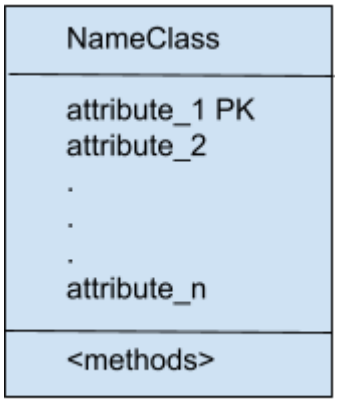
BDAD

1. UML - Unified Modeling Language

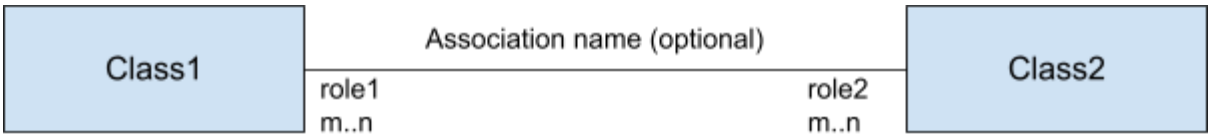
1.1. Classes

- NameClass: usually in the singular
- We can declare a primary key (PK)
- <methods>: don't matter for data modeling



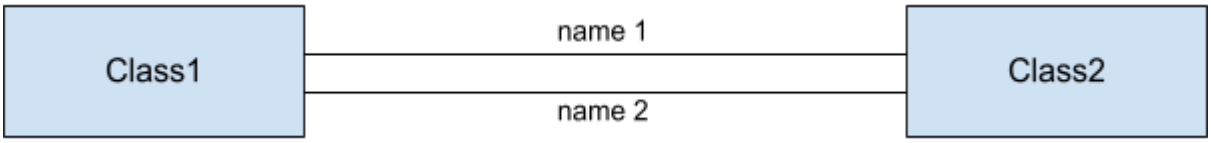
1.2. Associations

- Binary relationship between classes.



- **role1**: Each object of Class2 is related to at least *m* and at most *n* objects of Class1.
- **role2**: Each object of Class1 is related to at least *m* and at most *n* objects of Class2.

- There may be more than one association between the same pair of classes (having different names).



- Abbreviations:

*	is equivalent to 0..*; there's no restrictions on the number of objects
m..*	can have at least m objects; there's no maximum
	by default, the label is taken to be 1..1 (exactly one)

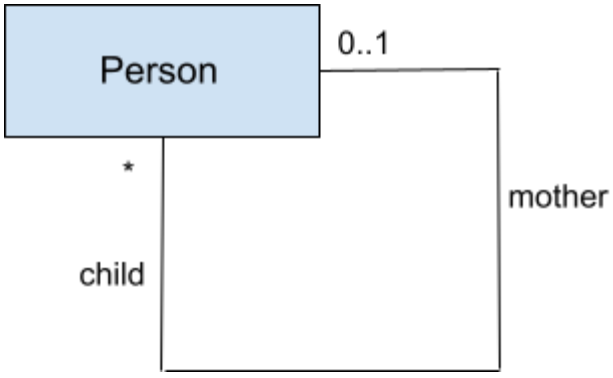
- Some multiplicities:

* - 0..1	many-to-one
0..1 - 0..1	one-to-one
* - *	many-to-many

- An association is complete when all objects participate in the association.
- If there's two classes with the same attributes, maybe the best is to define another class.

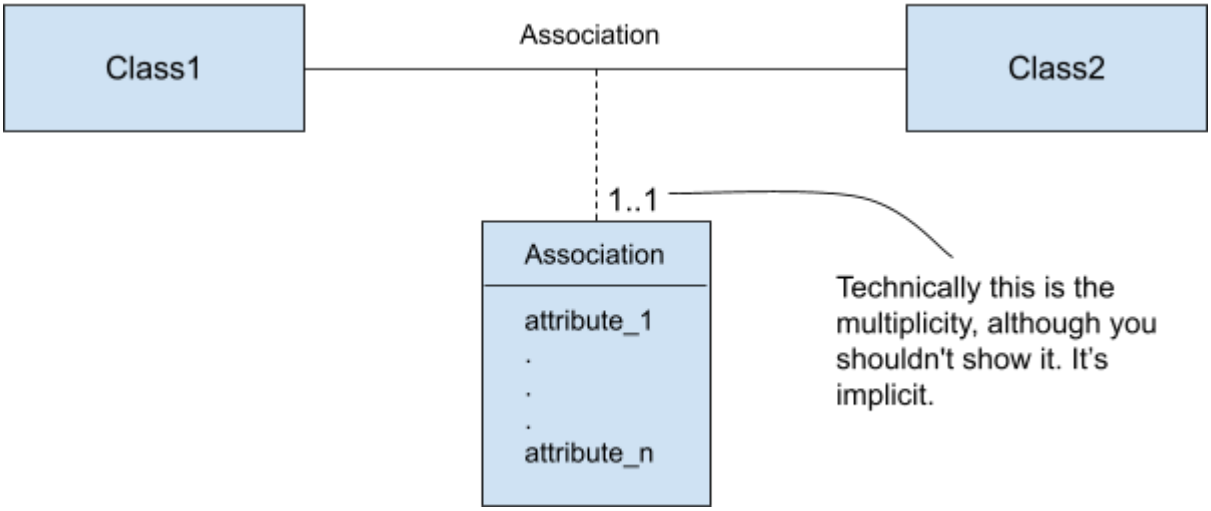
1.3. Self-association

- Associations that have both ends in the same class.
- Example:



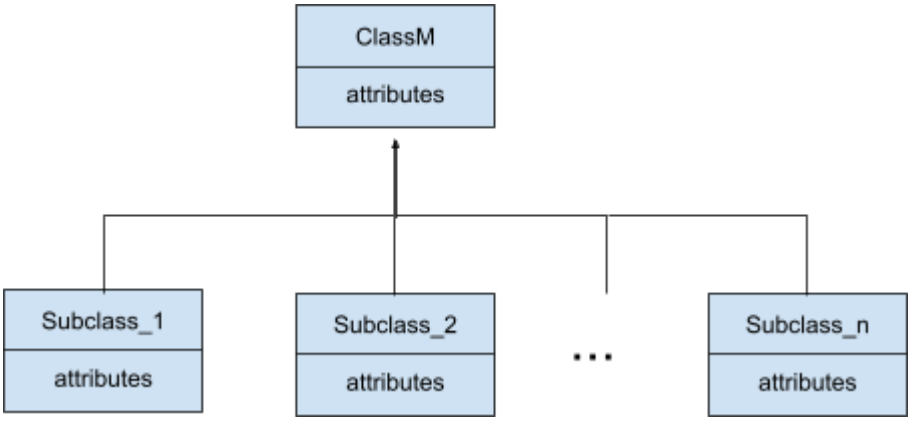
1.4. Associations classes

- Relationships between objects of two classes.
- If we want to eliminate an association class, we can do it when one of the roles is 0..1 or 1..1. The attributes from the association class goes to the many (*) side. (Isn't mandatory).
- Whenever there's two association classes, there also must be two separate associations (even if the multiplicity is equal).



1.5. Subclasses

- A subclass inherits the properties (attributes and associations) from its superclass.
- The subclass may have its own, additional, associations to other classes.
- *ClassM* is a **generalization** of its subclasses.
- Subclasses are **specializations** of *ClassM*.
- The subclasses can be:
    - **Complete**: an object must be in one of the subclasses
    - **Partial or Incomplete**: there may be objects that are not in neither subclasses
    - **Disjoint**: an object cannot be in two of the subclasses
    - **Overlapping:** an object can be in two or more of the subclasses



1.6. Aggregation and Composition

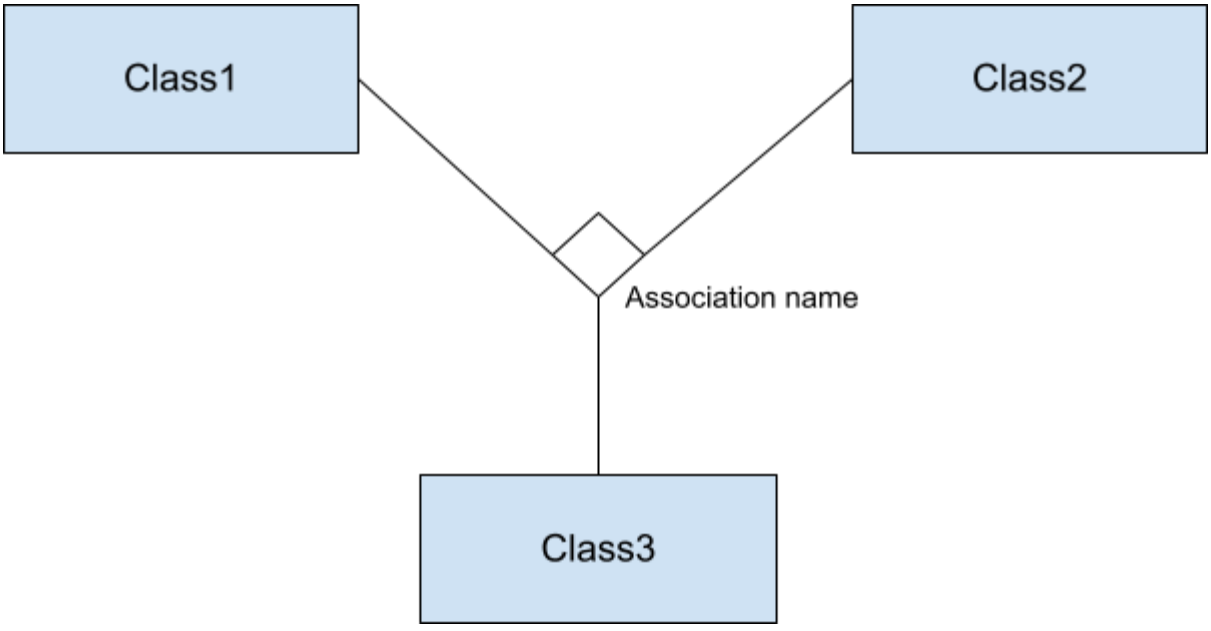
- Only applies to all/part models. In other words, models where you can use the sentence “Objects of *Class1* belong to objects of *Class2*”.

- **Aggregation**: Objects of *Class1* will contain a reference to a *Class2* object; that reference may be null.

- **Composition**: Objects of *Class1* will contain a reference to a *Class2* object; that reference cannot be null.

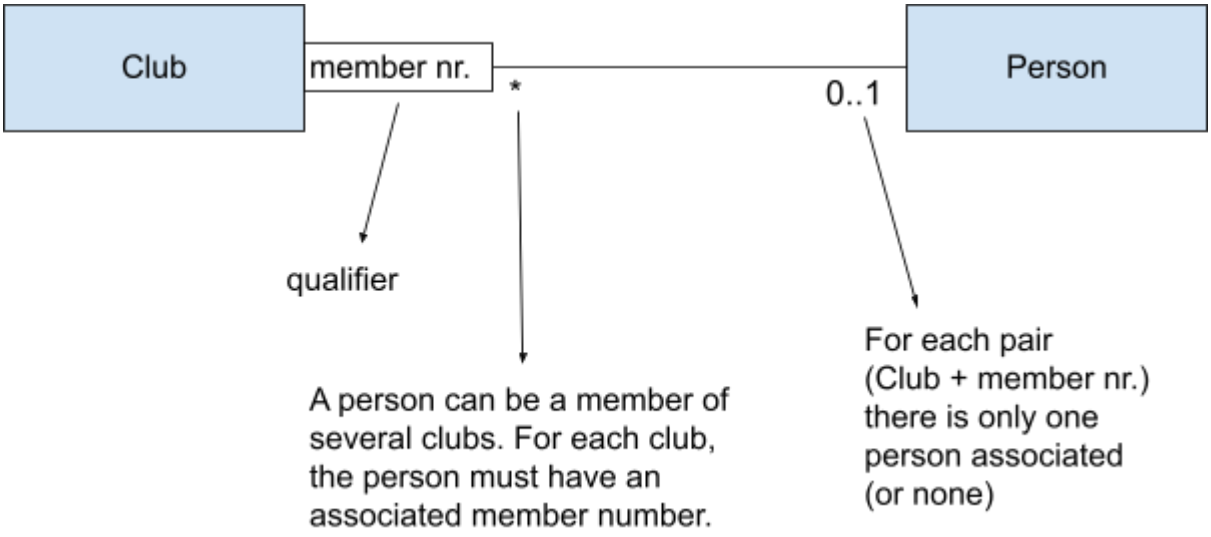
1.7. Associations n-arys

- Avoid this, because it complicates the model.
- **How to read it?** In pairs. *Class1* and *Class2* are associated with *Class3*. *Class2* and *Class3* are associated with *Class1*. And *Class1* and *Class3* are associated with *Class2*.



