Database management and database systems

Kristoffer Klokker

2022

Contents

1 Database introduction

Databases are a collection of data stored in a DBMS (database management system) which serves the purpose of:

- Create database and specifying their schemas (logical structure of the data)
- Query the data (questions about data or retrieving the data)
- Store large amount of data in long periods with easy access and modificatio of the data
- Durable and should be able to recover data in case of error or misuse
- Allow multiple user access at once

Today the norm in database systems are relation databasese which present the data as tables, and the underlying datastructure is not needed for use of the system.
In the case of multiple different database and systems which should be syncronised either a data warehouse is used where a periodically copy of the smaller databases is made. Another approach is a middleware which is a translation between two databases schemes.
A database has mainly two users, admin which can modify the schema using DDL command (data-definition language) which modify the schema by altering the metadata.
The other user being a normal user allowed to do DML command (data-manipulation language).
When a DML command is executed two subsystems are handling the command:

1.1 Query compiler

The compiler takes the query and creates a query plan (a sequence of actions) and passes it the the execution engine.
A request of data sends data in tuples to the buffer manager, which is responsible for all data transaction between disk storage and memory
The compiler consist of

- Query parser - which builds a tree from the textual query
- Query preprocessor - Sematic check of query to ensure a valid query and transforms the query into algebraic operators

- Query optimizer - Transofrm the query to the best avaliable sequence of operation on the actual data based on metadata and schema structure

1.2 Transaction manager

The transaction manager is used to log for possible recovering and ensuring durablity
Also the transaction has a concurrency-controle manager to ensure a bundle of transaction is executed as they were one unit and locking data when used to ensure no data is wrongly overwritten.
The transaction also manages such that every execution is isolated in case of revertion.
The transaction followed the ACID test, where

- A - atomicity which ensures that in case of error a transaction is never half completed
- C - consistency in data and data constraints
- I - isolation ofe ach operation done in order in a transaction
- D - durability of data such it is never lost after a transaction

2 The models of data

A data model is used for describing data and conisist of:

- Structure of data - Referred to as physical data model, but is simply a high level data structure
- Operations on the data - A limited set of operations in DBS at hight level, which makes it more flexible for underlying improvements
- Constraints of data - Constraints on data to ensure data integrity

2.1 The semistructured-data model

The data is setup in a relation more like a tree rather than table.
Here XML is mostly used to represent data by nested tags.

```
01 | <Movies>
02 |    <Movie title="Gone with the wind">
03 |       <Year>1939</Year>
04 |       <Length>231</Length>
```

```
05 |       <Genre>drama</Genre>
06 |    </Movie>
07 |    <Movie title="Star Wars">
08 |       <Year>1977</Year>
09 |       <Length>124</Length>
10 |       <Genre>sciFi</Genre>
11 |    </Movie>
12 | </Movies>
13 |
```

2.2 Document Type Definition model

This is model based on XML to standify a more text based model of a database. This model focus on all the data types in a database and how they are nested. An example is as follows:

```
01 | <!DOCTYPE Stars [
02 |    <!ELEMENT Stars (Star*)>
03 |    <!ELEMENT Star (Name, Address+, Movie*)>
04 |    <!ELEMENT Name (#PCDATA)>
05 |    <!ELEMENT ADDRESS (Street, City)>
06 |    <!ELEMENT Street (#PCDATA)>
07 |    <!ELEMENT CITY (#PCDATA)>
08 |    <!ELEMENT Movie EMPTY>
09 |       <!ATLIST Title CDATA #REQUIRED, Genre CDATA #IMPLIED>
10 | ]>
11 |
```

Here $\#PCDATA$ is some data for an element and it can here be seen how nesting is allowed by defining star with address wich is also defined later. Here the database Stars also include any number of star as stated at first. The last movie is an alternative writing were instead of $< Movie > Hello < /Movie >$ the $EMPTY$ makes it be $< MovieTitle = "Hello","Comedy" >$, $CDATA$ is simply characterData and $\#REQUIRED$ means it must be filled whereas $\#IMPLIED$ is optional.

2.3 The basics of relational database

Relation refers to the two dimensional table of data.With attributes being the coloumns and rows being a tuple. The tuple is then made of an relations where a relation with attributes are a schema.
A relation is defined by $Name(attribute : type, attribute2 : type)$ and a tuple is in the same order and valeis for the given attributes.
Relations comes in sets and not lists and therefore order is not important
A database may contain a key which is attribute(s) which define a unique

relation, if no combination of attributes are unique a ID for the relation can be created.

2.3.1 Keys

A PRIMARY KEY is used for securing no dublicates and only allows non null values in the key attribute.
UNIQUE allows null as a value in its attribute, but dublicates is still nto allowed.
When creating a table the key can be choosen by after an attribute after its type $PRIMARY\ KEY$ or $UNIQUE$ is inserted or at the end of the table definition $PRIMARY\ KEY\ (a)$ can be inserted where a are the attributes. Again Unique can also be used like this.

