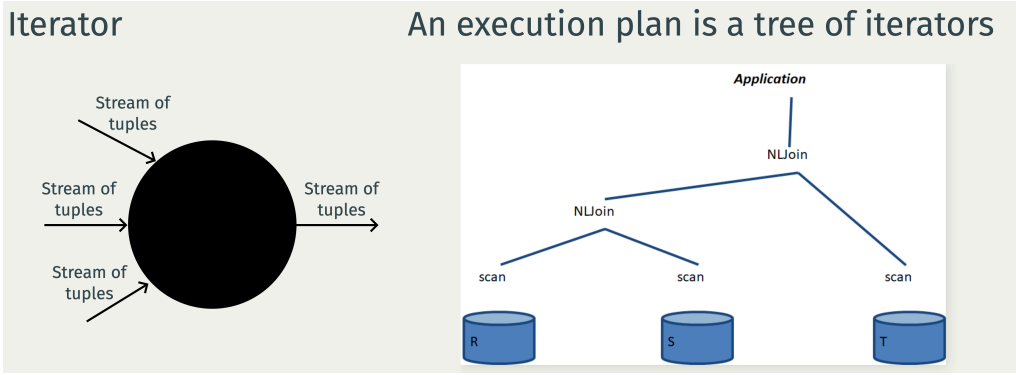
11 Query Optimization

The user gives the database a logical plan (by issuing a SQL query). The job of the query optimizer is to turn this plan into a physical plan:



11.1 Execution Model of Physical Plans

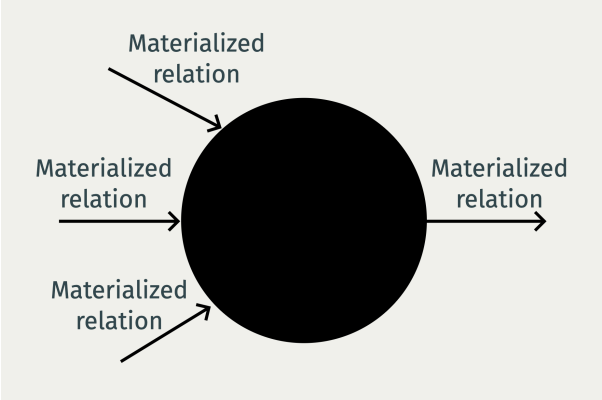
An iterator is a operator that takes a stream of tuples and outputs a stream of tuples. An execution plan is a tree of iterators:



The principle is that data flows bottom up in a plan and control (e.g. a next() call) flows top down. Advantages of the iterator model are that it is a generic interface for all operators, it's easy to implement iterators, there are no overheads in terms of main memory, pipelining is supported, and parallelism / distribution is supported by special iterators. Disadvantages are a high overhead of method calls and poor instruction cache locality.

Today, often block-at-a-time instead of tuple-at-a-time is used.

In the materialization model, each operator processes its input all at once and then emits its output all at once:



This model is good when the intermediate result is not much larger than the final result (otherwise there's a I/O overhead).

The vectorization model is similar to the iterator model, but each operator returns a batch of tuples. This greatly reduces the number of invocations per operator and allows for operators to use vectorized instructions to process batches of tuples.

11.2 Equivalent Plans

An important question when doing query optimization is how to define the space equivalent plans. There are many query rewriting rules that generate equivalent plans. A query rewriting rule takes as an input a relational algebra expression E, outputs a relational algebra expression E′. The property of E being equivalent to E′ means $\forall I \in \mathcal{I}, E(I) = E'(I)$ where $\mathcal{I}$ is all possible DB instances.

Some rules are:

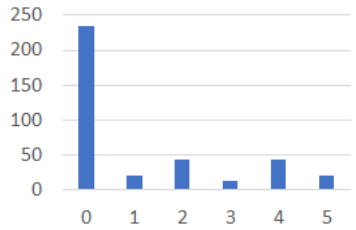
1. Conjunctive selection operations can be deconstructed into a sequence of individual selections: $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right)$
2. Selection operations are commutative: $\sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right) = \sigma_{\theta_2}\left(\sigma_{\theta_1}(E)\right)$

