DIS*Notes

Magnus Goltermann (xzb187)

May 2023

Indhold

*nuts???

1 E/R Diagram (Conceptual modeling)

An E/R diagram (entity/relationship) visually represents relationships between entities. It also models attributes and more complex relationships. However, not everything can be expressed in an E/R diagram and must be written/expressed in another way. E/R is subjective, as there are often many ways to correctly model a scenario. We however always try to reduce redundancy.

1.1 Entity

Represented as a rectangle. An entity is often a noun, which means it's often an object, set of objects, or similar. E.g. employees, fruits, etc. One entity has to have at least one key and can have attributes to describe the entity.

1.2 Attribute

Represented as a circle. An attribute is something that describes an entity or relation. It always has a domain, e.g. an Int or a string. An attribute can only have one value, if more is needed we need to create a separate entity to handle those.

1.3 Key (simple)

Represented as an underline of the attribute. Is a unique attribute, like ID or CPR-number. Can also combine two or multiple attributes to form a unique key pair(s).

1.4 Relationships

Represented as a rhombus (rectangle stretched and tilted). Association/relationship between two or more entities. E.g. entity1 hates entity2, then the relationship is "hates". A relationship can also have attributes.

1.5 Arrows

Arrows can be used to define uniqueness constraints. There must never be an arrow/line directly between two entities (must be a relationship between).

1.5.1 Many-Many

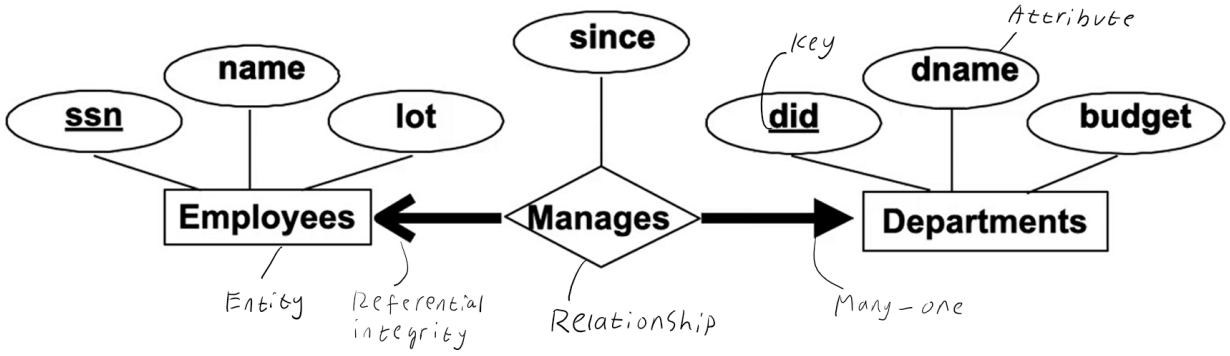
Regular line, no arrow tip (A-B). Indicate that A can have many B and B can have many A. E.g. one student can have many teachers, and one teacher can have many teachers.

1.5.2 Many-one

Filled arrowhead points toward the entity that has many, coming from the entity that has one. E.g. A<-B, then A can have many B, but B can only have at most one A.

1.5.3 Referential integrity

Pointy arrowhead (A<-B). Entity A has exactly one B, and therefore must have a B. Sometimes when making an Many-one like this, it can be usefull when making the tables in e.g. SQL to just incorporate the relationship in the table of the entity that has "one", since it can only have one relationship with an entity anyways.



Figur 1: Example of E/R diagram. A department has exactly one manager, but an employee may manage at most one department (or none).

1.6 Weak entities

Seen by bold or double lined around entity. A weak entity is like a "child"of another entity, an can only exist when it has a parent. It therefore must exist in a one-to-many relationship: One entity can have many weak entities, but one weak entity has only one "owner"entity. Since it only has one owner, the weak entity can be identified uniquely by its owner (e.g. by the owner's key). It is often used with a relationship, which is then also denoted by bold or double-lined. Whenever the owner entity is deleted the deletion has to cascade and also delete the child entities (the weak entities)

1.7 ISA ("Is a"hierarchy)

Denoted by a triangle. Like a class in object-oriented programming. ISA denoted the inheritance. E.g. A is a class, and B inherits from A (visually the hierarchy is denoted by height, the superclass has to be above).

Then B has all the same attributes as A, and can then also add their own. Smart when e.g. dealing with users, since a program/site may have many different users such as admins, employees, customers, etc. but they all need at least some base attributes.

2 Relational data model

A relational database is made of a schema and instances and is "just"a set of relations (denoted as tuples). The schema specifies the relations and columns of that relation (and type of the column). E.g. Student(sid:string, name:string, login:string, age:integer). The instance is the table that contains the data with rows and columns after the specified schema. The numbers of rows is called the cardinality. The number of columns/fields is called the degree/arity. In the relational data model all rows are distinct.

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Figur 2: Example of instance of a student relation. Cardinality=3 and degree=5

2.1 E/R to relational

An entity or relationship is each made of their own schema. E.g. the Employee entity from figure 1 is Employee(ssn:string, name:string, lot:int), and an instance can be made by Employee("141203", "John", 26). When doing relationships in relational it is also made of its own schema/table. E.g. we if A hates B, we then create a table Hates(A, B). The table does not implicitly tell us who hates who, but since we defined it as "left"hates "right", we have to keep track of it ourselves. The primary key of a relationship table is just the combined key of the entities that have the relation, in our example, it would be PRIMARY KEY (A.key, B.key), these keys are refered to as foreign keys.

2.2 Integrity constraints (ICs)

A condition that must be true for any instance of the database. The domain constraint (the type of the column must be the same, e.g. int or string), key constraint, and foreign key constraint. A legal instance of a relation is one that satisfies all integrity constraints. The database should not allow illegal instances and should enforce these rules .

2.2.1 Primary key constraints

A set of columns (fields), is a superkey for a relation, if not 2 tuples can have the same unique set. And if a set of fields is a superkey, we can call it a key. A subset of a key must not also be a superkey, since we want the smallest amount of fields for a key. If there are multiple keys, we just choose one (maybe at random, does not matter). In the "worst case"all columns can form a super key since no two rows are identical.

2.2.2 Foreign key

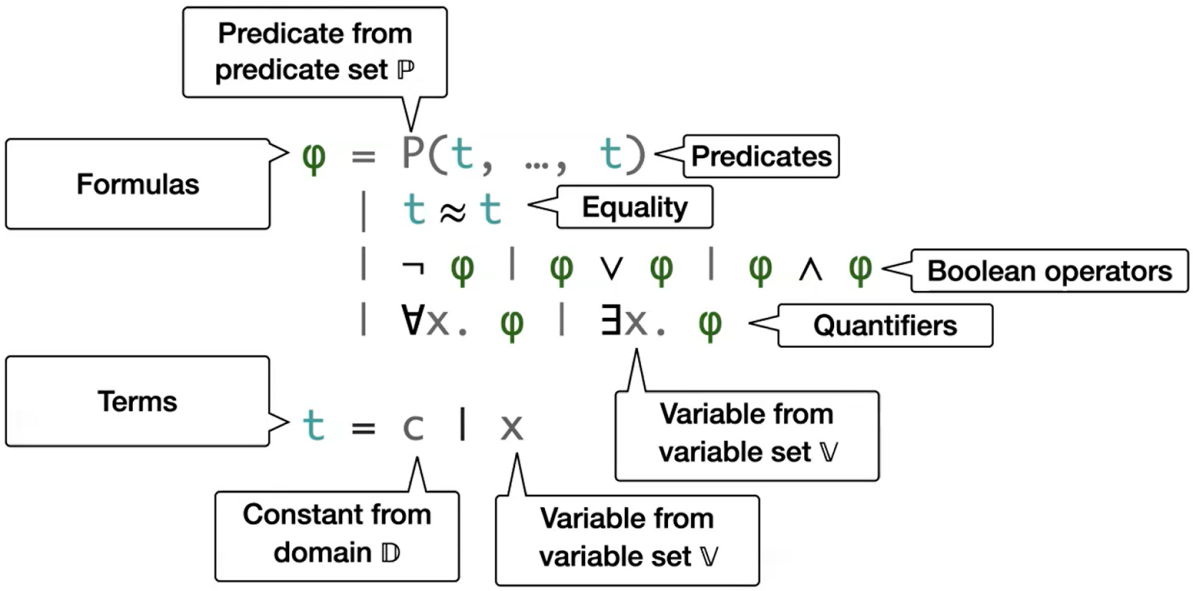
A foreign key is used when e.g. a relation table has to use a key from another entity as the primary key (or in combination with). Dangling references are not allowed, e.g. we cannot refer to a foreign key if the table it comes from does not exist.

2.2.3 Enforcing referential integrity

As dangling reference are not allowed, we cant just delete a table that is being used as e.g. a foreign key. We therefore first has to delete the dependent tables, and then delete the "main"table. We can also delist all references by setting the reference to NULL.

3 Relational Calculus (RC)

Describes the result of a query, and not how to compute it. Whenever we use quantifiers we "get rid"of the variable that we quantify over. The result of a query is the columns that are still free variables.



Figur 3: Relational calculus "syntax". Note that the predicate set is just a table.

$$
\begin{aligned}
v \vDash P(t1, \ldots, tn) &\iff (v(t1), \ldots, v(tn)) \in DB(P) \\
v \vDash t1 \approx t2 &\iff v(t1) = v(t2) \\
v \vDash \neg\, \varphi &\iff v \nvDash \varphi \\
v \vDash \varphi \vee \psi &\iff v \vDash \varphi \text{ or } v \vDash \psi \\
v \vDash \varphi \wedge \psi &\iff v \vDash \varphi \text{ and } v \vDash \psi \\
v \vDash \forall x.\ \varphi &\iff v(x \mapsto c) \vDash \varphi \text{ for all } c \in \mathbb{D} \\
v \vDash \exists x.\ \varphi &\iff v(x \mapsto c) \vDash \varphi \text{ for some } c \in \mathbb{D}
\end{aligned}
$$

Figur 4: Semantics of RC. v is a row that we compare, such that it matches with the formula. $\vDash$ means satisfies. So we want evaluations/rows that satisfies the right side.

$$
\begin{aligned}
fv(P(t1, \ldots, tn)) &= fv(t1) \cup \ldots \cup fv(tn) \\
fv(t1 \approx t2) &= fv(t1) \cup fv(t2) \\
fv(\neg\ \varphi) &= fv(\varphi) \\
fv(\varphi \vee \psi) &= fv(\varphi) \cup fv(\psi) \\
fv(\varphi \wedge \psi) &= fv(\varphi) \cup fv(\psi) \\
fv(\forall x.\ \varphi) &= fv(\varphi) - \{x\} \\
fv(\exists x.\ \varphi) &= fv(\varphi) - \{x\}
\end{aligned}
$$

Figur 5: How free variables are computed from RC. See that just $\exists$ and $\forall$ removes variables, others just combine or do nothing.

All examples when going over all the different quantifiers and such use the examples given below in figure 6.