3 SQL language

SQL is the language used to create queries. SQL has tree kinds of relations, stored called tables (relations), views (relation which are not stored but used for computation), temporary tables (tables constructed by SQL temporary)
The data types avaliable by SQL are:

- $CHAR(n)$ - Character string of fixed length n
- BIT - Logical value with possible values being TRUE, FALSE, UNKNOWN
- INT - Number can also be $SHORTINT$ for small number
- $FLOAT$ - Higher precision numbers here $DOUBLE$ can also be used for more precision
- $DECIMAL(n, d)$ - Numbers of length n and the decinam placed at d
- $DATE$ and $TIME$ - both essentially being strings with a strict format

The basic commands for modifying tables are:

- DROP TABLE R; which removes the table R with all its entries
- ALTER TABLE R ADD a type; Adds attribute a as a *type* to table R
- ALTER TALBE R DROP a; Removes the attribute a form table R

3.1 Default

SQL also has $DEFAULT$ which can be added after any attribute after type and describes the default value if non is given.
When creating a realtion, the default attribute can be used in case of a not given value.
CREATE TABLE CoolPeople(name CHAR(30) DEFAULT 'Kristoffer Klokker', phone CHAR(16));

4 Algebraic Query Language

The algebraic query langauge is the operation behind the SQL real language. This is not a programming language, but the simplicity makes in easier to optimize and faster.
All the operations works on boths sets and bags(the allowance of mutliple accourences of tuples)
The following table is in precedence order from first at top and last on bottom.

1. Same attributes (and same type) in same order
2. In case of attributes with same name they will be "renamed" to 'relationName.Attribute'
3. The standard aggreagation operations are: SUM,AVG,MIN,MAX,COUNT
4. Attribute A1 can also be 'A1 $rightarrow$ k' for renaming it to, or some expression like 'A1 + A2 $\rightarrow$ sum'

It can here be seen that there is multiple combinations which are equal. Here the highest priority readability due to the query compiler rewriting it anyway.
When writing linear notation be used as such:

$$R(a, y, l) := \sigma_{age<18}(People)$$

From which R now can be used as a variable.
Often when combining operationsa tree is used where root is the final product and branches are the first operations done.

Name	Symbol	Effect	*
Selection	$\sigma_C(R)$	Select tuples from R which meets the condition C	
Projection	$\pi_{A1,A2}(R)$	All tuples from R but only attributes $A1$ and $A2$	4
Rename	$\rho_{S(A1,A2)}(R)$	All tuples in R but rename attributes to A1 and A2 Alternative $R2_{A1,A2} := R$	
Duplicate Elem.	$\delta(R)$	Eliminate dublicate tubles in relation R	
Aggregation	Ex. $SUM(B)$	Computes operation on given attribute B	3
Group	$\gamma_{B,SUM(C)\to s}R$	Group relation R by attribute B and in this example creates attribute s with the sum of attribute C to each goup	
Sort	$\tau_{A1,A2}(R)$	Sort relation R according to A1 and A2	
Cartesian	$R \times S$	Every possible combination of tuples from R and S	2
Nat. join	$R \bowtie S$	Cartesian product but only where overlapping attributes are equal	2
Theta join	$R \bowtie_C S$	Every cartesian product of R and S which meet the C condition	2
Outer join	$R \rtimes\ltimes S$	Nat. join but none joined tubles, are padded with NULL/$\perp$. Also in variations of $\rtimes$ where only all tubles from R is kept, and $\ltimes$ which keep all from S	
Difference	$R - S$	Tuples which in R which is not in S	1
Union	$R \cup S$	All tuples from R and S	1
Intersect	$R \cap S$	Tuples which are in both R and S	1

4.1 Examples

A

A	B
1	5
2	4

B

B	C
5	8
4	9

C

A	B
1	5
7	0

$A \cup C$

A	B
1	5
2	4
1	5
7	0

$A \cap C$

A	B
1	5

$A - C$

A	B
2	4

$\sigma_{B<4}(C)$

A	B
7	0

$\pi_B(A)$

B
5
4

$A \times B$

A	A.B	B.B	C
1	5	5	8
1	5	4	9
2	4	5	8
2	4	4	9

$A \bowtie B$

A	B	C
1	5	8
2	4	9

$A \bowtie_{A.B=B.B} C$

A	A.B	B.B	C
1	5	5	8
2	4	4	9

$\rho_{G,K}(A)$

G	K
1	5
2	4

5 Queries in SQL

5.1 Select

The most simple query is the SELECT FROM WHERE.
This a simple select query which selects attributes from a database where a conditions is met.
The conditions can use the 6 comparisions operators: $=, <>, <, >, <=, >=$ where $<>$ is not equal.
Operations may also be done on values for int $+, -, *, /, \%$ and for string $||$ which adds two string together.
When comparing string the $<$ operators are defined upon lexicongraphic order.
For string the LIKE can be used to match a pattern. A pattern uses % and _ where % is any number of wildcard charachters and _ is a single charachter.
Strings are defined by ' therefore when working with them inside a string two ' is used like ''
Logic operators avaliable is AND, OR and NOT
Ex. SELECT title, genre FROM Books WHERE author = Kristoffer Klokker