3. Only the last in a sequence of projection operations is needed (t1 must be a subset of t2, otherwise the result is empty anyways): $\Pi_{t_1}\left(\Pi_{t_2}(E)\right) = \Pi_{t_1}(E)$
4. Selection can be combined with cartesian products and theta joins:
   $\sigma_\theta\,(E_1 \times E_2) = E_1 \bowtie_\theta E_2 \quad \sigma_{\theta_1}\,(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
5. Theta-join operators (and natural joins) are commutative: $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
6. Natural join operations are associative: $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$ Theta joins are associative in the following manner:
   $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$
7. Pushdown selection $\sigma_\theta\,(E_1 \bowtie E_2) = \sigma_\theta\,(E_1) \bowtie (E_2)$ if θ only involves attributes in E_1
8. Projections distributes over theta joins like that (if θ only contains attributes from $L_1 \cup L_2$): $\Pi_{L_1 \cup L_2}\,(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}\,(E_1)) \bowtie_\theta (\Pi_{L_2}\,(E_2))$
9. The set operations union and intersection are commutative (but not the set difference) and associative.
10. The selection operation distributes over $\cup, \cap$ and -:
    $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$
    $\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$
    $\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$
11. Projection distributes over union: $\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$

11.3 Performance estimation

We need to estimate the performance of a given physical plan without actually running the query. As we saw in the previous chapter, the selectivity is crucial for determining performance (e.g. of a join). We can estimate the selectivity and things like $B(R \bowtie S)$ from the cardinality of a given operator, i.e. how many tuples we expect the operator to return.

For cardinality estimation, different summarization techniques (that are done once and re-used for many queries) are used. One possibility is a histogram, e.g. for discrete values:



When there are multiple attributes, it becomes harder because there can be correlation between attributes. The baseline assumption is that attributes are independent, then the number of occurrences is just (#attr1 * #attr2)/total # tuples.

For continuous values, the histogram consists of bins and we assume that values are uniformly distributed within each bin.

If correlation is very important (e.g. for the estimation of some queries), multi-dimensional histograms can be built.

To then estimate the performance of a physical plan, the cardinality of each operator is estimated and the cost model (e.g. the one described in the previous chapter) is used to estimate the performance.

11.4 Finding the best plan

Searching the space of all possible plans is usually not feasible because the search space is huge (already exponential just for changing the order of joins).

IBM used dynamic programming for the problem. In a first pass, the best 1-relation plan is searched, then the best 2-relation plan (best way to join results of each 1-relation plan to another relation), etc… until in the n-th pass, the best way to join the result of a (n-1)-relation plan to the n'th relation is searched. The problem is that the search space is still possibly too small (e.g. when the result gets sorted later, a Sort Merge Join would be better, but the dynamic programming approach gives Hash Join because he considers only the pair-wise join).

Another approach is to do heuristic-based optimization, i.e. use heuristics to reduce the number of choices that must be made in a cost-based fashion. This approach transforms the query-tree by using a set of rules that typically improve execution performance.

12 Transactions

When multiple groups of SQL statements are running at the same time, we want the effect as if they are executed sequentially. 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.

The relevant syntax for using transactions is:

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

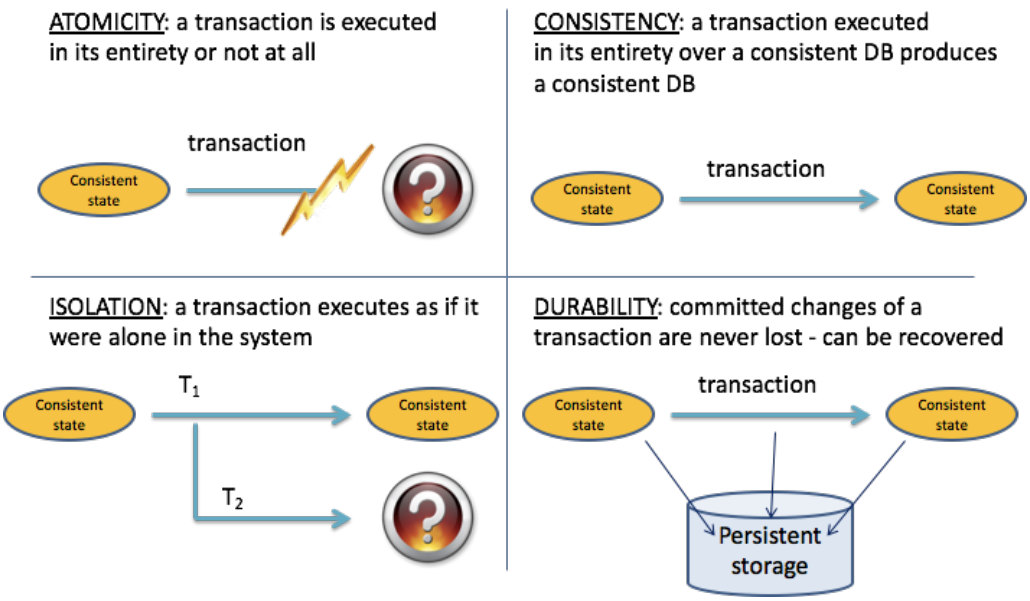
Or

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

AUTOCOMMIT turns each SQL into a transaction.