Employees

ssn	name	lot
0983763423	John	10
9384392483	Jane	10
3743923483	Jill	20

Figur 6: Caption

3.1 Predicates

Assignment of the free variables, such that the result must contain the assignment.

$$
v \vDash P(t1, \ldots, tn) \iff (v(t1), \ldots, v(tn)) \in DB(P)
$$

Figur 7: Predicate

$$
\left\{ \begin{array}{l} ssn \mapsto 3743923483 \\ name \mapsto Jill \end{array} \right\} \vDash Employees(ssn, name, 20)
$$

$$
\left\{ \begin{array}{l} ssn \mapsto 0983763423 \\ name \mapsto John \end{array} \right\} \nvDash Employees(ssn, name, 20)
$$

Figur 8: Predicate example using table in 6. See that the first one has lot set to 20, and therefore will only match rows that has lot to 20.

3.2 Equality

Equality is just setting conditions for a variable to be equal to something. This does not like predicate remove it as a free variable.

$$\nu \models t1 \approx t2 \iff \nu(t1) = \nu(t2)$$

Figur 9: Equality

$$\{\ lot \mapsto 10\ \} \models lot \approx 10$$

$$\{\ lot \mapsto 10\ \} \not\models lot \approx 20$$

$$\{\ lot \mapsto 10,\ lot' \mapsto 20\ \} \not\models lot \approx lot'$$

Figur 10: Equality example, using 6.

3.3 Negation

Gives results that do not satisfy the formula. E.g. we can almost say "does not equal".

$$\nu \models \neg\ \varphi \iff \nu \not\models \varphi$$

Figur 11: Negation

$$\left\{\begin{matrix} ssn \mapsto 0983763423 \\ name \mapsto John \\ lot \mapsto 10 \end{matrix}\right\} \not\models \neg\ Employees(ssn,name,lot)$$

$$\{\ lot \mapsto 10\ \} \models \neg\ lot \approx 20$$

$$\{\ lot \mapsto 20\ \} \not\models \neg\ lot \approx 20$$

Figur 12: Negation example, using 6. See that the first example does not satisfy, since the row is in the table, but we want everything not in the table the formula states. Second example simply gives everything where lot is not 20.

3.4 Disjunction

The "or"operator. Combines two formulas, where just one of them has to be satisfied. Be carefull, since it can often lead to infinite result, see section 3.9.

$$\nu \models \varphi \vee \psi \iff \nu \models \varphi \text{ or } \nu \models \psi$$

Figur 13: Disjunction

$$\left\{\begin{array}{l}\texttt{ssn} \mapsto \texttt{0983763423}\\ \texttt{name} \mapsto \texttt{John}\\ \texttt{lot} \mapsto \texttt{10}\end{array}\right\} \models \texttt{Employees(ssn,name,lot)} \vee \texttt{lot} \approx \texttt{20}$$

$$\left\{\begin{array}{l}\texttt{ssn} \mapsto \texttt{0983763423}\\ \texttt{name} \mapsto \texttt{John}\\ \texttt{lot} \mapsto \texttt{20}\end{array}\right\} \models \texttt{Employees(ssn,name,lot)} \vee \texttt{lot} \approx \texttt{20}$$

$$\left\{\begin{array}{l}\texttt{ssn} \mapsto \texttt{0909090909}\\ \texttt{name} \mapsto \texttt{Dmitriy}\\ \texttt{lot} \mapsto \texttt{20}\end{array}\right\} \models \texttt{Employees(ssn,name,lot)} \vee \texttt{lot} \approx \texttt{20}$$

Figur 14: Disjunction example, using 6. See that it often can lead to weird results, as in example 3, where Dmitriy satisfies, but is not in the original table.

3.5 Conjunction

The "and"operator. Combines two formulas, where both of them has to be satisfied for the row to be included.

$$\vee \models \varphi \wedge \psi \qquad \Longleftrightarrow \vee \models \varphi \text{ and } \vee \models \psi$$

Figur 15: Conjunction

3.6 Implication

The implication is often said as p derives q. Can sometimes be easier to just look at the truth table, see figure 16 below.

$$\varphi \rightarrow \psi := (\neg\varphi) \vee \psi$$

(a) Implication

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

(b) Truth table for implication

Figur 16

$$\left\{\begin{array}{l}\texttt{ssn} \mapsto \texttt{0983763423}\\ \texttt{name} \mapsto \texttt{John}\\ \texttt{lot} \mapsto \texttt{10}\end{array}\right\} \models \texttt{Employees(ssn,name,lot)} \rightarrow \texttt{lot} \approx \texttt{10}$$

$$\left\{\begin{array}{l}\texttt{ssn} \mapsto \texttt{0983763423}\\ \texttt{name} \mapsto \texttt{John}\\ \texttt{lot} \mapsto \texttt{20}\end{array}\right\} \models \texttt{Employees(ssn,name,lot)} \rightarrow \texttt{lot} \approx \texttt{10}$$

$$\left\{\begin{array}{l}\texttt{ssn} \mapsto \texttt{0909090909}\\ \texttt{name} \mapsto \texttt{Dmitriy}\\ \texttt{lot} \mapsto \texttt{20}\end{array}\right\} \not\models (\neg\ \texttt{Employees(ssn,name,lot)}) \rightarrow \texttt{lot} \approx \texttt{30}$$

Figur 17: Implication example, using 6.

3.7 Existential Quantifier

In words we often say "For some x there exists ...". In RC it is just used to remove a column/free variable. Since we just want a row, but we don't necessarily care about all/some variables.

$$\nu \vDash \exists x.\ \varphi \iff \nu(x \mapsto c) \vDash \varphi \text{ for some } c \in \mathbb{D}$$

Figur 18: Existential quantifier

$$\left\{\begin{array}{l} \text{ssn} \mapsto 0983763423 \\ \text{name} \mapsto \text{John} \\ \text{lot} \mapsto 10 \end{array}\right\} \vDash \text{Employees(ssn,name,lot)}$$

$$\left\{\begin{array}{l} \text{ssn} \mapsto 0983763423 \\ \text{name} \mapsto \text{John} \end{array}\right\} \vDash \exists \text{lot. Employees(ssn,name,lot)}$$

$$\{\ \text{name} \mapsto \text{John}\ \} \vDash \exists \text{ssn. } \exists \text{lot. Employees(ssn,name,lot)}$$

$$\{\ \} \vDash \exists \text{name. } \exists \text{ssn. } \exists \text{lot. Employees(ssn,name,lot)}$$

$$\{\ \} \vDash \exists \text{lot. lot} \approx 20$$

Figur 19: Existential quantifier example, using 6.

3.8 Universal Quantifier

Check whatever value we plug in is true. Read as "For all". Has to check all values in the domain. Often used in combination with the implication.

$$\nu \vDash \forall x.\ \varphi \iff \nu(x \mapsto c) \vDash \varphi \text{ for all } c \in \mathbb{D}$$

Figur 20: Universal quantifier

$$\left\{\begin{array}{l} \text{ssn} \mapsto 0983763423 \\ \text{name} \mapsto \text{John} \\ \text{lot} \mapsto 20 \end{array}\right\} \nvDash \text{Employees(ssn,name,lot)}$$

$$\left\{\begin{array}{l} \text{ssn} \mapsto 0983763423 \\ \text{name} \mapsto \text{John} \end{array}\right\} \nvDash \forall \text{lot. Employees(ssn,name,lot)}$$

$$\{\ \text{name} \mapsto \text{John}\ \} \nvDash \forall \text{ssn. } \forall \text{lot. Employees(ssn,name,lot)}$$

$$\{\ \} \nvDash \forall \text{name. } \forall \text{ssn. } \forall \text{lot. Employees(ssn,name,lot)}$$

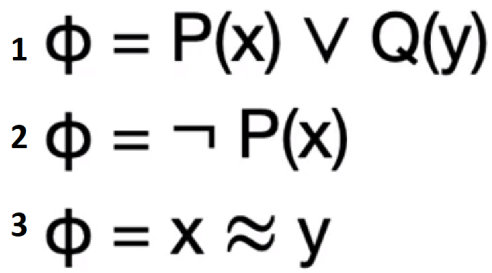
$$\{\ \} \nvDash \forall \text{lot. lot} \approx 20$$

$$\{\ \} \vDash \forall \text{lot. lot} \approx \text{lot}$$

Figur 21: Universal quantifier examples, using 6.

3.9 Infinite results

One large problem with relational calculus is that you can often create queries that give infinite relations/results. Some examples are given below in figure 22. Why each example is infinite is given below also:
1: Gives infinite because of the $\vee$ ("or"), since as long as e.g. P(10) is True, then Q(v) can be anything, and even rows not contained in Q, and therefore infinite.
2: Gives infinite because negating a table just gives everything not in the table, thus that is of course everything else, which is then infinite.
3: y and x can just be arbitrary numbers, hence 1=1, 2=2, 3=3 and so forth, giving infinite results.



Figur 22: 3 examples of infinite queries.

3.10 Domain independence

Given all tables (predicates) are finite, where does the "infinitely"of the results come from? It simply comes from the domain. E.g. if we have a column that contains integers, then the domain of the column is all integers, and not only the ones that are contained in the table. The same goes for strings or other types.
An "easy"way to understand if my thinking of the domain as a parameter of all queries, hence it is always there in the "background"lurking. A query is then **domain independent** if the query gives the same result no matter what domain is given as "input". For E.g. the conjunction "and"operator is domain-independent, as the query will only contain results that are either in the first or second table and not in some arbitrary domain. This is called safe, since they give a finite result.
Any formula without free variables has a finite query result, and is thus safe.

4 Relational Algebra

Describes the computation of the query as a sequence of table transformations. The result of one transformation can be used as input of another transformation.
All examples will use the tables given below in figure 23.

R1

sid	bid	day
22	101	10.10
58	103	11.12

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

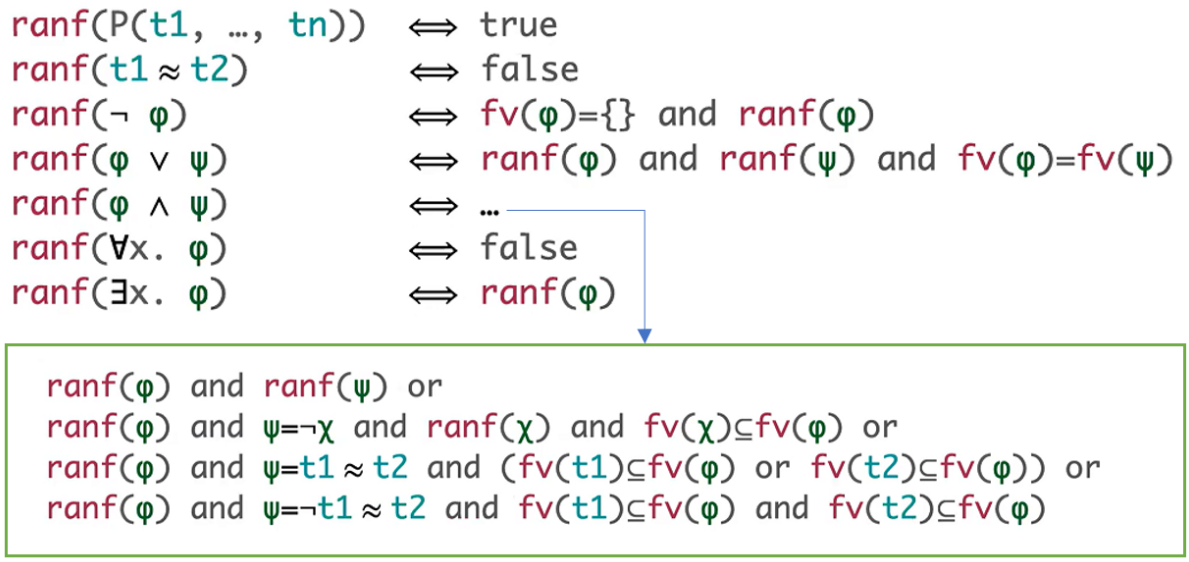
S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Figur 23: Example of tables from a boat rental. S:sailors, R:reserved.