AND year = 2022

5.2 Modifying the database

5.2.1 Insert

INSERT INTO R VALUES (A1,A2..)
Insert a tuble into relation R with values A1, A2 as the attributes in same order.
Alternative INSERT INTO R(Sum,Name) VALUES (A1,A2..), which will insert the value to the corresponding attrubute.
A1 and A2 can also be a tubles by itself from another DB, for inserting multiple tubles at once.

5.2.2 Deletion

DELETE FROM R WHERE <condition>
Delete every tuple in relation R which satifies the condition.
Remember deletion is not a linear process rather all satifiable tuples will be deleted once all are found.

5.3 Update

UPDATE R SET name = 'Kristofer' WHERE <condition>
Changes attribute value, where the condition is met.

5.4 Rename

In case of a rename of attributes the keywords AS is used, here it is also possible to modify values.
Ex. SELECT title, ' and the stone' AS name from Books
Constants are also avaliable if wanted.
Ex. SELECT title || ' and the stone' AS name, 'forever' AS readTime from Books

5.5 Date and time

Dates are defined as: DATE 'year-month-day' Ex: DATE '2022-12-04'
Times are defined as: TIME 'hour:minut:second' Ex: TIME '19:42:12.55'
In case of timezones TIME 'hour:minut:second-hour:minut' this will be the

amount of hours and minuts behind GMT. The minus could also be a plus.
Timestamps are used to combine date and time.
Timestamps are defined as 'TIMESTAMP 'year-month-day hour:minut:second'
These values can of course be compared both with equal and all the < as expected.

5.6 Working with null

A sql entry may be null due to: not knowing the value, no value really match the attribute or the value is withheld.
This can have consequence in the query condition.
When using any math operations on null it becomes null.
When comparing anything to null it becomes UNKNOWN.
To check if a value is null the IS NULL is used.
In the opearation 'UNKNOWN AND x' the statement is false when x is false and UNKNOWN when x is true.
In the operation 'UNKNOWN OR x' the statement is true when x is true and UNKNOWN when x us false.
When negating UNKNOWN the value is UNKNOWN.
UNKNOWN values will not interpretated as true and queries where the condition result in UNKNOWN will not show up.

5.7 Ordering

To order the output of a qeruy the ORDER BY <attributes> is used.
After the attributes list DESC can be added for a descending list whereas ASC is not needed due to ascending being the default.
Multiple attributes are used in case of a the first attribute being equal then the second attribute is used for ordering.

5.8 Multiple tables

When working with multiple tables the FROM can be filled up with multiple tables.
Ex. SELECT title, age FROM Books, CoolPeople WHERE author = name
In case of the table having the same name or the same table being used an alias can be created by a space after the table followed by the wanted name.
Ex. SELECT Book1.name, Book2.name FROM Books Book1, Books Book2 WHERE Book1.title = Book2.title AND Book1.name <> Book2.name

5.9 Union, intersect and except

The relational algebra operation can be used on two SELECT statement to get the expected output.
Ex. SELECT name FROM Books INTERSECT SELECT name FROM CoolPeople
For union and except intersect is replaced with UNION or EXCEPT.
These operations will eliminate dublicates to prevent this the ALL keyword can be used. Ex. SELECT name FROM Books INTERSECT ALL SELECT name FROM CoolPeople

5.10 Join

The simplest join is the CROSS join also known as cartesian product Ex. Books CROSS JOIN CoolPeople
Theta joing the cross product but with a condition is done like cross but with ON and then condition Ex. Books JOIN CoolPeople ON author = name
Natural join where attributes are matched is done by taking the two tables with NATURAL JOIN between Ex. CoolPeople NATRUAL JOIN SmartPeople
Outer join is like natural join except instead of throwing away non matching entries they are filled with null Ex. CoolPople NATURAL FULL OUTER JOIN SmartPeople.
This will include all entries from SmartPeople and CoolPeople but if an entry only exists in one the attributes from the other table is fileld with null.
Variant of LEFT and RIGHT also where left only includes all entries from the left table and only matching from the right Ex. CoolPeople NATURAL LEFT OUTER JOIN SmartPeople.