In a more formal definition of a transaction, a database is a fixed set of named data objects (e.g. A, B, C …). A transaction is a sequence of read and write operations where <- R(A) means reading the object A into variable a and W(B, b) means writing the value of variable b into object B. A new transaction starts with BEGIN and stops with either COMMIT or ABORT. If it stops with COMMIT, all changes are saved. If it stops with ABORT (which can be initiated by the user or the DBMS), all changes are undone so that it is as if the transaction never executed at all.

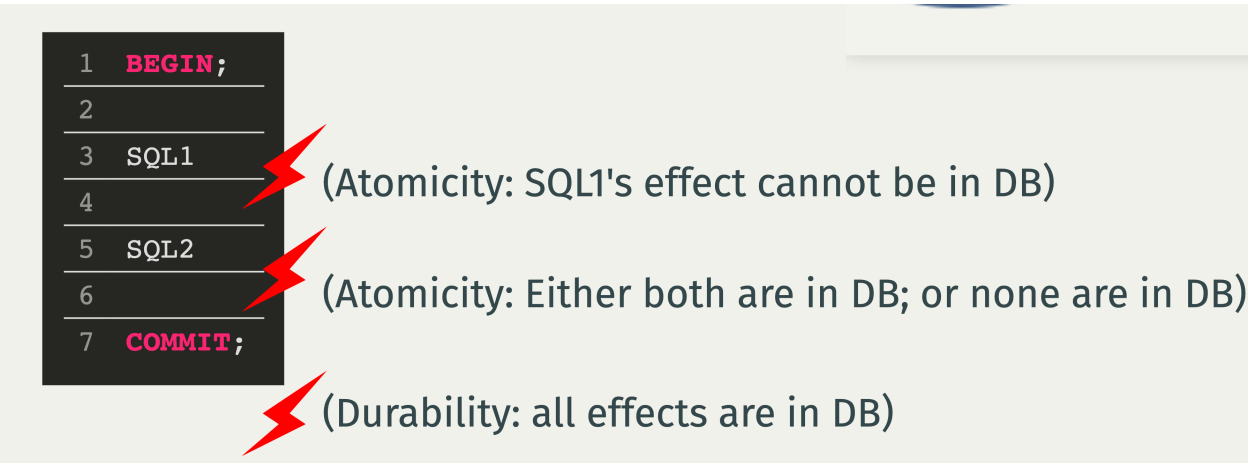
ACID is a set of properties of database transactions:



Isolation means that operations might be interleaved but execution must be equivalent to some sequential order of all transactions (the different orders may result in a different result, but this is fine as long as they are equivalent to some sequential order; enforcing a certain result / sequential order is the job of the application).

Durability means that if the system crashes after the transaction commits, all effects of the transaction remain in the database.

Atomicity means that each transaction is "all-or-nothing" and never left half done:

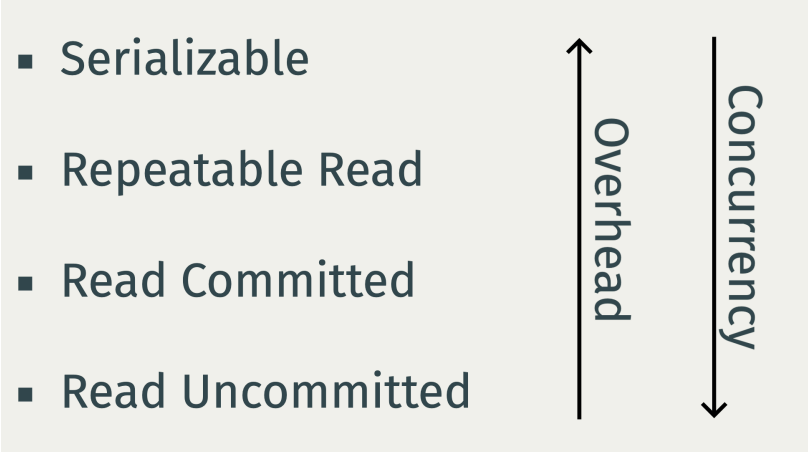


Consistency means that for each client and each transaction, all integrity constraints hold when the transaction begins, and all constraints hold when the transaction ends. Constraints can be violated within a single transaction. The constraint checking can be deferred which results in different granularities of the checking (per row, statement or transaction):

	NOT DEFERRABLE	INITIALLY IMMEDIATE	INITIALLY DEFERRED
CHECK	row	row	row
NOT NULL	row	row	row
UNIQUE	row	statement	transaction
PRIMARY KEY	row	statement	transaction
FOREIGN KEY	statement	statement	transaction
EXCLUDE	statement	statement	transaction

12.1.1 Isolation Levels

A database provides different isolation levels:



The isolation level is a property of each transaction that can be specified.