4.1 Relational algebra normal form(RANF)

Is domain-independent, and thus always returns a finite result. Is syntactic overapproximation to ensure to always get a finite result. Each subformlua is also finite of a RANF.



Figur 24: Relational algebra normal form recursive definition.

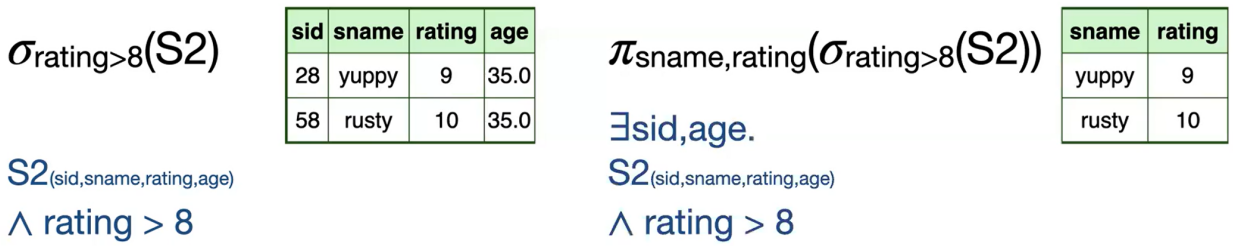
4.1.1 Codds theorem

Theorem 4.1 *For every domain-independent relational calculus query there exists an equivalent query in RANF.*

The theorem says that every domain independent in relational calculus, we can find a query in RANF that computes the same result.

4.2 Selection (σ)

Selects rows that satisfy the selection condition. Could e.g. be that a certain column has to have a value above x, be equal to x or such. Cant have duplicates in result, since we just have a subset of the original table, and the original table does not have duplicates (as it is a set).



Figur 25: Example of Selection, using tables in 23. In blue the equivalent is given in RC (note that ">"was not explicitly allowed in the version we did of RC). See in the example to the right, that we first transform it with a selection, and then take the output of that and use a projection.

4.3 Projection (π)

Removes columns. Takes as input the columns that we want to keep, and deletes the other columns not listed. It will delete duplicates in the final output since we are dealing with sets, and sets cant contain duplicates.

$\pi_{sname,rating}(S2)$

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\exists sid,age.\ S2_{(sid,sname,rating,age)}$

$\pi_{age}(S2)$

age
35.0
55.5

$\exists sid,sname,rating.\ S2_{(sid,sname,rating,age)}$

Figur 26: Example of projection, using tables in 23. In blue the equivalent is given in RC.

4.3.1 Extended projection with expressions (π_L)

The list L contains arbitrary expressions such as arithmic operations (e.g. column x multiplied by 2) or duplicate a column

$\pi_{rating+sid \rightarrow rs,age,age}(S1)$

rs	age	age
29	45.0	45.0
39	55.5	55.5
68	35.0	35.0

Figur 27: Example of projection, using tables in 23. Note that rating+sid is then renamed to rs.

4.4 Cross-Product ($\times$)

If we have $A \times B$, then each row of A is paired with each row in B. Like the Cartesian product. The resulting schema is all of A's combined with all of B's.

$S1 \times R1$

sid	sname	rating	age	sid	bid	day
22	dustin	7	45.0	22	101	10.10
22	dustin	7	45.0	58	103	11.12
31	lubber	8	55.5	22	101	10.10
31	lubber	8	55.5	58	103	11.12
58	rusty	10	35.0	22	101	10.10
58	rusty	10	35.0	58	103	11.12

$S1_{(sid1,sname,rating,age)} \wedge R1_{(sid2,bid,day)}$

Figur 28: Example of Cross product, using tables in 23. In blue the equivalent is given in RC.

4.5 Set-difference ($-$)

The set difference between the two tables. $A - B$, then the result is table A where all rows that are also in B is removed. Must be union compatible.

sid	sname	rating	age
22	dustin	7	45.0

$$S1 - S2$$

$$S1_{(sid,sname,rating,age)} \wedge \neg S2_{(sid,sname,rating,age)}$$

Figur 29: Example of set-difference, using tables in 23. In blue the equivalent is given in RC.

4.6 Union ($\cup$)

The union takes takes two tables and combines them into one. The tables must be union compatible: They must have the same number of fields (columns), corresponding fields have the same type.

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

$$S1 \cup S2$$

$$S1_{(sid,sname,rating,age)} \vee S2_{(sid,sname,rating,age)}$$

Figur 30: Example of Union, using tables in 23. In blue the equivalent is given in RC.

4.7 Intersection ($\cap$)

The intersect takes the "intersect"of the two tables, which means takes the rows that are in both tables. Must also be union compatible.

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$$S1_{(sid,sname,rating,age)} \wedge S2_{(sid,sname,rating,age)}$$

$$S1 \cap S2$$

Figur 31: Example of Intersect, using tables in 23. In blue the equivalent is given in RC.

4.8 Join ($\bowtie$)

4.8.1 Condition join ($\bowtie_C$)

Join two tables with a condition. Is equivalent of first taking the cross-product and then use selection with the condition. Also called theta join. The resulting schema is the same as cross-product.

$S1 \bowtie_{S1.sid<R1.sid} R1$

sid	sname	rating	age	sid	bid	day
22	dustin	7	45.0	58	103	11.12
31	lubber	8	55.5	58	103	11.12

$S1_{(sid1,sname,rating,age)} \wedge R1_{(sid2,bid,day)}$
$\wedge\ sid1 < sid2$

Figur 32: Example of Condition Join, using tables in 23. In blue the equivalent is given in RC.

4.8.2 Equijoin

A condition join, but the condition only contains equalities.

$S1 \bowtie_{sid} R1$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10.10
58	rusty	10	35.0	103	11.12

$S1_{(sid,sname,rating,age)} \wedge R1_{(sid,bid,day)}$

Figur 33: Example of Equijoin, using tables in 23. In blue the equivalent is given in RC.

4.8.3 Natural join

Natural join is just equijoin but on all columns. Combines all rows where the columns that are in both tables are equal.

4.9 Antijoin

Not covered in the lecture, but only mentioned on slides.

4.10 Divison ($\div$)

Not covered in the lecture, but only mentioned on slides.

4.11 Renaming (ρ)

Renaming can give a new schema to a table by changing the name of the columns. Often used if we later want to join in a specific way or when doing cross-product and we have two columns with the same name, we can rename one of them. Renaming can also be done when doing expressions and is denoted by an arrow.

4.12 Relational algebra on bags (duplicates allowed)

A bag is a multiset and is an unordered collection where duplicates are allowed. Like a set but allowed for duplicates. All sets are bags, but not all bags are sets (if they contain duplicates). SQL uses bag semantics.

$\pi_{age}(S2)$

age
35.0
55.5
35.0
35.0

$\sigma_{age<40.0}(\pi_{age}(S2))$

age
35.0
35.0
35.0

Figur 34: Example of transformation using bag semantics, using tables in 23.

All transformations are allowed in bag semantics, but in some transformations, like cross-product we no longer have to delete duplicates. One major difference is also in the unions, intersect and set-difference, since we now also have to look for how many times a certain element is in the bag.

4.12.1 Duplicate elimination (δ)

Deletes all duplicates in the table

$\pi_{age}(S2)$

age
35.0
55.5
35.0
35.0

$\delta(\pi_{age}(S2))$

age
35.0
55.5

Figur 35: Example of removing duplicates using bag semantics, using tables in 23.

4.13 Aggregation operators

Not a part of relational algebra, and will often only spit out one result, and thus not a table that other transformations can be used on. The most important are SUM, AVG, COUNT, Min, and MAX.

SUM(rating) = 32
COUNT(sid) = 4
MAX(age) = 55.5
AVG(rating) = 8

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Figur 36: Example of aggregate operators, using tables in 23.

4.14 Grouping operator (γ_L)

The list L either contains attributes or aggregate operators like AGG(A), where A is the attribute that the aggregate operator is being at. For each distinct value in R for the attributes given in L it forms a group. For each group it formed, use the aggregate operators given in L and name the column of the result of the aggregator as given with an arrow.

$\gamma_{\text{age,COUNT(rating)} \rightarrow \text{cr}}(S2)$

cr	age
3	35.0
1	55.5

$\gamma_{\text{age,MAX(rating)} \rightarrow \text{mr}}(S2)$

mr	age
10	35.0
8	55.5

Figur 37: Example of the grouping operator, using tables in 23. The first example: For each unique age in table S2, count how many ratings there are (renamed to cr). Second example: For each age, find the max rating for rows with that age (renamed to mr).

4.15 Sorting (τ_L)

Sort the table with regards to the first element given in L, then the second, third and so on.

$\tau_{\text{rating}}(S2)$

sid	sname	rating	age
44	guppy	5	35.0
31	lubber	8	55.5
28	yuppy	9	35.0
58	rusty	10	35.0

Figur 38: Example of the sorting operators, using tables in 23.

4.16 Outer join

When doing an outer join, we often have that not all rows are to be found in the other table, thus creating "dangling tuples"or null values. When doing outer join we preserve them by just inputting $\perp$ or by Null in SQL. Two variants are left/right outer join that preserves only dangling tuples in the left/right table. The regular just preserves both left and right.

$S1 \overset{\circ}{\bowtie}_L R1 = S1 \overset{\circ}{\bowtie} R1$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10.10
58	rusty	10	35.0	103	11.12
31	lubber	8	55.5	$\perp$	$\perp$

Figur 39: Example of the outer join operator, using tables in 23.

5 SQL

All direct SQL syntax is upper case like SELECT, WHERE, FROM, UNIQUE, etc. Same as with relational algebra, the result of one query can be used as the input of another.

5.1 Basic SQL query

The basic SQL query is of 3 steps. First, we SELECT what we want from the table, then FROM what table, and thirdly we can choose WHERE some condition in met. If we want to SELECT all columns in the query, we just put a stat "*".

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
[WHERE      condition]
```

Figur 40: Basic SQL query template. Optional are in brackets.