5.11 Subqueries

Subqueries are used for getting a value from a table which then is most often used in a compare statement or as a table the selection is taken from.
If a subquery only return one value it is called scalar and can be used in an condition.
Ex SELECT title FROM Books Where author = (SELECT name FROM CoolPeople WHERE key = 3)
In case a subquery return multiple result the following operators can be used:

- EXISTS R - return true if R is not empty

- s IN R - return true if s is an entry in R
- s (>) ALL R - returns true if the math operation here > is true for all entried in R
- s (>) ANY R - returns true if the math operation here > is true for any entry in R

All the operators can ofcourse be negated by puttin NOT infront.

5.12 Sets

To get a set in a query the DISTINCT is used after the SELECT Ex. SELECT DISTINCT names FROM...

5.13 Aggregation operations

This is operations which is done on the selection.
These operations include: SUM, AVG, MIN, MAX and COUNT.
Ex. SELECT AVG(age) FROM CoolPeople
In case of wanting only sets DISTINCT is used as follow SELECT SUM(DISTINCT age) FROM CoolPeople.
GROUP BY is often here used to get the aggregation to a given group. Ex. SELECT name, SUM(age) FROM CoolPeople GROUP BY name.
This will give the age sum for each name in cool people.
NULL is ignored when calcilation an aggregation but a group can be in NULL.
THE GROUP BY can also works as a condition with the keyword HAVING.
Ex. SELECT name, SUM(age) FROM CoolPeople GROUP BY name HAVING MIN(age) > 18
This will return the names and the sum of ages with the same name but only people over 18 count.

6 Design theory for Relational Databases

6.1 Functional dependencies

A functional dependecy is defined as:

$$A_1, A_2, ..., A_n \rightarrow B_1, B_2, ..., B_n$$

Functional dependecies are constraints on a relation which states:
If two tuples agree on the A attributes of the dependency they must agree on the B attributes
The transitive rule states for the FDs $A \to B$ and $B \to C$ will $A \to C$ be true.
The splitting combining rule states for the FD $A \to BC$, it can be spliited into $A \to B$ and $A \to C$, or combined the other way around.
The trivial FD for the attribute A is $A \to A$.
A basis of FD's are just the given FD's
The minimal basis of FD's are only singletons, and removing any FD will not be equivalent to the basis.

6.1.1 Keys

A key is a minimal set of attributes which will be unique for every set of attributes.

$$\{A_1, A_2, ..., A_n\}$$

Here the attributes are part of the key.
Superkey: A set of attributes of which one is a key.
Another terminoligy which is often used is: A candidate key is a possible minimal key and a superset of that key is a non minmal key.

6.1.2 Closures

A closure for the attribute A, is all attributes, which can be computed from which $\{A\}^+ \to \{A, B, C\}$, from the singletons $A \to B$ and $B \to C$
The closure is found by starting with the trivial, then FD's which include the derived attributes is inserted until no more informations from FD's can be added.
The closure of $\{superKey\}^+$ will result in all attributes.
The colsure of $\{key\}^+$ will also result in all attributes, but no attribute of the key can be removed.
Projecting FD's onto a new relation with a limited amount attributes, will only the FD's which involves the new relation hold.

7 Designing a Database

7.1 Anomalies

Anomailies are repeated information in multiple tuplees.
The can lead to lost of data, if repeated data turns out to be the last data when deleted or non updated values for all repeated data.
To prevent this decomposing relations can be done
This is where all the repeated data is in one relation, and the none repated attribute is in another relation with key also.

7.2 Normalizations

Normalizations are design constraints upon a database schema. The constraints come in different layers with the higher layer the more costraints which must be met likewise ealier layer constraints.

7.2.1 1NF.

The first nomal form states:

- Eliminate repeating groups in individual tables.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key.

In other words, an attribute in a table should be atomic.
For example an employee table with employee data, should not include an attributes or multiple attributes of monthly payments, rather a table with employee and a payment should be made.

7.2.2 2NF.

The second normal form states: A non-prime attribute (an attribute which does not belong to any possible key) can not depend on part of a possible key.
That means in a function dependency $A \to B$ if A is part of a candidate key, B must also be part of a candidate key.
In more nonformal terms:

- Create separate tables for sets of values that apply to multiple records.

- Relate these tables with a foreign key.

An example of this is if a value in one table has a group which it belongs. For example a table which include yearly earning which is based on another attribute the amount of years worked.
This table should not include the yearly earning but rather a seperate table should exists with years worked and yearly earning, where the year is a foreign key connecting the tables.

7.2.3 3NF.

The third normal form builds upon 2NF, but adds that an attribute not part of the key can not relate to an attribute which neither is part of a key.
This means that all attributes in a table should relate to the key.
Here functional dependencies come into play, due to it saying that in a table all functional depencies should be from the key to the attributes.
In case of an attribute A depending on an attribute B a new table should be made such A can be a key and B can be removed from the first table and putted into the new table.