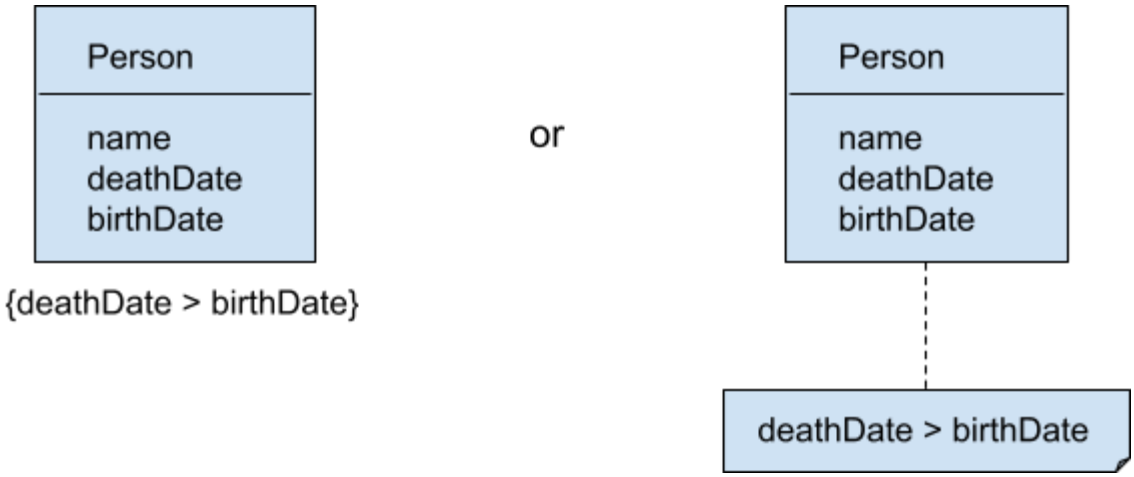
1.8. Qualified associations

- **Qualifier**: one or more attributes of an association used to navigate from it's associated class (*Club*) and the other class (*Person*).



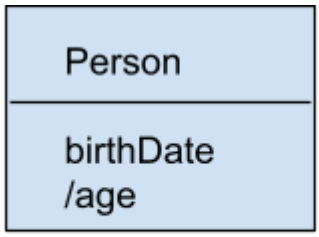
1.9. Constraints

- Specifies a condition that has to be present in the system.
- There are two ways to do this:



1.10. Derived elements

- Element (class, attribute or association) computed using other elements in the model.
- Derived elements are always redundant.
- ‘/’ before the name of the derived element.
- Usually have an associated constraint.



{age = now() - birthDate}

2. From UML to Relations

2.1. The relational model

- **Database**: set of named relations (or tables)

Table:

attribute_1	attribute_2	...	attribute_n
instance_1	instance_1	instance_1	instance_1
instance_2	instance_2	instance_2	instance_2
...	...	...	...
instance_m	instance_m	instance_m	instance_m

Relation:
`Relation_name(attribute_1, attribute_2, …, attribute_n)`

- **Attributes**: columns of the table; an attribute describes the meaning of the entries of each column (instances). Each attribute has a type or domain.

- **Tuples**: rows of the table; are the contents (instances); *instance_1* corresponds to a tuple, *instance_2* corresponds to another tuple.

- **Domain**: is the type of the attribute. In the relational model, each attribute must be atomic, that is, it must only have one type. They cannot be lists, arrays, or any other type that resembles sets.

- **Schemas**: is the name of the relation and it's set of attributes.

- **Arity**: of the relation is the number of attributes or the number of columns (*n*).

- **Cardinality**: of the relation is the number of tuples/instances or the number of rows (*m*).

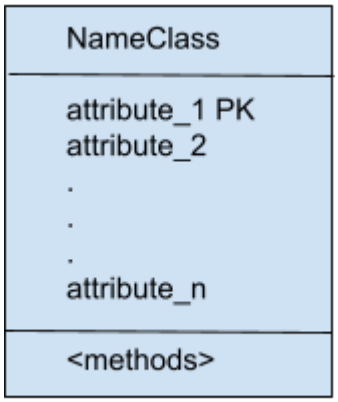
- **Key**: set of one or more attributes whose combined values are unique within a relation. Often denoted by underlying the set of key attributes. Each relation has only one key (that may be composed by just one attribute or more).

`Relation1 (attribute-1, attribute-2, …, attribute-n)`
or
`Relation2 (attribute-1, attribute-2, …, attribute-n)`

- **Foreign key**: an attribute (or set of attributes) that always matches a key attribute in another relation.

```
Relation3 (attribute-1, attribute-2, attribute-3 -> Relation1)
```

2.2. Classes




```
NameClass (attribute-1, attribute-2, …, attribute-n)
```

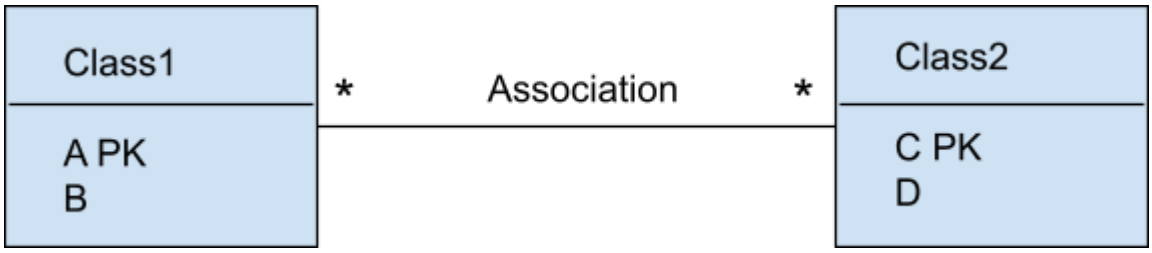
2.3. Associations

- The names of the attributes of the associations must be clear.