```
SELECT  S.Name
FROM    Sailors S
WHERE   S.Age > 25
```

```
SELECT DISTINCT S.Name
FROM   Sailors S
WHERE  S.Age > 25
```

Figur 41: Example of basic SQL query, using tables in 23. Finds names of sailors, from the database sailors renamed to S that are over 25 in age. The distinct keywords removes duplicates.

5.2 Select-Project-Join queries (SPJ)

The It first computes the cross-product between the given FROM tables, then discards rows that do not match in WHERE conditions, and finally removes columns not specified in SELECT. Remove duplicates if DISTINCT is put.

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid = R.sid AND R.bid=103
```

sid	sname	rating	age	sid	bid	day
22	dustin	7	45.0	22	101	10.10
22	dustin	7	45.0	58	103	11.12
31	lubber	8	55.5	22	101	10.10
31	lubber	8	55.5	58	103	11.12
58	rusty	10	35.0	22	101	10.10
58	rusty	10	35.0	58	103	11.12

Figur 42: Example of a SPJ SQL query, using tables in 23. Finds name of sailors who reserved boat 103. The tables shows how it conceptually computes it by first finding the cross-product, then matching S.sid=R.sid and then R.bid=103 and lastly returning the name of that row.

5.3 CREATE TABLE

Creates a table with given columns, their name and types. One column can only have one type.

```
CREATE TABLE Employees
            (ssn CHAR(11),
             name CHAR(20),
             lot  INTEGER,
             PRIMARY KEY (ssn))
```

```
CREATE TABLE Departments
            (did INTEGER,
             dname CHAR(20),
             budget FLOAT,
             PRIMARY KEY (did))
```

Figur 43: Example of creating two tables with 3 columns.

5.3.1 PRIMARY KEY

The primary key is unique and can consist of one or multiple columns. See the section 2.2.1 on keys.

5.3.2 FOREIGN KEY

When doing relationships or such we use the keys of the entities to create a key that combines their keys. We therefore use FOREIGN KEY (newNameForKey) REFERENCE Entity. SQL automatically knows the key of the entity, so no need to specify, and we can therefore just rename the key to avoid confusion.

```
CREATE TABLE Works_In(
        ssn  CHAR(11),
        did  INTEGER,
        since  DATE,
        PRIMARY KEY (ssn, did),
        FOREIGN KEY (ssn) REFERENCES Employees,
        FOREIGN KEY (did) REFERENCES Departments)
```

Figur 44: Example of using a foreign key. Creates a relationship table by combining the two tables used in example 43.

When referencing a foreign table, we can specify what should happen if something is updated or deleted in the table we refer to. We then have 4 things we can do: NO ACTION, CASCADE, SET NULL, SET DEFAULT. They are covered in the next sections.

5.3.3 NO ACTION

When something is updated/deleted in the reference table, we just do nothing.

5.3.4 CASCADE

Delete/update all tuples that refer to deleted/updated tuples. Often used in weak entities, since a weak entity cant exist without its parent, and therefore must also be deleted.

```
CREATE TABLE  Dep_Policy
  (pname  CHAR(20),
   age  INTEGER,
   cost  REAL,
   ssn  CHAR(11) NOT NULL,
   PRIMARY KEY  (pname, ssn),
   FOREIGN KEY  (ssn) REFERENCES Employees
      ON DELETE CASCADE)
```

Figur 45: Example of using CASCADE. The reference table is Employees, and the new table Dep_Policy is a weak entity of Employees. When a row in Employees is deleted, then the corresponding row in Dep_Policy is also deleted.

5.3.5 SET NULL

When an update/deletion happens in the reference table, just set the value to NULL.

5.3.6 SET DEFAULT

When an update/deletion happens in the reference table, just set the value to a default value.

5.4 Inserting rows

Insert a row by defining the values by each column manually or by taking existing values from another table using a SELECT statement.

```
INSERT INTO Table
  VALUES(A1,A2,…,An)
```

```
INSERT INTO  Students (sid, name, login, age, gpa)
VALUES  (53688, 'Smith', 'smith@ee', 18, 3.2)
```

Figur 46: Row insert template and example.

```
INSERT INTO Table
  Select-Statement
```

```
INSERT INTO Students(sid, name, login, age, gpa)
SELECT NULL, name, login, age, 0.0
FROM   Other_Students
WHERE  school = 'KU'
```

Figur 47: Row insert with SELECT statement template and example.

5.5 Deleting rows

Delete all rows that match the WHERE statement.

```
DELETE FROM Table
WHERE Condition
```

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

Figur 48: Delete rows with WHERE statement template and example.

5.6 Updating rows

Update all rows that match the WHERE statement is true and SET the value to x.

```
UPDATE Table
SET A1=Expr1,A2=Expr2,…,An=Exprn
WHERE Condition
```

```
UPDATE Employees
SET salary = salary * 1.1
WHERE age >= 36
```

Figur 49: Update template and example. This updates all salaries by 10% for employees over the age of 35.

5.7 UNION

UNION just combines the two tables or results if used between two different queries. UNION ALL does the same but keeps duplicates.

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
  AND R.bid=B.bid
  AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
  AND R.bid=B.bid
  AND B.color='green'
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
  AND R.bid=B.bid
  AND B.color='red'
UNION ALL
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
  AND R.bid=B.bid
  AND B.color='green'
```

Figur 50: Example of UNION and UNION ALL, using tables in 23. Finds sid's of sailors who reserved a red or a green boat.

5.8 INTERSECT

Takes what is in both two tables or the result if used between queries then the result has to be in both queries.

```
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
  AND  B.color='red'
INTERSECT
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
  AND  B.color='green'
```

Figur 51: Example of INTERSECT, using tables in 23. Finds sid's of sailors who reserved a red and a green boat.

5.9 EXCEPT

The set difference. Removes all rows that happened to be in the other table/result of query.

```
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
  AND  B.color='red'
EXCEPT
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
  AND  B.color='green'
```

Figur 52: Example of EXCEPT, using tables in 23. Finds sid's of sailors who reserved a red and not a green boat.

5.10 Joins

5.10.1 CROSS JOIN

The same as the cartesian product. Is used between two tables, and can also just be denoted with a comma ","between the tables.

MovieStar

name	address
Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23

```
SELECT *
FROM MovieStar CROSS JOIN MovieExec
```

MovieExec

name	address
Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Sandra Bullock	564 Center B

name	address	name	address
Harrison Ford	789 Palm	Harison Ford	789 Palm
Harrison Ford	789 Palm	Iben Hjejle	Øster Alle 4
Harrison Ford	789 Palm	Sandra Bullock	564 Center B
Iben Hjejle	Øster Alle 4	Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4	Iben Hjejle	Øster Alle 4
Iben Hjejle	Øster Alle 4	Sandra Bullock	564 Center B
Mads Mikkelsen	Sverresgata 23	Harison Ford	789 Palm
Mads Mikkelsen	Sverresgata 23	Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23	Sandra Bullock	564 Center B

Figur 53: Example of CROSS JOIN. Would have the same result if we just put a comma instead of CROSS JOIN.

5.10.2 Theta Join

Use CROSS JOIN but have a WHERE statement to specify what columns have to be equal. Or use the JOIN and ON statement to specify what column should be compared.

```
SELECT S.*, E.*
FROM   MovieStar S,
       MovieExec E
WHERE  S.name = E.name
```

```
SELECT S.*, E.*
FROM   MovieStar S
       JOIN MovieExec E
       ON S.name = E.name
```

Figur 54: Example of Theta joins.

5.10.3 Equijoin

Using INNER JOIN to join the two tables USING to denote what column that has to be equal for the two tables.

```
SELECT *
FROM   MovieStar S
       INNER JOIN MovieExec E
       USING (name)
```

Figur 55: Example of Equijoin.

5.10.4 NATURAL JOIN

The columns that the two tables have in common they must match. Like equijoin on all common columns.

```
SELECT *
FROM MovieStar
     NATURAL JOIN MovieExec
```

Figur 56: Example of NATURAL JOIN.

5.10.5 OUTER JOIN

A special join that also joins non matching rows, but can then have dangling tuples.
LEFT OUTER JOIN: Returns non-matching tuples from the left table.
RIGHT OUTER JOIN: Returns non-matching tuples from the right table.
FULL OUTER JOIN: Returns non-matching tuples from both tables.

```
SELECT DISTINCT S.sid, R.bid
FROM   Sailors S
       LEFT OUTER JOIN
       Reserves R
       ON S.sid = R.sid
```

sid	bid
22	101
58	103
31	NULL

Reserves

sid	bid	day
22	101	10.10
58	103	11.12

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figur 57: Example of LEFT OUTER JOIN.

5.11 Null values

Values in rows that are missing/unknown/not assigned or such are denoted by Null.

5.11.1 Three-valued logic

Since we sometimes use predicate logic to compare fields e.g. age>20, when age is then Null then the whole operation return Null, thus creating three-valued logic.

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
NULL	TRUE	NULL	TRUE	NULL
NULL	NULL	NULL	NULL	NULL
NULL	FALSE	FALSE	NULL	NULL
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Figur 58: Truth table of the three-valued logic.

5.11.2 IS NULL

Since predicate logic just returns Null when compared to Null, if we then want to know if some value is null, we then use IS NULL, that returns true if the value is null.

```
SELECT S.sid
FROM   Sailors S
       LEFT OUTER JOIN
       (SELECT sid
        FROM   Sailors NATURAL JOIN Reserves R
        WHERE  R.bid = 103) AS S_103
       ON S.sid = S_103.sid
WHERE S_103.sid IS NULL
```

Figur 59: Example of using IS NULL.

5.12 Expressions and strings (AS and LIKE)

When doing arithmetic operations on an entire column (e.g. multiply all by 2 or subtract 5) we can then rename the result of that using AS.
To match string we can use LIKE where "_"matches a single arbitrary character and "%"matches 0 or more arbitrary characters.

```
SELECT  age, age-5 AS age5, 2*age AS age2
FROM    Sailors
WHERE   sname LIKE '_u%'
```

Figur 60: Example of using AS and LIKE. Finds the age, age-5 and age times 2 of sailors who's names second letter is u.

5.13 table expression WITH

Define a table outside as a query using WITH that can be used in a later query. It is like a nested query, but to keep the queries simpler, we can move it out and use a WITH instead or if we want to use it multiple times.

```
WITH S_103(sid) AS (
  SELECT sid
  FROM   Sailors NATURAL JOIN Reserves
  WHERE  bid = 103
)
SELECT S.sid
FROM   Sailors S
       LEFT OUTER JOIN
       S_103
       ON S.sid = S_103.sid
WHERE S_103.sid IS NULL
```

Figur 61: Example of WITH. Here it creates a new table called S_103 with one column sid.

5.14 Nested queries with correlation

When doing a nested query, it is also possible to reference the parent query like in example 62 below.

```
SELECT  S.sname
FROM    Sailors S
WHERE   EXISTS (SELECT  *
                FROM  Reserves R
                WHERE  R.bid=103 AND S.sid=R.sid)
```

Figur 62: Example of a nested query with correlation. Here the nested query uses the sid from the parent query. This query returns all names of sailors who have reserved boat 103.

5.15 Set-comparison operators (ANY/ALL)

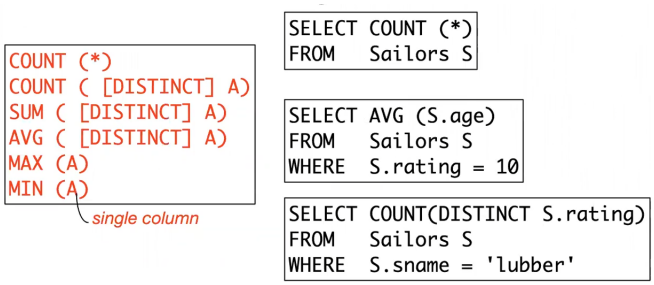
Using the operators $>, <, =, <=, >=, <>$ (denoted as "op", short for operator), we can use op ANY or OP all.
op ANY: Checks if any (atleast one) satisfies this.
op ALL: Checks if all satisfies this.