7.3 Boyce-Codd Normal Form

BCNF is a constraint upon 3NF due to a loophole in the definition. In a table with possible keys such every attribute is part of a key will be in 3NF. For example the table with Products(ReleaseYear, Name, Popularity, ReleaseYearAndMonth).
This table has the possible keys:

- $\{Name\}$
- $\{RelearYear, Ranking\}$
- $\{ReleaseYearAndMonth, Ranking\}$

This table is in 3NF but anomalies can result if year is updated but not year and month.

BCNF therefore states: A functional dependency in a table must depend on a possible key (not minimal), with exception of the trivial dependency.
Therefore the example is not BCNF due to $ReleaseYearAndMonth \rightarrow ReleaseYear$ and ReleaseYearAndMonth is not a key by itself.

It can here be noted that, transitive relations may be used.

Example the relation $R(A, B, C, D)$ with the FD's $A \rightarrow B$, $B \rightarrow C$ and $B \rightarrow D$.
It can here be seen A is the key due to being the only attribute from which all other attributes can be found.
Therefore the relation can be described in two relations $R(A, B)$,$R(B, C, D)$.
Here BCNF will hold.

7.3.1 Decomposing a relation into BCNF

First check if the relation R is in BCNF if so return R.

Let the violation be $X \rightarrow Y$, then compute X^+ and choose $R_1 = X^+$, and R_2 have X and all attributes not in X^+.

Compute funcitonal dependencies in the two relations by:
For all attributes in the relation X calculate X^+ and add all functional dependencies of which $X \rightarrow A$ and A is in X^+ and in the given relation.
Then remove all redundant dependencies, if $\{Y, Z\} \rightarrow B$ and two FD's is $P \rightarrow Y$ and $P \rightarrow B$ then, the only needed FD is $P \rightarrow B$.
Using this method find the functional dependencies for R_1 and R_2 and go through this algorithm again with each.

7.4 Valid decompose

A measure if the decomposition is correct, is if when the natural join is performed on the given decomposed relations, we shall get the original relation back with the same number of tuples and correct tuples.
The case method, consist of writing a tablau, with a row representing a decomposed relation. Then the row is filled with know values and unknown valeus are subscribted with the row number.
Then from the FD's the rows should be able to be combined by finding equal variables from which the subscribted variables are removed, ending up with a row with no subscribted values.

7.5 Constraints

7.5.1 Key

Keys are a way of a constraint which limits for an attribute no tubles can not have the same keys.
done by when creating the attributes in a schema fx.

```
workerId INT PRIMARY KEY
```

Or if multiple then at end of schema the following can be done:

```
PRIMARY KEY (workerID, name)
```

Foreign keys are a way of referencing a key from another table. This constraints it such a tuble with the foreign key must exist before in the foreign table before a tuble can be created.
If this constraint is unwanted the

```
DEFERALE INITIALLY DEFERRED
```

can be used such a tuple can be created even though the key does not exists. This is done by at attribute level:

```
FOREIGN KEY(worker) REFERENCES workers.workerId
```

When updating the foreign key, which a tuple depend upon the defualt behavior is to reject the changes.
It can also be cascaded to update the depended value, or null the depended tuple.
It can also be set upon each action DELETE and Update by

```
FOREIGN KEY(worker) REFRERENCES workers.workerId ON
   DELETE SET NULL ON UPDATE CASCADE
```

This will when deleted set the depended tuble value to null and when updated the depending tuple is updated with the update.

7.5.2 Checks

The check constrain is used when creating a schema, and defines a way to only allow tuples if the value upholds a constraint.
Example can be

```
price INT CHECK (price <= 100)
```

this constraints the price to be equal or above 100.
A check can also be at the end and check multiple attributes

```
CHECK (price <= 100 OR name = 'cheapProduct')
```

7.5.3 Assertion

Assertions are global rules in the database, which are checked upon any changes.
Example

```
01 | CREATE ASSERTION onlyValidWorkers CHECK (
02 |    NOT EXTSTS (
03 |       SELECT worker FROM workers WHERE workerId < 0
04 |       )
05 | );
```

This checks if any workerId is negative.
Assertion is not a thing in PostSQL where tiggers are used instead.

7.5.4 Event condition action rules / tirggers

```
01 | CREATE TRIGGER workTrig
02 |    AFTER INSERT ON Workers
03 |    REFERENCING NEW ROW AS NewTuple
04 |    FOR EACH ROW
05 |    WHEN (NewTuple.workerId NOT IN (SELECT workerId FROM
        Coworkers)
06 |    INSERT INTO Coworkers(workerId, name) VALUES (
       NewTuple.workerId, newTuple.name);
```