A read is dirty if it was written by a different uncommitted transaction (and therefore might be a value that never exists in the database, e.g. if the other transaction aborts).

In the Read Uncommitted isolation level, a transaction may perform dirty reads.

In the Read Committed isolation level, a transaction may not perform dirty reads (but this doesn't guarantee serializability because the values between reads may still change when another transaction commits).

In the Repeatable Read isolation level, a transaction may not perform dirty reads and an item read multiple times cannot change values (but this doesn't guarantee serializability because different values might be updated, causing the schedule to still not be serializable). Furthermore, phantom tuples (newly inserted tuples between two operations in a DB) are possible, which can cause the schedule to not be serializable as well.

Transactions can be set read only which helps the system to optimize performance.

Because the notion of "serializability" (which concerns only about the result; i.e. if it is equal to the result acquired by some serial schedule) is hard for the database to handle, Conflict Serializability is introduced.

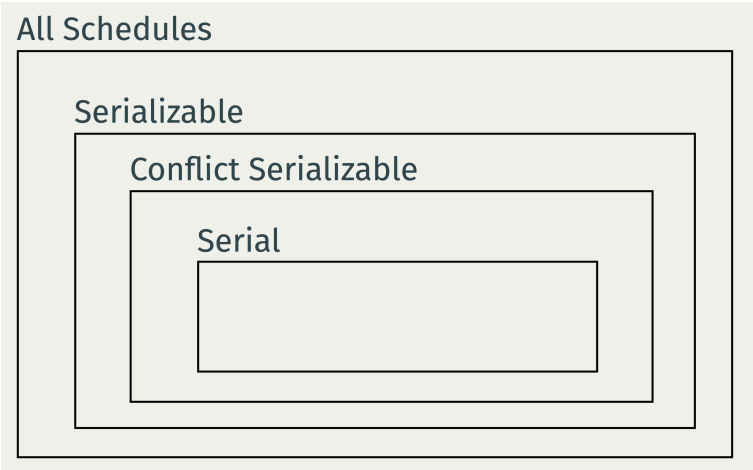
A conflict of operations occurs for same transactions for all adjacent operations, i.e. if we have … O1 … O2 …, then O1 and O2 are in conflict. For different transactions, O1 and O2 are in

conflict if one of them is a write to the same location. A schedule is serializable if it can be translated into a serial schedule with a sequence of nonconflicting swaps of adjacent actions.

There are Read-Write, Write-Read and Write-Write conflicts. Two schedules are conflict equivalent iff they involve the same actions of the same transactions and every pair of conflicting actions is ordered in the same way. Conflict serializable can now also be defined as: A schedule S is conflict serializable if it is conflict equivalent to some serial schedule. An operation can also be in conflict to an ABORT if a reorder leads to different values for the operation (e.g. a read).

For checking whether a schedule is conflict-serializable, one can use the definition with the swaps. It's also possible to construct a dependency graph like this: Each transaction is a node in the graph. There is an edge from Ti to Tj if an operation Oi in Ti is in conflict with an operation Oj in Tj and Oi appears earlier than Oj. With this definition, a schedule is conflict serializable iff its dependency graph is acyclic (has no loop).

The following hierarchy exists between the schedules:



12.1.2 Enforcing serializability

There is an optimistic (non-serializable schedule will be very rare event) or a pessimistic (non-serializable schedule will happen all the time) approach for ensuring serializable schedules. The optimistic approach uses snapshot isolation, the pessimistic locking.

12.1.2.1 Locking

The idea is that before accessing a data object X, it is locked so other transactions cannot access it at the same time. It is only released when it is safe to do so. The question is when to acquire locks and when to release locks.

There are different types of locks, S Lock / Shared Lock (Read Lock) which is required for read operations and X Lock / Exclusive Lock which is required for write operations. Depending on the current lock, the request is granted or not:

Request \ Current Lock	No Lock	S	X
S	Grant	Grant	Wait
X	Grant	Wait	Wait

As a locking strategy, two phase locking (2PL) is used. In the first phase (growing), each transaction requests the locks that it needs from the DBMS's lock manager and it cannot release locks. In the second phase (shrinking), the transaction is only allowed to release locks that it

previously acquired. It cannot acquire new locks. 2PL guarantees conflict serializability. A problem of 2PL can be cascading aborts where an abort causes an abort of another transaction, e.g.:

```
1   -- T1 --
2   BEGIN
3   XLock(A)
4   READ(A)
5   Write(A)
6
7   UNLOCK(A)
8
9
10
11  ABORT
```

```
1   -- T2 --
2   BEGIN
3   XLock(A)
4
5
6
7
8   Read(A)
9   Write(A)
10
11
```

If T2 has already committed, it gets even worse as the DBMS would need to rollback T2 (which is against ACID, namely no D).