```
SELECT *
FROM   Sailors S
WHERE  S.rating > ALL (SELECT  S2.rating
                       FROM  Sailors S2
                       WHERE S2.sname='lubber')
```

Figur 63: Example of a using op ALL. The query finds all sailors whose rating is greater than all sailors called lubber.

5.16 Aggregate operators

Same as relational algebra covered in section 4.13. Also only gives one value,



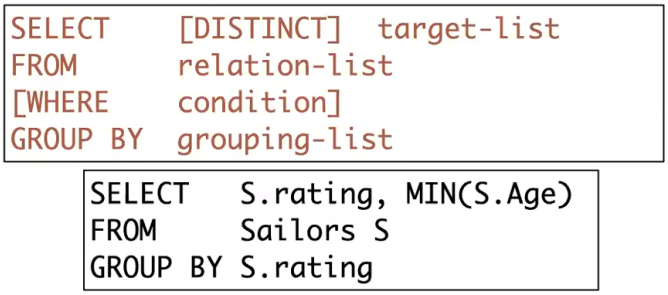
Figur 64: All aggregate operators and some examples. First counts all sailors, second finds the average age of sailors with a rating of 10, and the third counts ratings (without duplicates) of sailors with the name of lubber.

5.17 GROUP BY

If we want to find the minimum age of all the different ratings, like in the example 65 below. This is not good, since we won't get the result in a table, but rather 10 different values just spit out.

Figur 65: Example of getting the minimum age for all different ratings.

To fix this we group them, as covered in relational algebra in section 4.14. It again create a group for each distinct row of the attribute we group over, and then use the aggregate operator for that value.



Figur 66: Query template when using GROUP BY and example of getting the minimum age for all different ratings using GROUP BY.

5.18 HAVING

HAVING is used in combination with GROUP BY to only make a group if the condition is met.

```
SELECT   [DISTINCT]  target-list
FROM     relation-list
[WHERE   qualification]
GROUP BY grouping-list
HAVING   group-qualification
```

```
SELECT   S.rating, MIN(S.Age)
FROM     Sailors S
WHERE    S.age >= 18
GROUP BY S.rating
HAVING   COUNT(*) > 1
```

Figur 67: Query template when using HAVING, and example. The query finds the age of the youngest sailor with age >=18 for each rating with at least 2 such sailors.

5.19 ORDER BY (sorting)

Sorts/orders the result by a given attribute.

```
SELECT   S.rating, AVG(S.age) AS avgage
FROM     Sailors S
GROUP BY S.rating
ORDER BY avgage
```

Figur 68: Example of ORDER BY. The query finds the average age for each rating and then orders the results in ascending order by average age.

5.20 CONSTRAINT

A query can be used to create a CONSTRAINT, and can also be named. Is like adding an integrity constraint (see section 2.2). If the constraint is not met, we can tell the DMBS to handle it, and e.g. not allow a row/tuple that does not satisfy the constraint.

```
CREATE TABLE  Reserves (
      sname  CHAR(10),
      bid  INTEGER,
      day  DATE,
      PRIMARY KEY  (bid,day),
      CONSTRAINT  noInterlakeRes
      CHECK  ('Interlake' <>
                  (SELECT  B.bname
                  FROM  Boats B
                  WHERE  B.bid=bid)))
```

Figur 69: Example making a CONSTRAINT on a table. This constraint make it such, that the table must not contain a boat name called "Interlake".

5.21 ASSERTION

A constraint over multiple tables/relations.

```
CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

Figur 70: Example making an ASSERTION over multiple tables. This assertion makes sure that the sum of numbers of sailors and boats is below 100.

6 Data redundancy

6.1 Functional Dependencies (FDs)

A functional dependency between sets of attributes X and Y holds over a relation R if, for every instance r of R is

$$\pi_X(t) = \pi_X(u) \text{ implies } \pi_Y(t) = \pi_Y(u) \text{ for all } t \in r \text{ and } u \in r \tag{1}$$

In words, it means that if the X value is the same for two tuples, then the Y value must also be the same. Hence it is like a function, where one x value must correspond to exactly one y value. An example could be if sailor A has a rating of 9 and thus an hourly wage of 100, and if sailor B also has a rating of 9 then they have a functional dependency if B must have an hourly wage of also 100.

6.1.1 Closure

$F+$ is the closure of a set of FDs that are implied by F. E.g. if we have the FDs $\{K \to B, B \to D, B \to A, O \to P\}$ then $K+ = \{K, B, D, A\}$.

6.1.2 Amstrongs Axioms

Reflexivity: If $Y \subseteq X$, then $X \to Y$.
Augmentation: If $Z \to Y$, then $XZ \to YZ$ for any Z.
Transitivity: If $X \to Y$ and $Y \to Z$, then $X \to Z$.
Following from those axioms are the following:
Union: If $X \to Y$ and $X \to Z$ then $X \to YZ$.
Decomposition: If $X \to YZ$, then $X \to Y$ and $X \to Z$

6.1.3 Example of using FD axioms

Given a schema $Contracts(cid, sid, jid, did, pid, qty, value)$ (short: $CSJDPQV$) where C is the key and therefore $C \to CSJDPQV$. We have to FDs other than the key: $JP \to C$ and $SD \to P$. Check whether $SDJ \to CSJDPQV$:

$$SDJ = JSD \xrightarrow[SD \to P]{\text{Augmentation}} JP \xrightarrow[JP \to C]{\text{Transitive}} C \to CSJDPQV$$

And hence SDJ is a super key for the table.

6.2 Decomposition of a relational scheme

Instead of storing the entire table, we can use functional dependencies to decompose the table into multiple tables to avoid too much redundancy.

6.2.1 Problems with decompositions

We must always consider the trade-off: Issues listed below vs. redundancy.

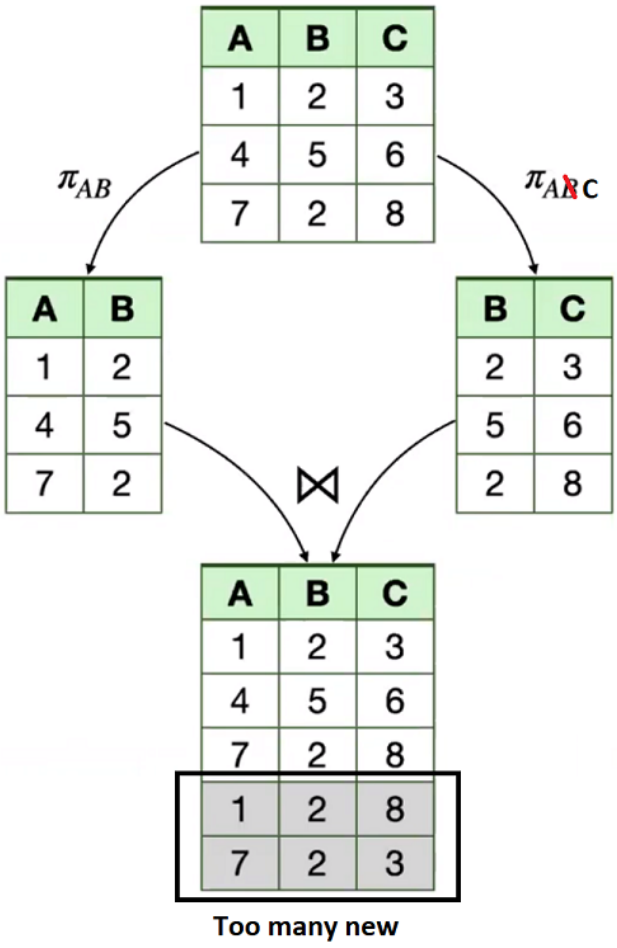
- Expensive: Some queries become more expensive. E.g. if we have to multiply two columns together.
- Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation.
- Checking dependencies may require joining the instances of the decomposed relations.

6.2.2 Lossless-join decomposition

Decomposing R into X and Y is a lossless-join with respect to a set of FDs if for every instance r satisfies the set of FDs:

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

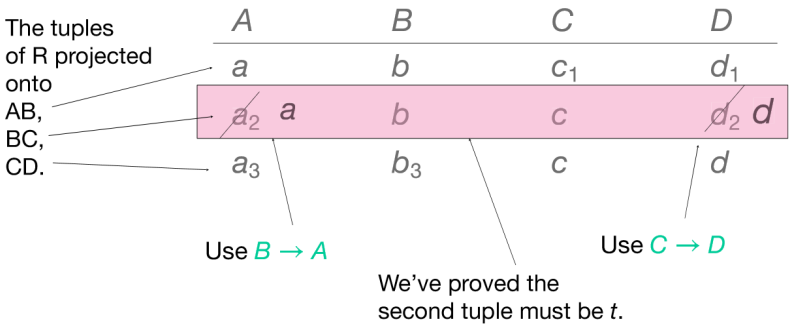
In other words, it means that we must not generate more or less tuples than the original. E.g. we have to be able to reconstruct the original table again using the decomposition.
This avoids problem 2 listed in section 6.2.1.



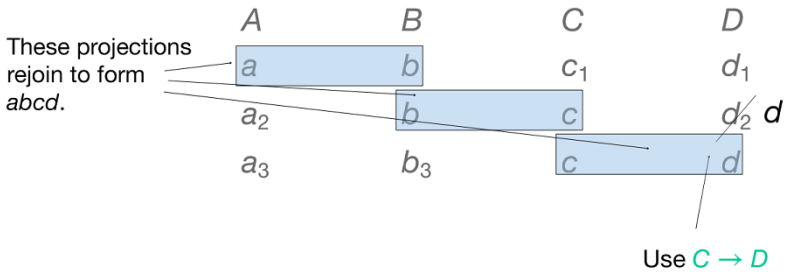
Figur 71: Example of a decomposition that creates new tuples, and is therefore not a lossless-join.

6.2.3 The chase test

The chase test can be used to check whether a decomposition is a lossless join. Given a decomposition and FDs, we join the decompositions and then use FDs to try and prove whether the tuples in the join must be in the reconstructed table or not.



Figur 72: Example of a chase test for if the tuple $t = abcd$ is in the table $R = ABCD$, decompositions AB, BC and CD and FDs $C \to D$ and $B \to A$. This decomposition is lossless, since we can prove that the tuple is in the join using FDs.



Figur 73: Example of a chase test for if the tuple $t = abcd$ is in the table $R = ABCD$, decompositions AB, BC and CD and FD $C \to D$. This decomposition is lossy, since we cannot prove that the tuple is not in the join.