As it can be seen the event is at line 1, when the tirgger is checked.
Then line 3 and 4 reference the new inserted rows and for each of them checks the condition of line 5.
The condition checks if tubles values exists in another table and if true the action on line 6 is triggered.
The event can use the keywordst DELETE, INSERT or UPDATE ... ON attribute
The foreach can be absent to work on the whole statements even if multiple inserts or such.
In case of more wanted actions the action part can start with BEGIN and END at the end.
The reference part on an update can have both row by

```
01 | REFERENCING
02 |    OLD ROW AS ooo
03 |    NEW ROW AS nnn
04 |
```

but in PostSQL referencing is not used and the row is just OLD and new is NEW on update.
Likewise PostSQL only allow functions in the trigger, where a function is defined as

```
CREATE FUNCTION checkWorkTime() RETURNS TRIGGER AS
    $$BEGIN
    IF NEW.time < OLD.time THEN
    UPDATE workers SET fishy = true WHERE workerId = NEW
    .workerId;
    END IF;
    RETURN NEW;
END$$
LANGUAGE plpgsql;
```

This funciton can then be triggered in postSQL with:

```
CREATE TRIGGER fishyWorker
    AFTER UPDATE ON Workers
    FOR EACH ROW
    EXECUTE PROCEDURE checkWorkTime();
```

A trigger may not fix the situation but can use

```
IF True
THEN RAISE EXCEPTION 'Worker is fishy!'
END;
```

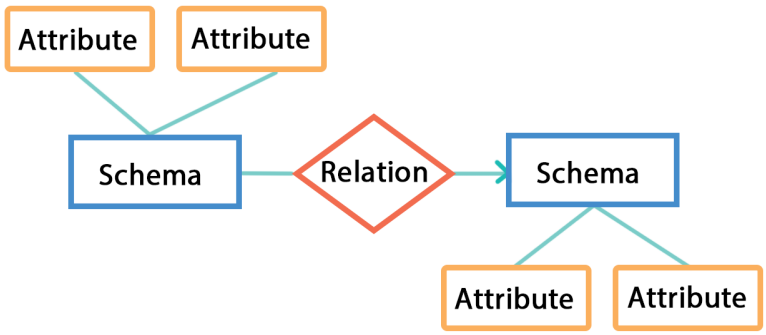
8 High-level DB models

The most used model is the ER diagram, which describes schemas and their relation to other schemas.
The ER model consist of:

- Rectangles - The sets / schemas
- Ovals - The attributes
- Diamonds - The relation between two schemas

A tuple in the model is called a relationship.
A relation may have mannnny sets attached.

Figure 1: ER model, with a schema, attributes and a relation.
A many-one relation, so right schema can only have 1 left shema.



Edges may also have labels, to indicate if two schemas are related in more ways.
A relation may also have attributes, in cases where it makes the most sense. This can also be replaced by a relation more if, many attributes may accour. Some models like UML does not allow more than a binary relation. To combat this a connecting schema, from which relations can go out to each schema.

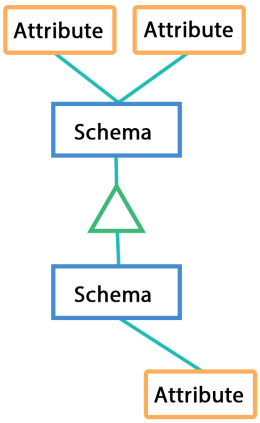
8.1 Multiplicity of ER models

ER diagrams can also show restrictions, this include the number of relations a schema can have.
The standard is many-many, this means a relation may involve as many as it wants from both sides.
Then there is many-one which dictates that, a relation may only be related to one relation. This is indicated by an arrow which points towards the set if which only one can exist.
There are also one-one which dictates only one relation can relate to another relation. This is indicated by an arrow at both ends of an edge.
A rounded arrow means exactly one

8.2 Subclasses in ER models

A subclass use a isa object to show the subclass relation.
The object is in form of a triangle, where the parent is connected at the top and the subclass at the bottom.
The subclass will then inhert all attributes from the parent and can have

Figure 2: ER model with the button schema being a subclass of top schema.



new attributes or relations.
A isa object is always one-one, but the arrows are not needed.

8.3 Designing a model

To create a good model, the model should be simple and not include redundant infomration.
Be of course as faithfull to what it tries to model.
In some cases, a relation may not be the answer. If the the set E:

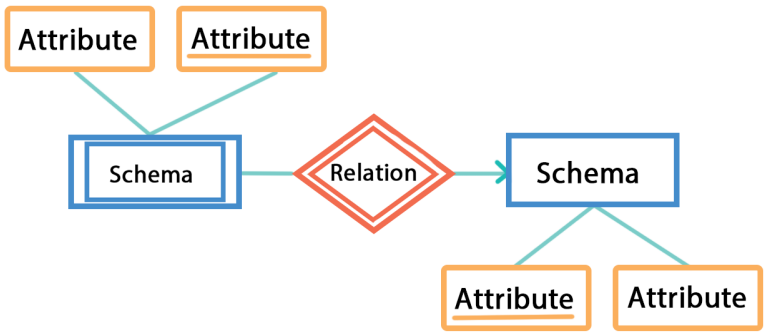
- Has only arrows entering it.
- The only key in E is all attributes.
- No relationship involves E more than once

If all these cases are met, the relation and the set should be removed, and the attributes of E and the relation should be in the related schema.