- **One-to-one**: add a foreign key to the relation that is expected to have less tuples.

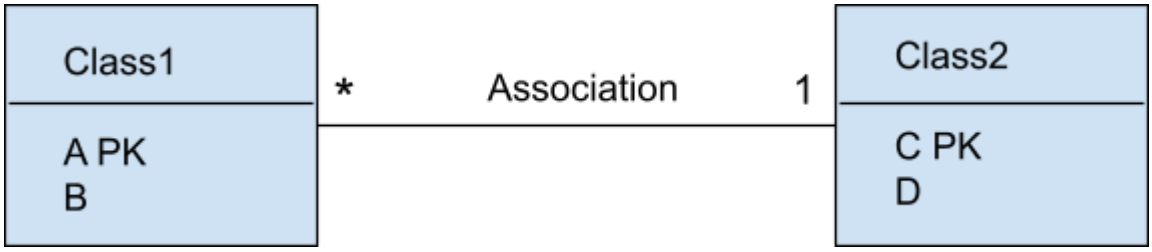

```
Class1 (A, B, class2-id -> Class2)
Class2 (C, D)
```

- **Many-to-many**: add a relation with the key from each side.




```
Class1 (A, B)
Class2 (C, D)
Association (class1-id -> Class1, class2-id -> Class2)
```

- **Many-to-one**:
  - add a foreign key to the many side of the relationship
    or
  - add a relation with the key from the many side. The *Class1* only relates to one object of *Class2*, so we just need the key from the many side to identify the relation.



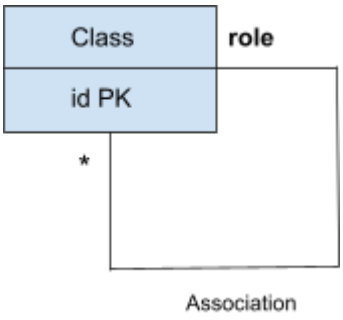

```
Class1 (A, B, class2-id -> Class2)
Class2 (C, D)
```

or

```
Class1 (A, B)
Class2 (C, D)
Association (class1-id -> Class1, class2-id -> Class2)
```

2.4. Self-association

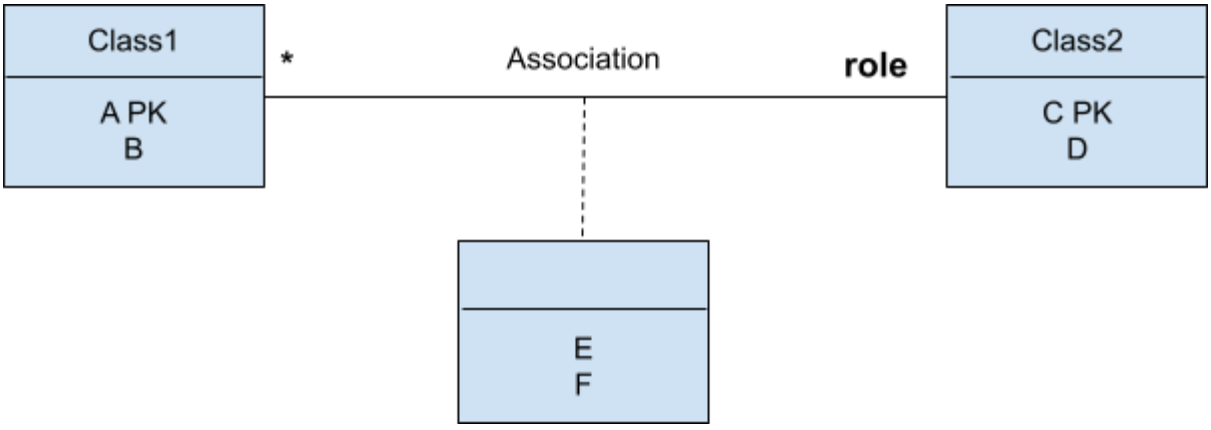
- The key is associated with the many side.




```
                Class (id)
        If role = * Association (id-1 -> Class, id-2 -> Class)
If role = 0..1 ou 1..1 Association (id-1 -> Class, id-2 -> Class)
```

2.5. Associations classes

- Add attributes to relation for the association.
- The key is always associated with the many side.



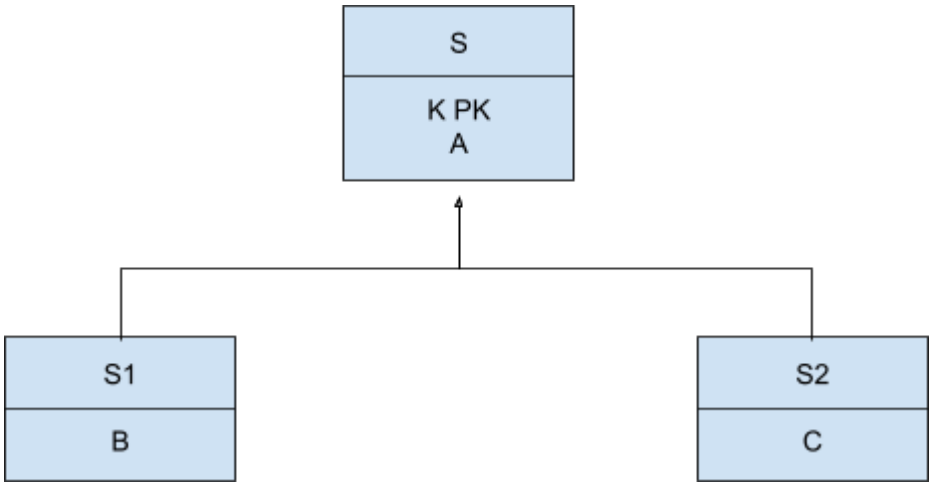

```
Class1 (A, B)
Class2 (C, D)
```

If role = * `Association (class1-id -> Class1, class2-id-> Class2, E, F)`

If role = 0..1 ou 1..1 `Association (class1-id -> Class1, class2-id-> Class2, E, F)`

2.6. Subclasses

- There are 3 strategies: E/R style, object-oriented and use nulls.