6.2.4 Dependency-preserving decomposition

If a table R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, Y and Z, then all FDs that are given to hold on R must also hold. This avoids problem 3 is section 6.2.1.
A decomposition is a dependency preserving if the closure of the union of FDs of the decomposed tables is equal to the closure of the FDs from the original table. I.e. the decomposition uses the original FDs to decompose to preserve dependency. Given formally as

$$(F_X \cup F_Y)+ = F+$$

A dependency preserving decomposition does not imply a lossless join.

6.3 Boyce-Codd Normal Form (BCNF)

A table is in BCNF if, for all $X \to Y$ in $F+$, $Y \subseteq X$ or X contains a key for the table.

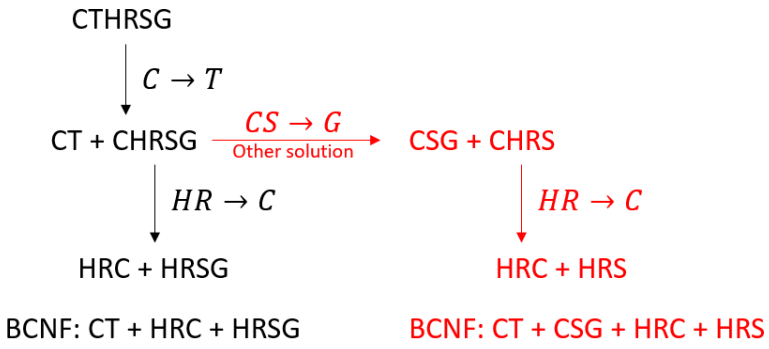
6.3.1 Method to decompose into BCNF

Given a table R with FDs F. If $X \to Y$ violates BCNF, then decompose it into $R - Y$ and XY. We keep doing this until all tables are in BCNF.

The result is not unique when decomposing into BCNF, as we can decompose in different orders.
First we start of by finding all keys. And then check for all FDs if they violate, and if they do, then "remove"the right side from the table and create a new table with the left (X) and the right side (Y).

6.3.2 Example of decomposing

An example of decomposing the table CTHRSG with the FDs:
$C \to T$, $HR \to C$, $HT \to R$, $HS \to R$ and $CS \to G$.



Figur 74: Two examples of decomposing the table into BCNF.

Here we have shown two different decompositions. We keep dividing the tables until no Y in FD $X \to Y$ is in the table along with a X. In other words, we always try to remove the "right side"of all FDs as long as the "left side"is existing in the table. Walking through each example:
The black decomposition:
First we establish a key, which in this example is HS (just find the closure of each FD and find one that can find all columns).

1. C and T are in $CTHRSG$, and we, therefore, have to "remove"T from the table, and thus decompose into $CT + CHRSG$.
2. HR and C are in the table; therefore, we have to "remove"C by creating a new table HRC and removing C from the old table.
3. We then look through all FDs, and see that no FD has its "left side"and "right side"in the table $HRSG$:
   - $C \to T$ are both not present.
   - $HR \to C$, C is not present.
   - $HT \to R$, T is not present.
   - $HS \to R$ are BOTH present, but as HS is the key, it cannot be split up further. [1]
   - $CS \to G$, C is not present.

The red decomposition:
We do the same thing, but we just take a different order of FDs to decompose from.

6.4 Third normal form (3NF)

A table R with FDs F is in 3NF if for all $X \to Y$ in F^+:

- $Y \subseteq X$ (called trivial FD) or
- X contains a key for R or

[1] See if we split it, it would instead increase data redundancy since the entire key would have to be present in two places.

- Each attribute in Y is part of some key for R (each attribute in Y is prime)

If a table is BCNF, it is also 3NF. Some redundancy is possible compared to BCNF, but used when BCNF gives worse performance or other considerations.
Algorithm to convert into 3NF is not given is this course.

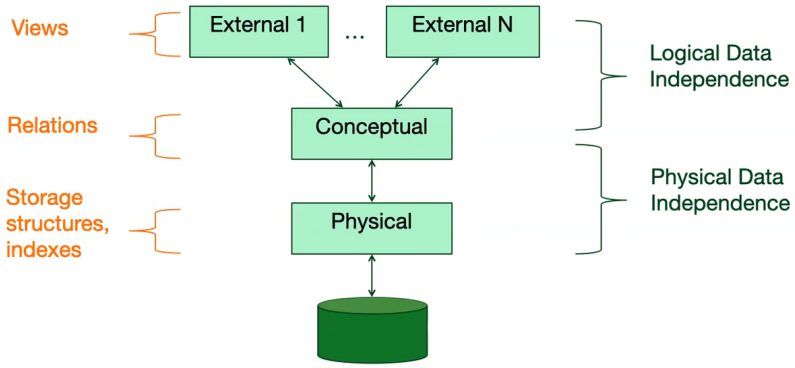
6.5 Normal Forms

- First normal form (1NF).
  - The domain of each attribute contains only atomic values.
  - No duplicate data.
  - Each entry contains only one value.
  - All columns are regular.
- Second normal form (2NF).
  - 1NF.
  - No non-prime attribute is dependent on any proper subset of any key of the table.
- Third normal form (3NF).
  - 2NF.
  - Every non-prime attribute is directly dependent on every superkey of the table.

7 Other stuff covered in last lecture

7.1 VIEWs

Derived tables from our database using logic. Is used to show data to a higher-level user, e.g. to a customer, who does not need to see the entire table, but maybe only one or two columns. It can also be made to make queries easier/faster/natural. Are used a lot in "real"applications.



Figur 75: Structure of DBSM

```
CREATE VIEW Red_Green_Sailors (sid, sname) AS
  SELECT DISTINCT S.sid, S.sname
  FROM   Sailors S, Boats B, Reserves R
  WHERE  S.sid=R.sid AND R.bid=B.bid
    AND  (B.color='red' OR B.color='green')
```

```
DROP VIEW Red_Green_Sailors
```

Figur 76: Example of a view in SQL. This creates a view that only contains the sid and sname of sailors who have reserved either a red or green boat.

7.1.1 Virtual vs. materialized view

A virtual view does not "create"the view when it is made, but only when needed. A materialized view is created and stored when it is made. One problem is that some DBMS does not update the view when the original table is updated, hence the view does not have the correct data. We always need to consider advantages when choosing.

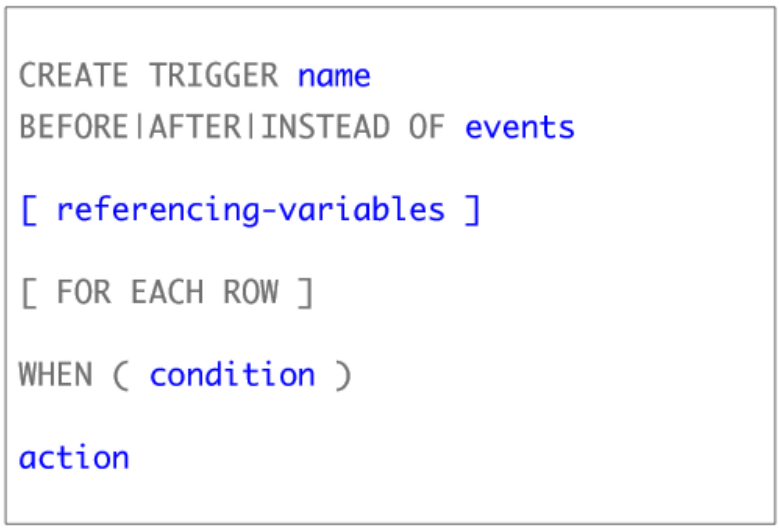
```
CREATE MATERIALIZED VIEW CA-CS AS
SELECT C.cName, S.sName
FROM College C, Student S, Apply A
WHERE C.cName = A.cName AND S.sID = A.sID
AND C.state = 'CA' AND A.major = 'CS'
```

Figur 77: Example of a materialized view in SQL.

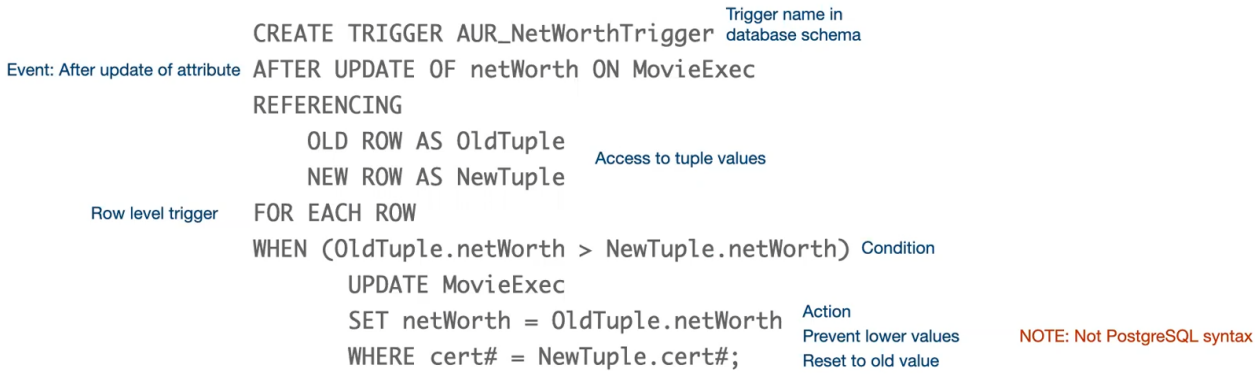
A materialized view can be updated whenever the original table is changed, daily, event-based (when e.g. 10 updates have been made), or at some other specified time. If we do the update immediately after an update we get a view that is always updated, it can however be expensive. And the other way around of only doing it e.g. daily.

7.2 TRIGGER

An event condition action rule: When an event occurs, check some condition; if true, do some action. We can check events on either row level or statement level. Row level is updated to a single row, and statement level is single update that changes one or multiple rows.



Figur 78: Trigger template in SQL



Figur 79: Trigger example in SQL, NOT pSQL syntax. Whenever networth of a movieExec is updated we check whether it is larger than their old netWorth. If it is not larger, we keep the old netWorth.

Using postgressSQL we have to use python to handle the trigger, and it gets complex quite quickly.

7.3 Transactions

A transaction is made to reliably have concurrency in a DBMS, such that multiple users can insert/update tables. It has to have the ACID properties in order to reliably do this:

- Atomicity: transactions are all-or-nothing (we make them to the end or we don't)
- Consistency: transactions take database from one consistent state to another
- Isolation: transaction executes as if it was the only one in the system.
- Durability: once the transaction is "committed", results are persistent in the database

Transaction T1: TRANSFER

```
BEGIN;
  UPDATE accounts
  SET balance = balance - 10
  WHERE name = 'Dmitriy';
  --
  UPDATE accounts
  SET balance = balance + 10
  WHERE name = 'Magnus';
COMMIT;
```

Transaction T2: INTEREST

```
BEGIN;
  UPDATE accounts
  SET balance = balance * 1.01;
COMMIT;
```

- BEGIN and COMMIT keywords delimit transaction
- COMMIT confirms that work should be made durable in the database

Figur 80: Trigger example in SQL. The code within "begin"and "commit"is executed atomically.