8.4 Keys in ER models

In ER models keys work like in DBMS.
Here the constraints are that a set must have a key, otherwise it is not a set, which a relation can form.
There may be more than one key, but a primary key must be picked.
With isa's the enitity must have a key which is inherted from the parent.
The key is shown by underlining the key attribute(s).

Figure 3: ER model with a weak key, where the left schema has two keys its own attribute and the attribute from the right schema



A weak key is a key from another set.
Weak keys can be valid, in a many-one binary relation which must exist.
In this scenario a key from set F can be used as a weak key for set E.
This is illustrated by the a double rectangle on the set with the weak link and a double border on the relations diamond which provides the weak key.

8.5 Constraints

A model may also include constraints in the relations.
By using rounded arrows, not only is it many-one relation, it states that at least one set must exists in the relation.
In the case of a restriction on the many-many relation, number critera (Ex, < 10) is written at the intersect of edge and set and states the minimum.

8.6 Converting from ER diagram to DB model

To convert it is pretty straight up. The given sets and relations are converted into schemas, with the given attributes.
It may be needed to combine some schemas, this is common if a schema mostly just contains keys, and a many-one is in place.
In this case a big schema can be made, where the relations attributes and the schema with mostly key attributes are added.
When working with weak keys, the key attribute simply is the weak schema, the relation schema and of course the original schema.

8.6.1 isa objects

To convert isa object there are multiple ways.
The most strightforward way is creating everything as schemas.
Then the keys will connect each schema to the other to obtain all information.
The object oriented approach, makes every possible schema combination according to the isa object.
The null approach creates a schema with all attributes, from every set in the isa. Then null values are made when the object is parent etc.
The null approach makes queries the most simple and fast, but uses more unused space unlike the object orented approach which has only one tuble for every entry with no nulls, but many schemas

9 Storing and indexing tuples

9.1 Storing

A tuble is stored as a record in the following format:

- File header containing: pointer to schema, record length, timestamp of creation.
- Attributes of tuble

Each part stored as a multiple of 4 bytes
Each record are gathered in a block with a header file in the following format:

- Link to one or more blocks for index purpose
- Info about relation which the records belong to
- A ”directory” with info about offset of each record in the block
- Timestamp for last modification and/or access

The blocks are then pointed to in the database address space, as either a physical address (location on drive) or a logical address (A map for a drive). Most often a combination is used, due to the phiscal address being faster but the logical address being able to moved around and being overall more flexible.

9.1.1 Pointer swizzling

Pointer swizzling is a technique for optimizing pointers.
When a pointer is secondary memory it points to the secondary address of an object. The swizzling then is used when the record is moved into main memory along with the object the pointer is swizzled into pointing at the main memory address.
So instead of having to translate upon every action on the record with the pointer, the pointer is already translated. This only require an extra bit in the pointer indicating if the pointer is swizzled or not.
There are two types of swizzling automatic which upon moving the block to main memory every pointer is tried to be swizzled and demand which tries a swizzle once the pointer is used for the first time.
Once the record is returned to secodnary memory the pointers must be unswizzled.
This may lead to objects being pinned in main memory until pointers to it has been unswizzled, this is handled through a linked list of pointers such a chain reaction of unswizzling can be done.

9.1.2 Non fixed length of attributes

There are two cases of non-fixed length attributes: non-fixed length variables and repeatable objects.
For repeatable objects the method is to in the end have an open field from which pointers can be putted, and a header with the amount of pointers. The problem is the non fixed size and therefore a large capazity is reserved. Another alternative is an fixed length of pointers and a pointer to another place with additional pointers if needed. The makes searches faster but reading all pointers is slower due to more I/O calls.
For non-fixed length variables, they can be used at the end of the record, such the varying size does not affect the fixed sized attributes.
In case of more flexxible schema such as with XML, each attribute can be used as: a byte for codename, a byte for codename of type, a byte for length of the variable and the variable itself.

9.1.3 Variables of size larger than a block

These types of variables span multiple records and each records is called a record fragment.
The record fragment needs additional header information containing:

- A bit representing if the records is a fragment
- A bit if the fragment is the first or the last fragment for its record
- A pointer for the previus fragment or further fragment if existing

An object which must span over mutliple records are BLOBs, which a re binary large objects.
A BLOB being stored on multiple records are an advantage for like a movie the appropriate parts of the BLOB may be sent.

9.1.4 Inserting a record

To insert a record, empty space in a block may be found or new block may be created.
In case of a order such as a key descending the record must come in at appropriate point.
There are different methods for this:
An offset table which tells which records is in correct order. In case of a block not having space for the new record, the either largest or smallest appropriate to sorting is inserted into the next block.
An overfllow block which stores items which should belong to the current block and in case of that overflowing a new one is created.