Strict 2 PL solves this issue. All locks are kept until the end of the transaction (commit or abort).

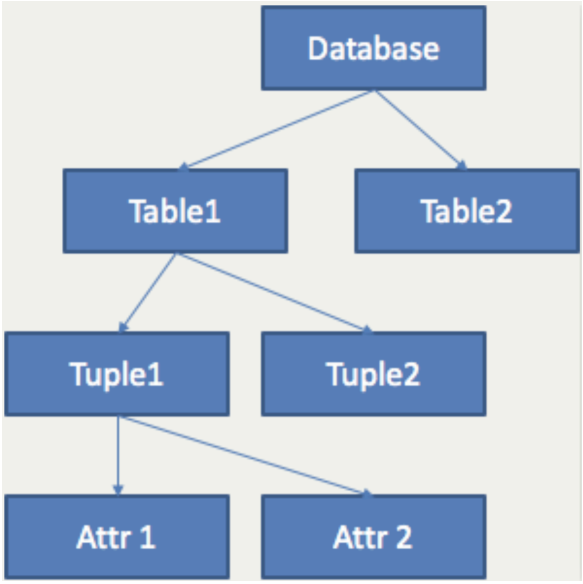
A problem of 2 PL and strict 2 PL are deadlocks, e.g.:

```
1   -- T1 --
2   BEGIN
3   XLock(A)
4
5
6
7   READ(A)
8   Write(A)
9   XLock(B)
10
11
12
13
```

```
1   -- T2 --
2
3
4   BEGIN
5   XLock(B)
6   Read(B)
7
8
9
10  XLock(A)
11
12
13
```

To detect deadlocks, the wait-for graph is constructed. Each node is a transaction and there is an endge from Ti to Tj if Ti is waiting for a lock currently hold by Tj. The system has a deadlock if the wait-for graph is not acyclic. The DBMS checks for deadlocks periodically and kills transactions until the graph is acyclic. There are many criterions on how to decide which transactions to kill (e.g. age, progress, etc…).

The database has a hierarchical lock structure and locks are supported (for correctness and performance) at multiple granularity.



To implement this, three new types of locks are introduced. Intention shared (IS) which means that some lower nodes are in shared, Intention exclusive (IX) which means that some lower nodes are in Exclusive and Shared and Intention Exclusive (SIX) which means that the root is locked in shared and some lower nodes are in exclusive.

When the system wants to lock a tuple1 in S, it starts from the root and acquires IS on the database, IS on Table1 and S on Tuple1. The compatibility matrix looks like this:

Current Lock

Request

Mode	NL	IS	IX	S	SIX	X
NL	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	Yes	No	No	No
S	Yes	Yes	No	Yes	No	No
SIX	Yes	Yes	No	No	No	No
X	Yes	No	No	No	No	No

Different isolation levels can be implemented using variations of Strict 2PL:

	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted			
Read Commited	N		
Repeatable Read	N	N	
Serializable	N	N	N

- Serializable -- Strict 2PL + Index Lock (Intuitively, lock not only existing objects, but also all other relevant objects that one could insert in the future)
- Repeatable Read -- Strict 2PL
- Read Committed -- S locks are released immediately
- Read Uncommitted -- No shared lock (S)

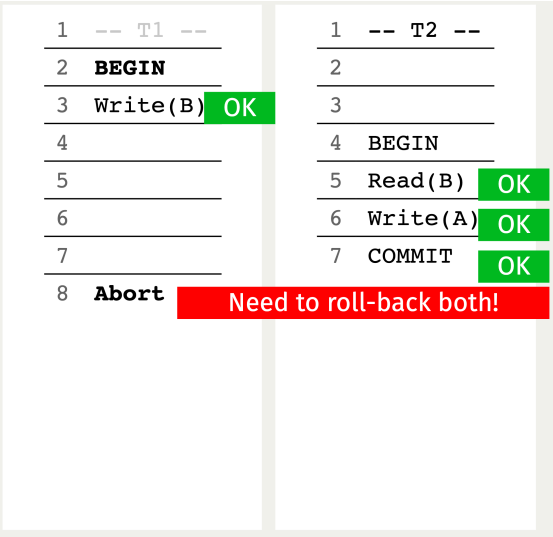
12.1.2.2 Concurrency Control with Timestamps

The intuition is to pre-define a serial order and roll back all transactions that break this order. Each transaction is assigned a timestamp TS(Ti). If TS(Ti) < TS(Tj), then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where Ti appears before Tj. Each database object is also associated with a timestamp. RT(X) is the read time of X i.e. the highest timestamp of a transaction that has read X. WT(X) is the write time of X i.e. the highest timestamp of a transaction that has written X.