- **E/R style**: relation per each class; good for overlapping generalizations with a large number of subclasses.

```
S (K, A)
S1 (K -> S, B)
S2 (K -> S, C)
```

- **Object-oriented**: subclasse relations contain all attributes; good for disjoint generalizations and if superclass has few attributes and subclasses many attributes.

```
S (K, A)
S1 (K -> S, A, B)
S2 (K -> S, A, C)
```

  - ❏ In overlapping generalizations, we need to add relations for each combination.

```
S1S2 (K -> S, A, B, C)
```

  - ❏ In complete generalizations, we may eliminate the superclass (class *S*).

- **Use nulls**: one relation with all the attributes of all the classes; good for heavily overlapping generalizations with a small number of subclasses.

```
S (K, A, B, C)
```

  If the object only belongs to S2, then B will be null.

2.7. Aggregation and Composition

- Treat it as a regular association.

- **Aggregation**


```
Class1 (A, B, class2-id -> Class2)
Class2 (C, D)
```

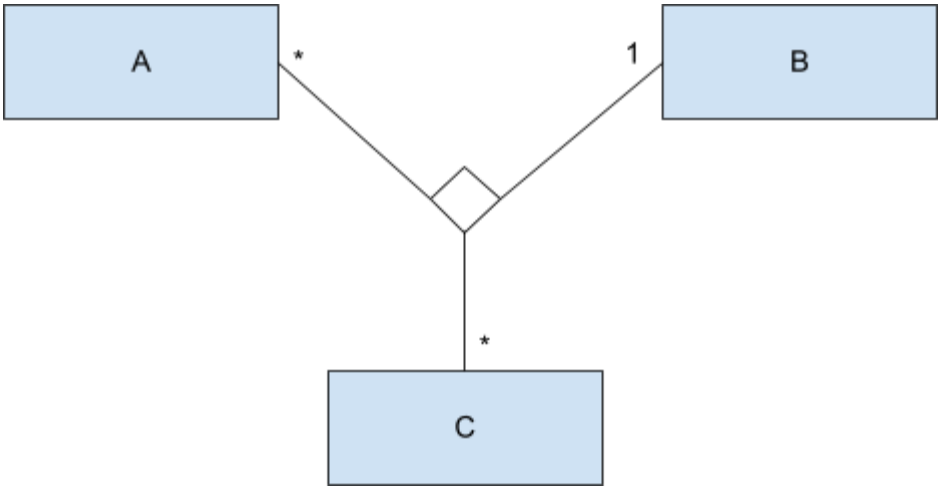
class2-id can be null

- **Composition**


```
Class1 (A, B, class2-id -> Class2)
Class2 (C, D)
```

2.8. Associations n-arys

- Always results in (n + 1) relations and is affected by the multiplicity.




```
A (id, …)
B (id, …)
C (id, …)
ABC (idA -> A, idB -> B, idC -> C)
```

2.9. Qualified associations


```
Club (clubID, ...)
Person (personID, ...)
Member (clubID -> Club, personID -> Person, member_nr)
     {clubID, personID} UK
```

3. Relational Design Theory

3.1. Overview

- **Design theory** is about how to represent your data to avoid anomalies.
- **Anomalies**:
  - **Redundancy**: information may be repeated unnecessarily in several tuples.
  - **Update**: we may change information in one tuple but leave the same information unchanged in another.
  - **Deletion**: If a set of values becomes empty, we may lose other information as a side effect.
- The accepted way to eliminate these anomalies is to decompose relations.

$$R(A_1,\ ...,\ A_n)$$
$$R_1(B_1,\ ...,\ B_n)$$
$$R_2(C_1,\ ...,\ C_n)$$

- **Decomposition of relations**: the replacement of one relation by several, whose sets of attributes together include all the attributes of the original.

- R_1 and R_2 are a decomposition of R if:
  - $\{B_1, B_2, ...,\ B_n\} \cup \{C_1, C_2, ...,\ C_n\} = \{A_1, A_2, ...,\ A_n\}$
  - $R_1 \bowtie R_2 = R$ ($\bowtie$: junction of all attributes)

- **Functional dependencies (FD)**
  - If two tuples of R agree on all of the attributes $A_1, A_2, ..., A_n$ (the tuples have the same values in their respective components for each of these attributes), then they must also agree on all of another list of attributes $B_1, B_2, ..., B_n$.

    $\{A_1, A_2, ...,\ A_n\}$ functionally determines $\{B_1, B_2, ...,\ B_n\}$.

  - If we can be sure every instance of a relation R will be one in which a given FD is true, then we say that R satisfies the FD.
  - When we say that R satisfies and FD f, we are asserting a constraint on R, not just saying something about one particular instance of R.

- **Key/Candidate key**: we say a set of one or more attributes ($A_1, A_2, ..., A_n$) is a key for a relation R if:
    - Those attributes functionally determine all other attributes of the relation.
    - No proper subset of $\{A_1, A_2, ..., A_n\}$ functionally determines all other attributes of R (a key must be **minimal**).

- **Primary key**: is one and only one of the candidate keys.

- **Superkey**: short for “superset of a key”, a set of attributes that contains a key.

- Rules about FD:
    - **Splitting rule**:

      $$\bar{A} \to B_1, B_2, \dots, B_n \Rightarrow \bar{A} \to B_1, \bar{A} \to B_2, \dots, \bar{A} \to B_n$$

      The contrary is not valid.

    - **Combining rule**:

      $$\bar{A} \to B_1, \bar{A} \to B_2, \dots, \bar{A} \to B_n \Rightarrow \bar{A} \to B_1, B_2, \dots, B_n$$

    - **Transitive rule**:

      $$\bar{A} \to \bar{B} \ \wedge \ \bar{B} \to \bar{C} \Rightarrow \bar{A} \to \bar{C}$$

    - **Trivial-dependency rule**:

      $\bar{A} \to \bar{B}$ is a trivial dependency if $\bar{B} \subseteq \bar{A}$

    - **Nontrivial dependency**:

      $\bar{A} \to \bar{B}$ is a nontrivial dependency if $\bar{B} \not\subseteq \bar{A}$

    - **Completely nontrivial dependency**:

      $\bar{A} \to \bar{B}$ is a completely nontrivial dependency if $\bar{A} \cap \bar{B} = \emptyset$

3.2. Closure of attributes

- $\{A_1, ..., A_n\}^+$ is the closure of $\{A_1, ..., A_n\}$
- If $\{A_1, ..., A_n\}^+$ contains all attributes of R, then $\{A_1, ..., A_n\}$ is a key.
- **Goal**: find the set of attributes functionally determined by $\{A_1, ..., A_n\}^+$
- **Algorithm**:
  1. Start with $\{A_1, ..., A_n\}$
  2. Repeat until no change:
     2.1. If $\{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_n\}$ and $\{A_1, ..., A_n\}$ in set, add $\{B_1, ..., B_n\}$ to set.