7.4 Index

Normally when the DBMS searches for instances e.g. when joining or searching it uses a naive nested approach, which is usually relatively slow. To improve this, we can order the data using indexing. E.g. strings alphabetically or numbers sorted (but only one column). This will improve the runtime by much since the DBMS knows when to stop and hence does not need to look through the entire table.

```
SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

(a) 25s

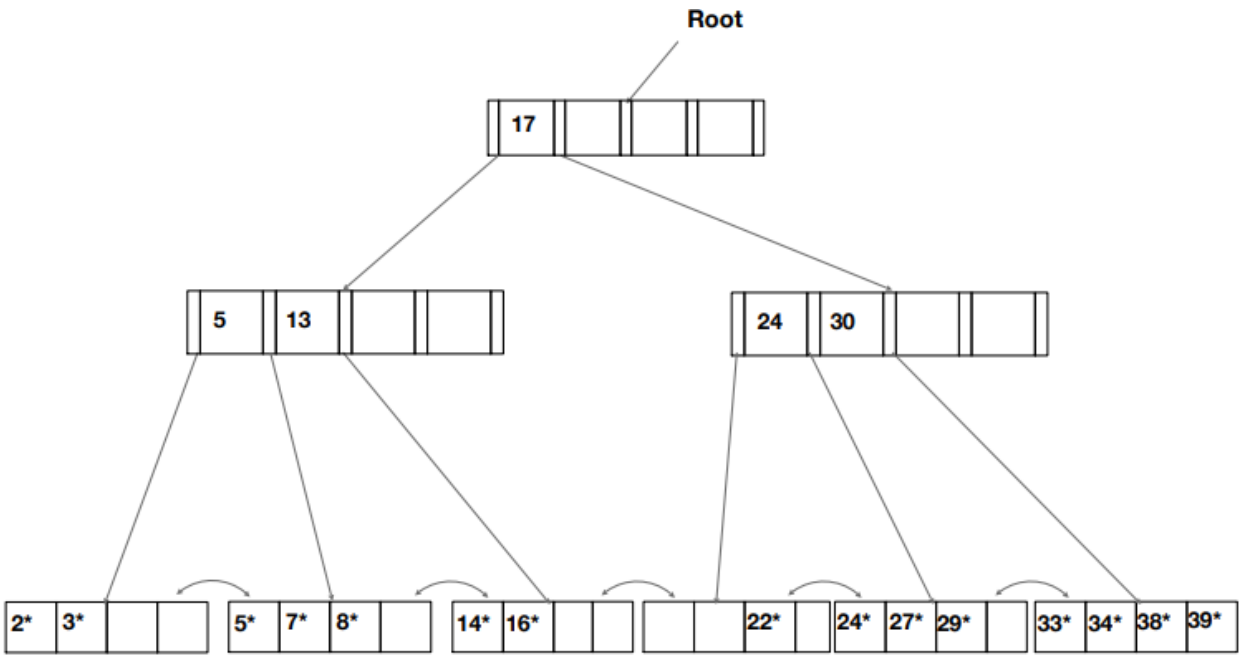
```
CREATE INDEX covidi ON covid(location, new_cases, date);
SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

(b) 7.5s

Figur 81: Two queries, that compute the same, but (b) uses indexing.

7.5 B+ Tree algorithm

The B+ tree is one of the most used indexing for range indexing. It is like a binary tree, but with multiple branches for each node. Each node normally has up to between 100-200 fanouts. Each "node"is usually a set of entries with space for $2d$ entries, and thus must always remain at least half full with d entries (except the root). Do note, that only the bottom (leaf nodes) have the data, the other nodes are only for indexing (noted by a star in the figures)



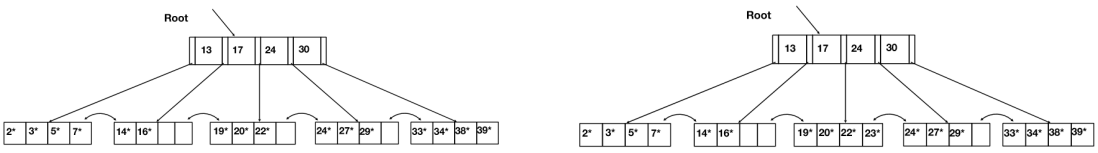
Figur 82: Example of a B+ tree

7.5.1 Searching

When searching for an element in a B+ tree just use the same procedure as with a binary tree, but look at ranges instead. E.g. from 82 above, if we wanted to search for 7, we first compare $7 < 17$, and go left. Then we see that $5 < 7 < 13$ and then go to the subtree between 5 and 13, and then just look through that subset to find 7.

7.5.2 Inserting

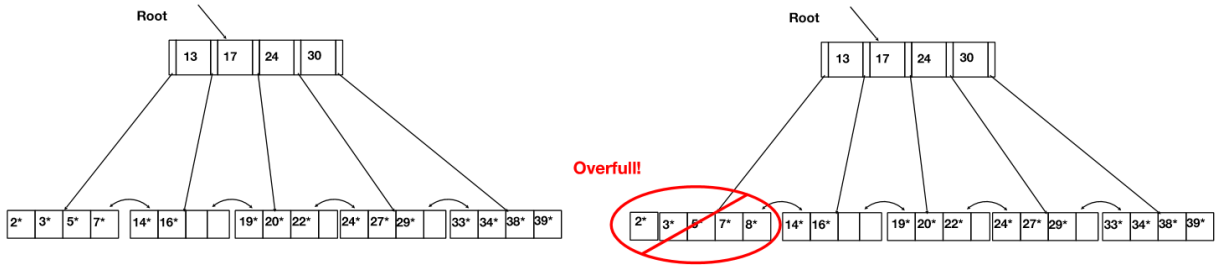
When inserting an element we just search for it, and then place it where we end up in the search. If the subset is full, we have to split the overfull node up, and create two new. Without overfilling see example 83.



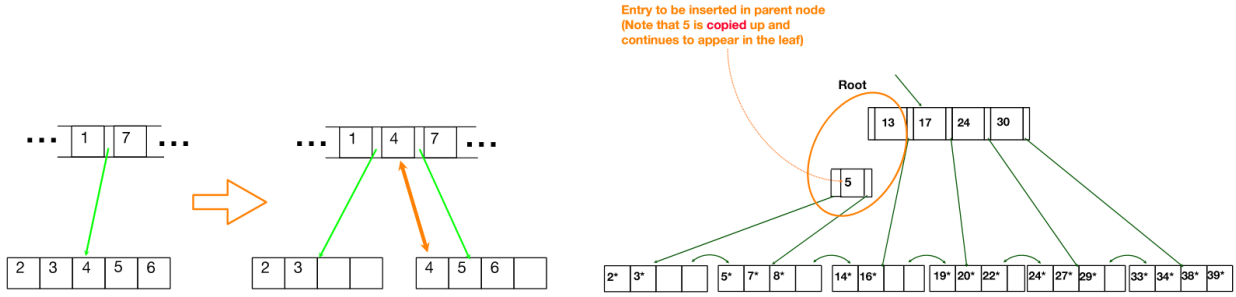
(a) Before (b) After

Figur 83: Inserting 23 into the tree

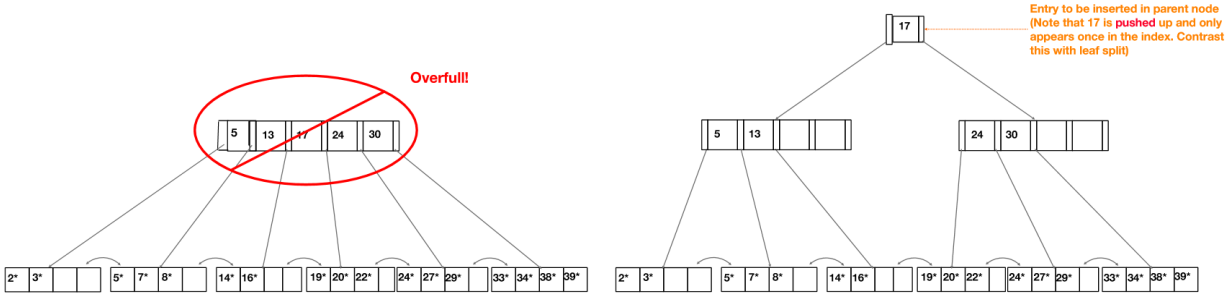
As mentioned when the subset node is full, we have to split the overfull node. To do this, we move the middle of the subset into its parent node to create two new ranges. However, This problem can cascade up, as the parent set can also get overfull, but we just do the same procedure. See the examples in the figures below where we try to insert 8 into a full subset:



Figur 84: First we search for 8, and then insert it into the leaf node. We note that it is full, so we have to split it.



Figur 85: We split the leaf node and create a new leaf node to the right, we then take the first node to the right of the middle as the lowest in the new node. This node is then larger than everything in the old and smaller than everything in the new node, so we use that in the parent node for indexing.

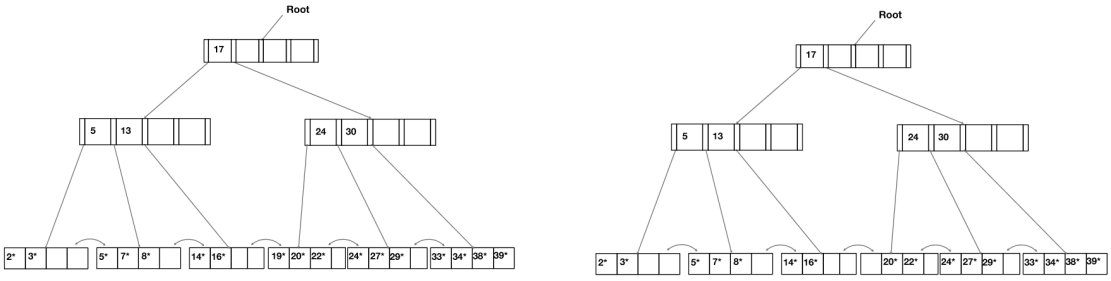


Figur 86: When 5 is inserted in the parent node, we see that it is also full, we then have to split. We take the middle element, and use that as the new root. Note that since these are all indexes and not actual datapoints, 17 is only seen once in contrary when we had to split the leaf node and make two 5's. Also note that the tree height now has increased.

7.5.3 Deleting

Search for the element, and then just remove it, see figure 87. That was the easiest of the 3 cases that a delete can create. First is do nothing, since it is not underfull. The next is redistribute if the node is underfull. And the third is merging/deleting nodes.