Each transaction reads / writes objects without locks. If Ti tries to access an object "from the future" (an object X with a higher read / write timestamp than TS(Ti), abort and restart. The details are:

- Imagine T_i wants to read X with write time $TS(T_i) < WT(X)$
  - This cannot happen -- why?
  - Abort T_i, restart.
- Imagine T_i wants to read X with write time $TS(T_i) > WT(X)$
  - Allow T_i to read X
  - Update read time $RT(X) = \max(RT(X), TS(T_i))$
- Imagine T_i wants to write X with write time $TS(T_i) < WT(X)$
  - This cannot happen -- why?
  - Abort T_i, restart.
  - Else: Allow T_i to write to X, update $WT(X)$.
- Imagine T_i wants to write X with read time $TS(T_i) < RT(X)$
  - This cannot happen -- why?
  - Abort T_i, restart.
  - Else: Allow T_i to write to X, update $WT(X)$.

The protocol always generates a schedule that is conflict serializable and there are no deadlocks. But long transactions could "starve to death" (keep getting cancelled). Furthermore, the problem of cascading aborts can happen as well (but can be solved by marking writes as tentative and acting based on that):



12.1.2.3 Snapshot Isolation

The idea behind snapshot isolation is that the DBMS maintains multiple physical versions of a single logical object in the database. When a transaction writes to an object, the DBMS creates a new version of that object and when a transaction reads an object, it reads the newest version that existed when the transaction started.

On commit, the DBMS checks for conflicts. T1 is aborted if a T2 exists such that T2 committed after TS(T1) and before T1 commits, and T1 and T2 updated the same object.

Advantages of snapshot isolation are that writers and readers do not block each other and read-only transactions have always access to a consistent DB snapshot without locks. Because snapshot is at the table level, phantom tuples can't occur.

But there's a certain overhead and a problem is write skew: Checking integrity constraint happens in the snapshot. Two concurrent transactions (that update different objects) might be ok, but the combination not.

12.1.3 Recovery

A DBMS has to recover from aborts of a single transaction, system crash with a loss of main memory but an intact disk or a system crash with a loss of disk (where a backup has to be read from external sources).

Given a schedule, we say T1 reads from T2 if T1 reads a value written by T2 at a time when T2 was not aborted. Now, the different notions of recoverability can be defined:

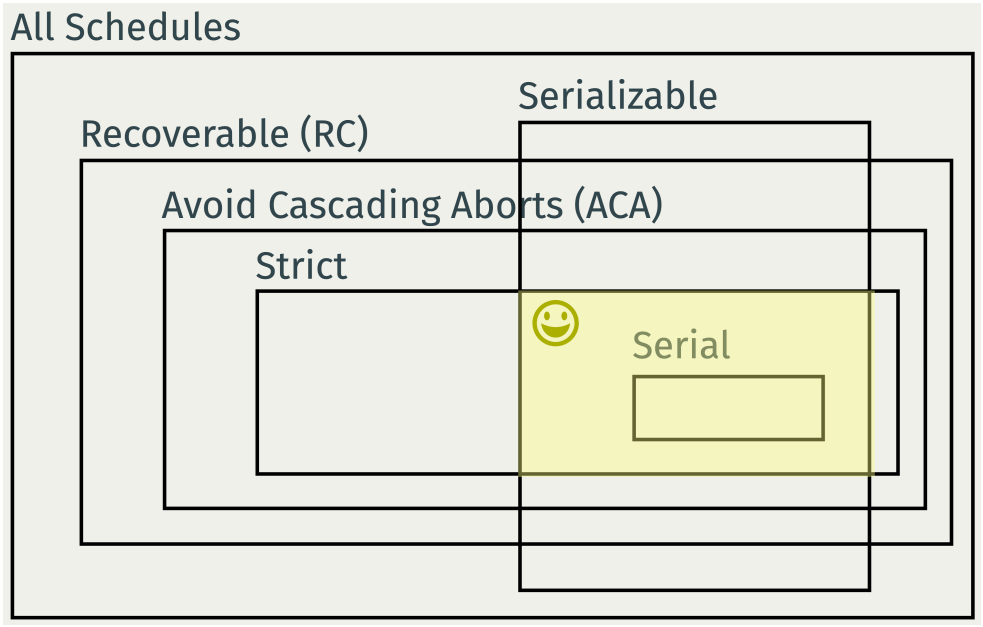
- Recoverable (RC) schedule:
  - If T_i reads from T_j and commits, then $c_j < c_i$
  - c_i is the commit time of T_i
- Avoids Cascading Aborts (ACA) schedule:
  - If T_i reads X from T_j, then $c_j < r_i[X]$
  - $r_i[X]$ is the time T_i reads X
- Strict (ST) schedule:
  - If T_i reads from or overwrites a value written by T_j, then $c_j < r_i[X]/w_i[X]$ or $a_j < r_i[X]/w_i[X]$
  - a_j is the abort time of T_j

The idea behind recoverable schedules is that there is no need to undo a committed transaction.