9.1.5 Deletion

The deletion of a record, may lead to more space.
In case of of no order, the space is moved such the empty space is moved into the block buffer space.
To prevent pointers to the deleted object pointing wrongly gravestones are used.
These are either placed in the logical record map, or at the start of the physical position.

9.2 Index

9.2.1 Indexing 101

An index is a datastucture to find tubles, using constant attributes.
The attributes used for index is called index key.
To search for a tuble using index a balanced binary tree is created by the DBMS.
To create an index the following is done

```
01 |        CREATE INDEX keyIndex ON table(KeyAttribute1,
       KeyAttribute2)
02 |
```

An index can be on as many attributes as wanted, and the order matters due to the first attribute in a multi index can also be used as a single index.
An index should be created in the case of many lookups but not many inserts, deletions and updates due to it being slowed down.
In most cases the most usefull index is the key due to it returning 1 tuble or nothing and speedsup a join.
Some DBMS includes tools which can suggest index options based on the query history.
The options are found through a greedy algorithms which simply test all possible outcomes, by reusing the history in the query optimizer to estimate the time save.
This is done over all options until non result in time save.

9.2.2 Types of indexes

A dense index is an index used with the attributes in order and pointers to the whole record.
This is faster due to the fewer data blocks, the abillity to binary search due to the order and the size being small enough to often keep in main memory.
A sparse block is even smaller only having a pointer and value to each block.
This requires the datablocks to be sorted.
Multiple level index, is index of the index file.
Secondary index are index on non primary key and has therefore to be dense.
A usecase of the secondary index is the heap structure, the placement is not in order and therefore the secondary index is used.
A secondary index may waste space due to many records having the same value. The can be solved by buckets.
Here a indirect secondary indexare used
Very similary to the multilevel index, the index points towards a list of pointers to the records. Then each pointer between to index is for the same value.
This can also improve search time when buckets exist for multiple search conditions. Then when eliminating buckets for each condition only the records left with pointers are answers.
This methods saves I/O time due to only looking up the index and the final results.

9.2.3 B-Trees

B-Trees are balanced trees used for indexing. The balanced tree has a variable n which determines the amount of children minus 1.
Therefore the structure of the tree is:
A node has $n + 1$ pointers to blocks and $n - 1$ keys. The relation between key and pointer is p_1 is lower than k_1 and p_2 is higher than k_1 but lower than k_2 etc...
A leaf will have $n-1$ keys and $n-1$ pointers to records, each corresponding to eachother. The n'th pointer is used for the next leaf in the tree in sequence.

The trees root may only have 2 pointers and at every node and leaf at least half the pointers must be in use.
The leaf pointer to next leaf is used using a range search. Here the lower range is found and each record of the leaf is returned until the upper leaf is achieved.
When inserting into the tree and the node is already filled up, a sibling node is created. The keys are then splitted but there will always be one left which is inserted into the parent.
When removing, there are three cases:

- The removal can be done without problems
- The removal leads to under half pointers are used, but a sibling leaf can transfer a pointer
- The removal leads to under half pointers are used an the sibling can not transfer due its own limit, then the siblings are merged, and same deletion procedure is done at parent.

The tree wiil lead to very effective lookups, the typical tree has $n = 255$, and by only 3 layers up to 16.6 million pointers can be stored. With root in main memory it will therefore only take 2 I/O lookups to find the pointer.

9.2.4 Hashtables

A hashtable for index is made of blocks the records can be stored in. In case of overflow a overflow bucket is created.
An ideal hash table will only take one I/O lookup but if many records are stored in a bucket the list has to be looked through.
An Extensible hash table, holds only pointers. When the buckets are overflowing, the bucket is split into two and the number of considered bits in the buckets from the hash are incrementet.

When illustrated the box to the left of the datablock is the number of considered bits in the hash value.
The hashing method, insures small buckets in the most minimal space, but it requires a lot of buckets for large data and the doubling takes a lot of computation.
Linear hashing is an alternative, with overflow buckets. The bucket key is $\lceil \log_2 n \rceil$ (where n is the number of bucket) bits of the hash value from the right side.
If the keys binary value is larger than the number of bucket the far left bit is set to 0.
To determine if a new bucket is needed is if the current number of records r exceeds n times a ration like 1.7
When adding a new bucket in case of the number of buckets excced the possible bucket with the current number of bits used from the hash value i, then every key get a zero infront.

10 NoSQL

NoSQL are DBMS system, which differs from the RDBMS, in less constraints, not able to join and less structures data.
But it is faster and stores larger amouint of data, due to its large scaleability.
The structure of NoSQL does require schemas and therefore more flexible.
NoSQL comes in different data model:

- Graph - nodes contain data and edges connect data
- Key-Value - A mapping between string key and primitive value
- Coloumn family - Key and collection of values in coloumns
- Document - JSON like model