- Example:
  Given the relation $R(A, B, C, D, E, F)$ and the following functional dependencies:
  (1) $\{A, B\} \rightarrow \{C\}$
  (2) $\{A, D\} \rightarrow \{E\}$
  (3) $\{B\} \rightarrow \{D\}$
  (4) $\{A, F\} \rightarrow \{B\}$
  (a) Compute $\{A, B\}^+$
  (b) Compute $\{A, F\}^+$

(a) $\{A, B\}^+ = \{A, B\}$, including its own attributes $\{A, B, C, D\}$, using FD's (1) and (3) $\{A, B, C, D, E\}$, using FD (2) We can't use FD (4) because F isn't in the set.
(b) $\{A, F\}^+ = \{A, F\}$, including its own attributes $\{A, F, B\}$, using FD (4) $\{A, F, B, C, D\}$, using FD's (1) and (3) $\{A, F, B, C, D, E\}$, using FD (2) $\{A, F\}$ is a key for R, because its closure includes all attributes of R

3.3. Finding keys

- If $\{A_1, ..., A_n\}^+$ contains all attributes of R, then $\{A_1, ..., A_n\}$ is a key.
- **Algorithm**: Consider every subset of attributes and compute it's closure to see if it determines all attributes. Start with single attributes, then go to pairs and so on.

- Example:
  Find the keys given $R(A, B, C, D)$ and $\{A, B\} \rightarrow \{C\}$, $\{A, D\} \rightarrow \{B\}$, $\{B\} \rightarrow \{D\}$.

single attributes:

$$\{A\}^+ = \{A\}$$
$$\{B\}^+ = \{B, D\}$$
$$\{C\}^+ = \{C\}$$
$$\{D\}^+ = \{D\}$$

pairs of attributes:

$$\{A, B\}^+ = \{A, B, C, D\}$$
$$\{A, C\}^+ = \{A, C\}$$
$$\{A, D\}^+ = \{A, B, C, D\}$$
$$\{B, C\}^+ = \{B, C, D\}$$
$$\{B, D\}^+ = \{B, D\}$$
$$\{C, D\}^+ = \{C, D\}$$

three attributes:

$$\{A, B, C\}^+ = \{A, B, C, D\}$$
$$\{A, B, D\}^+ = \{A, B, C, D\}$$
$$\{A, C, D\}^+ = \{A, B, C, D\}$$
$$\{B, C, D\}^+ = \{B, C, D\}$$

four attributes:

$$\{A, B, C, D\}^+ = \{A, B, C, D\}$$

Superkeys:

- $\{A, B\}$ -> key (must be minimal)
- $\{A, D\}$ -> key (must be minimal)
- $\{A, B, C\}$
- $\{A, B, D\}$
- $\{A, C, D\}$
- $\{A, B, C, D\}$

3.4. Inferring FDs

- **Goal**: inferring all FDs given a set of (incomplete) FDs
- **Algorithm**:
  1. Compute X^+, for every set of attributes X
  2. Enumerate all FDs $X \rightarrow Y$ such that:
     - $Y \subseteq X^+$ and
     - $X \cap Y = \emptyset$
- Example: Infer all FDs given $R(A, B, C, D)$ and $\{A, B\} \rightarrow \{C\}$, $\{A, D\} \rightarrow \{B\}$, $\{B\} \rightarrow \{D\}$.

X^+, for every set of attributes X	all FDs $X \rightarrow Y$ such that $Y \subseteq X^+$ and $X \cap Y = \emptyset$
$\{A\}^+ = \{A\}$	-
$\{B\}^+ = \{B, D\}$	$\{B\} \rightarrow \{D\}$
$\{C\}^+ = \{C\}$	-
$\{D\}^+ = \{D\}$	-
$\{A, B\}^+ = \{A, B, C, D\}$	$\{A, B\} \rightarrow \{C, D\}$
$\{A, C\}^+ = \{A, C\}$	-
$\{A, D\}^+ = \{A, B, C, D\}$	$\{A, D\} \rightarrow \{B, C\}$
$\{B, C\}^+ = \{B, C, D\}$	$\{B, C\} \rightarrow \{D\}$
$\{B, D\}^+ = \{B, D\}$	-
$\{C, D\}^+ = \{C, D\}$	-
$\{A, B, C\}^+ = \{A, B, C, D\}$	$\{A, B, C\} \rightarrow \{D\}$
$\{A, B, D\}^+ = \{A, B, C, D\}$	$\{A, B, D\} \rightarrow \{C\}$
$\{A, C, D\}^+ = \{A, B, C, D\}$	$\{A, C, D\} \rightarrow \{B\}$
$\{B, C, D\}^+ = \{B, C, D\}$	-
$\{A, B, C, D\}^+ = \{A, B, C, D\}$	-

3.5. Projecting functional dependencies

- We have a relation R with set of FDs S, and we project R by computing $R_1 = \pi_L(R)$, for some list of attributes R. What FDs hold in R_1?
- **Algorithm**:
  - Let T be the eventual output set of FDs. Initially T is empty.
  - For each set of attributes X that is a subset of the attributes of R_1, compute X^+. This computation is performed with respect to the set of FDs S, and may involve attributes that are in the schema of R but not in R_1. Add to T all nontrivial FDs $X \rightarrow Y$ such that Y is both in X^+ and an attribute of R_1.
  - Now, T is a basis for the FDs that hold in R_1, but may not be a minimal basis. We may construct a minimal basis by modifying T as follows:
    - If there is an FD F in T that follows from the other FDs in T, remove F from T.
    - Let $Y \rightarrow B$ be an FD in T, with at least two attributes in Y, and let Z be Y with one of its attributes removed. If $Z \rightarrow B$ follows from the FDs in T (including $Y \rightarrow B$), then replace $Y \rightarrow B$ by $Z \rightarrow B$.
    - Repeat the above steps in all possible ways until no more changes to T can be made.