The idea behind ACA schedules is that aborting a transaction does not cause aborting others.

The idea behind strict schedules is that undoing a transaction does not undo the changes of other transactions.

We have the following relation of recoverability & serializability:



Strict 2PL ensures both serializability and strict recoverability.

12.1.3.1 Recovering from Failure

The basic assumption is that the disk (secondary storage) is safe. The log is a file opened for appending only. Like other DB files, it is first in memory buffer and then flushed to disk. To the log file, START T, COMMIT T, ABORT T and UPDATE <T, X, v> can be appended (but they don't actually cause this action, these are just logs) and FLUSH flushes the log to disk.

In undo logging, when T modifies a database element X, <T, X, old value> is written to disk before the change X is written to disk. If a transaction commits, the COMMIT record must be written to disk only after all other changes are written to disk. The recovery process then scans the log file and looks for all START T without COMMIT T (uncommitted transactions). The actions of these transactions are reverted using the log file and ABORT T is written at the end of log.

In redo logging, when T modifies a database element X, <T, X, new value> is written to disk. Before any modification on disk, the logs are flushed and COMMIT is written to disk. The recovery process then scans the log file and if <COMMIT T> is not in the log then no changes of T appears on disk. If <COMMIT T> is in the log, it doesn't mean that all changes are already on disk. The changes have to be "redone" using the log. The log can become very long because having COMMIT in the log doesn't mean all its changes are in the database. There are potential solutions like using checkpoints.

Here is the difference between undo logging and redo logging illustrated in an example:

Undo Logging

1		
2	READ(A, t)	
3	t := t * 2	
4	WRITE(A, t)	<T, A, 8>
5	READ(B, t)	
6	t := t * 2	
7	WRITE(B, t)	<T, B, 8>
8	FLUSH LOG	
9	OUTPUT(A)	
10	OUTPUT(B)	
11		<COMMIT T>
12	FLUSH LOG	

Redo Logging

1		<START T>
2	READ(A, t)	
3	t := t * 2	
4	WRITE(A, t)	<T, A, 16>
5	READ(B, t)	
6	t := t * 2	
7	WRITE(B, t)	<T, B, 16>
8		<COMMIT T>
9	FLUSH LOG	
10	OUTPUT(A)	
11	OUTPUT(B)	
12		

Undo logging causes potentially more I/O (data written before COMMIT flushed to log) whereas redo logging causes potentially larger buffers.

A problem of redo logging is that when two transactions make changes to the same page (but different objects in the page), the one that first commits has to wait until the second is done, otherwise it would flush uncommitted data to disk (which isn't allowed). A solution to this problem is Undo/Redo logging. Before modifying any database element X on disk, <T, X, old value, new value> is written to disk and the disk is flushed before the modifications are written to disc. Object X can be flushed before or after the <COMMIT T>. An example looks like this:

1		<START T>
2	READ(A, t)	
3	t := t * 2	
4	WRITE(A, t)	<T, A, 8, 16>
5	READ(B, t)	
6	t := t * 2	
7	WRITE(B, t)	<T, B, 8, 16>
8	FLUSH LOG	
9	OUTPUT(A)	
10		<COMMIT T>
11	OUTPUT(B)	

The recovery is then pretty simple. If <COMMIT T> is not in the log, it is an incomplete transaction that needs to be undone. If it is in the log, the transaction needs to be redone.

13 Distributed Database

There are different ways that a distributed database can be constructed: Shared Memory, Shared Disk or Shared Nothing. The shared nothing architecture is described here. In this architecture, each worker is a single-machine DB that uses all the techniques described so far. The master receives queries and coordinates (optimization, planning, concurrency control, logging and recovery).

The basic idea is to partition the database to each worker and each worker only deals with its own partition. The naïve solution is that each worker holds one table. Another solution is to use horizontal partitioning. A tables' tuples are split into disjoint subset by putting all tuples that

share values on some column(s) on the same machine. There are multiple ways to do this like range partition or hash partition.