First case: Do nothing

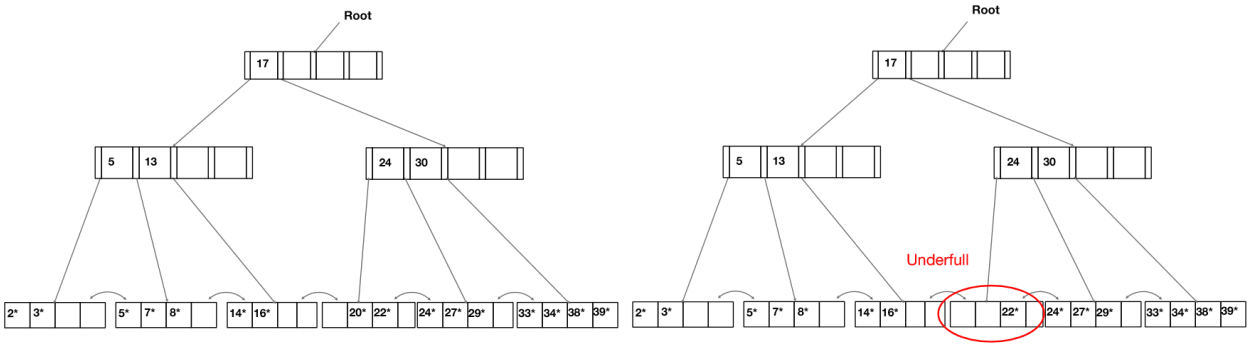


(a) Before (b) After

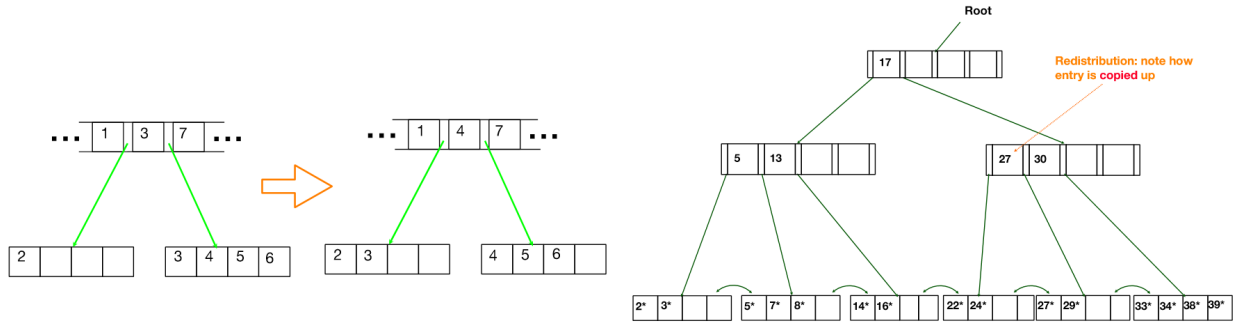
Figur 87: Deleting 19* from the tree. Case 1, do nothing

Second case: Redistribute

If the node is underfull we have to redistribute the remaining elements, se example below of deleting 20*:



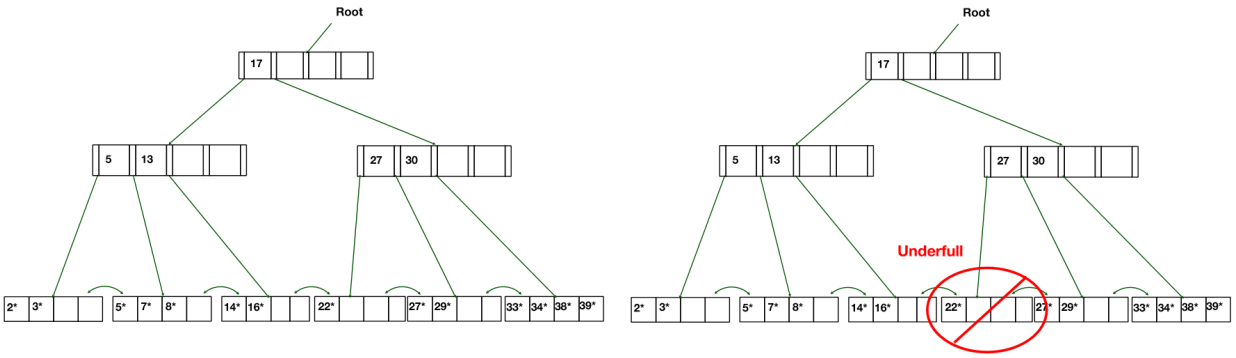
Figur 88: Remove 20*, and we note that the leaf note is now underfull.



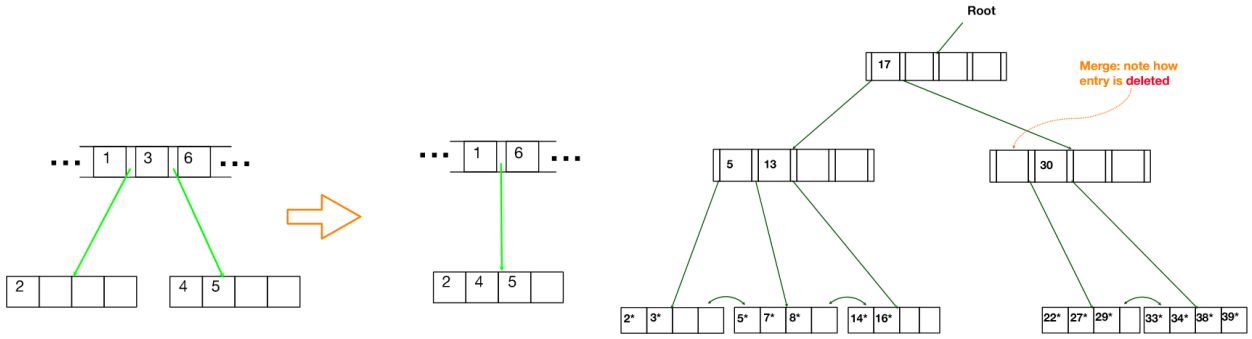
Figur 89: Since it is underfull, we have to redistribute the elements from another leaf to the underfull node. We do this by taking the smallest element from its right sibling (or largest from the left sibling), changing the index to fit the new intervals and put the element in the leaf node. The left image is just an example of how to redistribute from the right to the left node.

Third case: Merge/delete

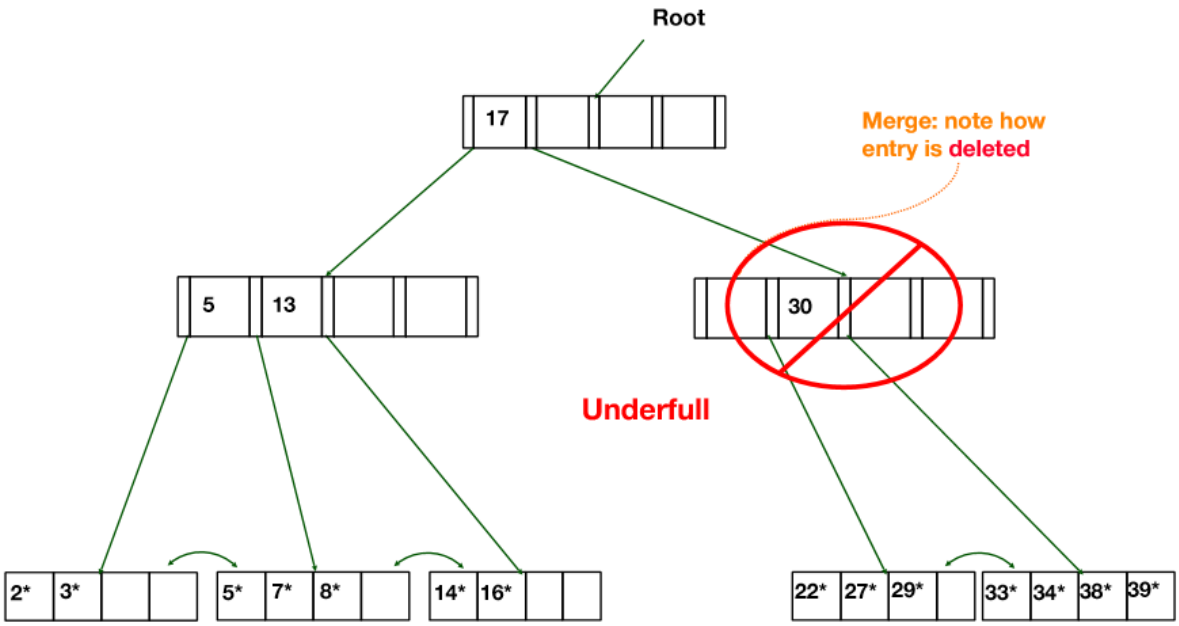
In the last case we again remove another element making the node underfull, but now its sibling also becomes underfull if we redistribute. We now have to merge/delete a node. See example below of deleting 24*:



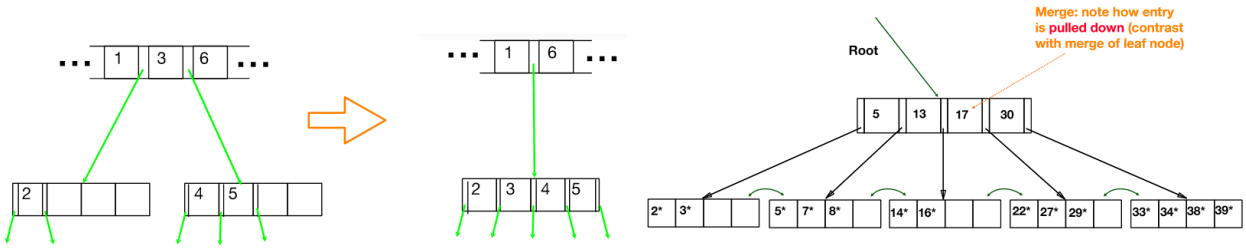
Figur 90: First we find and delete 24*. And then note that the leaf node is underfull.



Figur 91: Since the sibling node only had 2 elements, and thus will become underfull if we redistribute, we have to merge. We therefore delete the underfull node, and if the sibling has room (which this has), we move the element into that node, thus the nodes have been merged. We then also remove the index for the range of the deleted node.



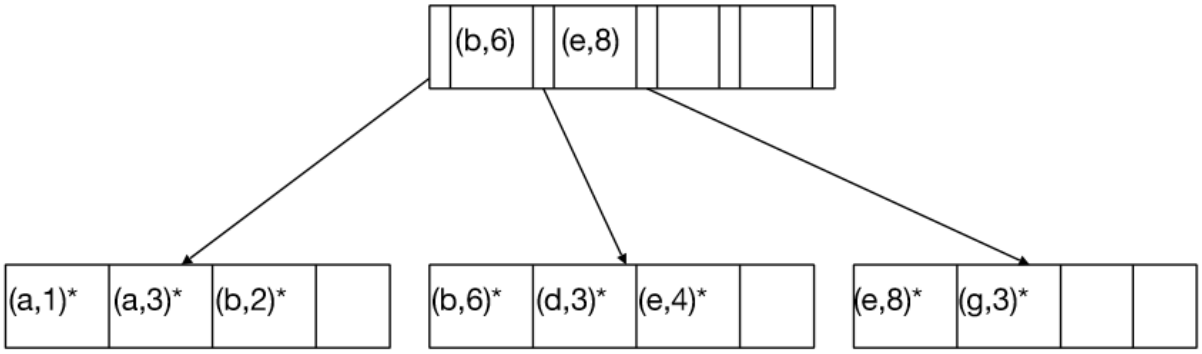
Figur 92: Since we deleted from the parent index node, this has become underfull. We have to handle this as well.



Figur 93: We then merge the index nodes, thus decreasing the tree height.

7.5.4 Composite key search

Each element in e.g. a B+ tree can have multiple values (represented by multiple columns in a table for example). But when ordering the data by indexing, we can only do that for one variable for the first order, but we can use another variable as a second order sorting criteria if two first order variables are equal.



Figur 94: B+ tree with multiple values. Sorted after first value as a first order, and then the second value if the first are equal.