There are multiple ways to do distributed query optimization and different issues that arise. For instance, when one wants to join two tables, some possibilities are:

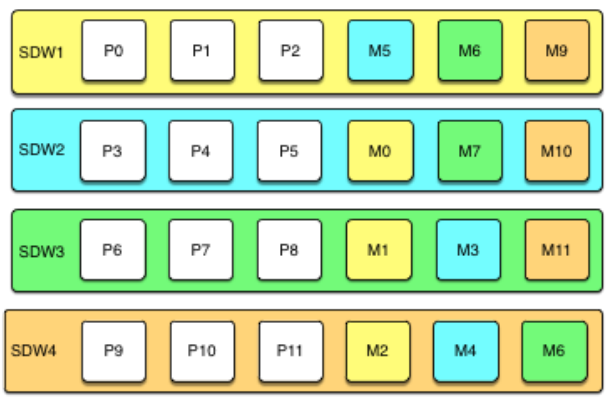
1. One table is replicated at every node and each node joins its local data and then sends their results to a coordinating node.
2. Tables are partitioned on the join attribute and each node performs the join on local data and then sends to the same node.
3. Both tables are partitioned on different keys. If one of the tables is small, it is broadcasted to all nodes.
4. Both tables are not partitioned on the join key. Tables are copied by reshuffling them across nodes.

As described in the relational algebra part, semi-join reduction can be important in distributed databases.

13.1 Data Replication

Group Mirroring is a replication at machine level where the contents of one machine are also stored on another:

A machine failure will lead to 2x more load on another machine. In contrast, when using spread mirroring, the additional load is evenly distributed:

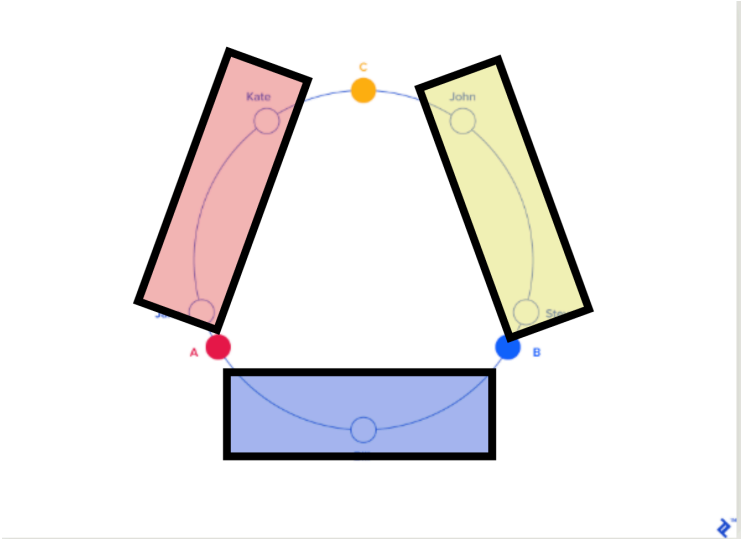


14 Key Value Store

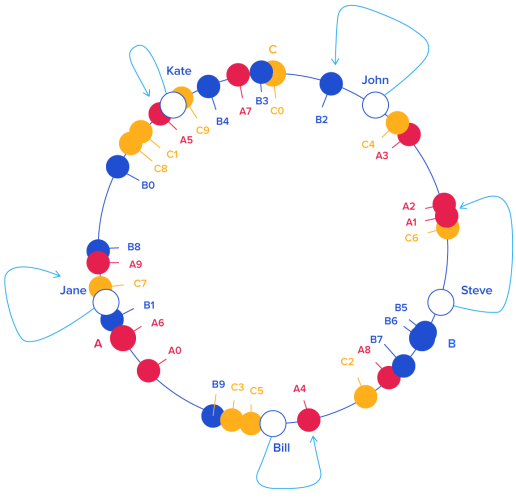
A key value store gets rid of schemas (just a key and blob attached to it), SQL (get and set as the only operations) and ACID (eventual consistency, potentially no transactions). Horizontal partitioning with replication of partitions is usually used for these stores.

Key value stores provide very fast lookups, are easy to scale and useful in many applications. But there is no easy support for queries beyond point queries like ranges, joins or query on attributes (because there are no attributes). Often times, the data is inconsistent. The complexity and responsibility is pushed to the application and some operations are very costly to implement.

A problem when using a simple hash-partition method is that in case of a failure of a machine, other data potentially needs to be moved as well (because their Hash mod n-1 changes). An alternative is to hash both machine and data and each machine deals with the data points in the clockwise direction, before the next machine.



To improve load balancing, virtual nodes are introduced where the load is better distributed (with a small number of machines):



For replication, data can be stored on the next r-1 nodes in clockwise direction. To guarantee consistency among replicas, the number of machines contacted in read operations (R) + the number of machines that have to be blocked in write operations (W) have to be bigger than N, i.e. $R + W > N$.