- Example:
  Given $R(A, B, C, D)$ and the following functional dependencies:

  $A \rightarrow B$

  $B \rightarrow C$

  $C \rightarrow D$

  $R_1(A, C, D)$ was projected.

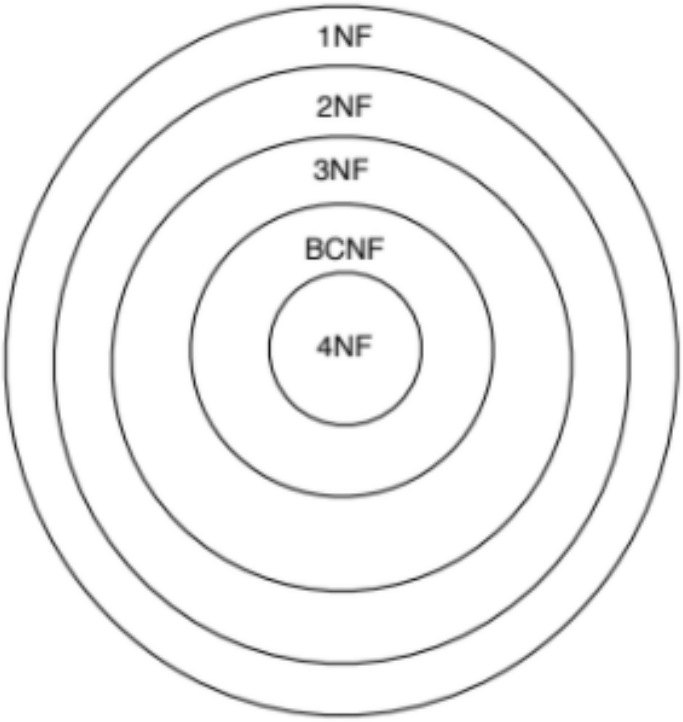
  What are the functional dependencies for R_1?

X^+, for every set of attributes X of R_1	T: all FDs $X \rightarrow Y$ such that $Y \subseteq X^+$, Y is an attribute of R_1 and $Y \notin X$
$\{A\}^+ = \{A, B, C, D\}$	$\{A\} \rightarrow \{C, D\}$
$\{C\}^+ = \{C, D\}$	$\{C\} \rightarrow \{D\}$
$\{D\}^+ = \{D\}$	-
$\{A, C\}^+ = \{A, C, B, D\}$	$\{A, C\} \rightarrow \{D\}$
$\{A, D\}^+ = \{A, B, C, D\}$	$\{A, D\} \rightarrow \{C\}$
$\{C, D\}^+ = \{C, D\}$	-
$\{A, C, D\}^+ = \{A, B, C, D\}$	-

T	construct a minimal basis by modifying T
(1) $\{A\} \rightarrow \{C, D\}$	(5) $\{A\} \rightarrow \{C\}$ e (6) $\{A\} \rightarrow \{D\}$
(2) $\{C\} \rightarrow \{D\}$	-
(3) $\{A, C\} \rightarrow \{D\}$	$\{A\} \rightarrow \{D\}$ follows from (6) $\{C\} \rightarrow \{D\}$ follows from (2)
(4) $\{A, D\} \rightarrow \{C\}$	$\{A\} \rightarrow \{C\}$ follows from (5) $\{D\} \rightarrow \{C\}$ doesn't follow anything
(5) $\{A\} \rightarrow \{C\}$	-
(6) $\{A\} \rightarrow \{D\}$	we can eliminate this FD, because it follows from (5) and (2) by the transitive rule

Answer: The FDs for R_1 are T : $[\{A\} \rightarrow \{C\}, \{C\} \rightarrow \{D\}]$

3.6. Normal forms



3.6.1. First normal form (1NF)

- A relation is in first normal form if every attribute in that relation is a single valued attribute.
- Example:

Student	Courses
Mary	{CS145, CS229}
Joe	{CS145, CS106}

violates 1NF

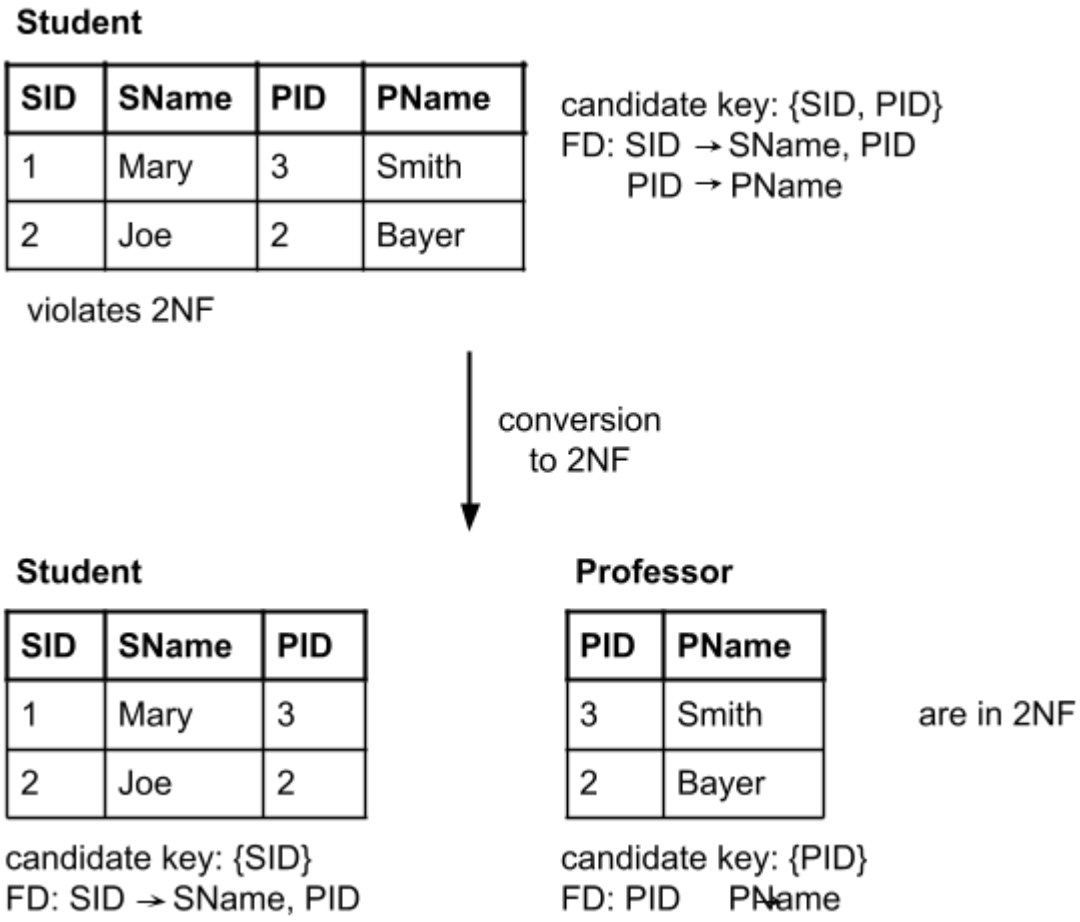
conversion to 1NF →

Student	Course
Mary	CS145
Mary	CS229
Joe	CS145
Joe	CS106

is in 1NF

3.6.2. Second normal form (2NF)

- To be in 2NF, a relation must be in 1NF and a relation must not contain any partial dependency.
- **Partial dependency**: proper subset of candidate key that determines non-prime attributes (attributes that don't belong to any candidate key).
- Example:



3.6.3. Third normal form (3NF)

- A relation is in 3NF if at least one of the following conditions holds in every non-trivial function dependency $X \rightarrow Y$:
    1. X is a superkey
    2. Y is a prime attribute (each element of Y is part of some candidate key).

- Example:
  Given $R(A,\ B,\ C,\ D,\ E)$ and the following functional dependencies:
  $A \rightarrow BC$
  $CD \rightarrow E$
  $B \rightarrow D$
  $E \rightarrow A$
  The candidate keys for R are: $\{A,\ E,\ CD,\ BC\}$
  Is the relation R in 3NF?

$A \rightarrow BC$	Respects condition 1 and condition 2.
$CD \rightarrow E$	Respects condition 1 and condition 2.
$B \rightarrow D$	Violates condition 1, but respects condition 2.
$E \rightarrow A$	Respects condition 1 and condition 2.
Therefore, it is in 3NF.	

3.6.4 Boyce-Codd normal form (BCNF)

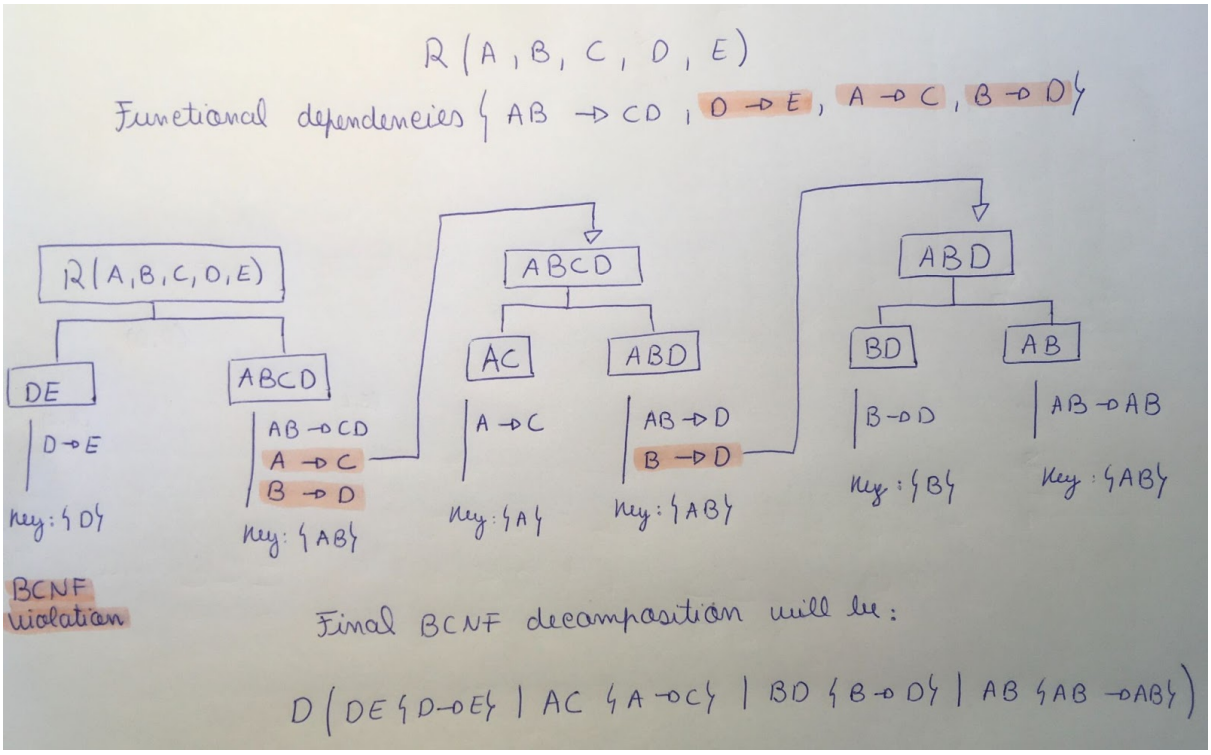
- A relation is in BCNF if the left side of every nontrivial FD is a superkey (contains a key).
- Example:
  Given $R(A,\ B,\ C,\ D,\ E,\ F,\ G,\ H)$ and the following functional dependencies:
  $A \rightarrow BCG$
  $D \rightarrow EF$
  The key is $\{A,\ D\}$. Is the relation in BCNF?

It is not in BCNF, because none of the FDs contains a key on the left side.

3.7. BCNF decomposition

- Assures lossless joins, but dependency preservation is not always possible
- We want a BCNF decomposition of the relation R with set F of FDs for R.
- **Algorithm**:
  1. Compute keys for R
  2. Repeat until all relations are in BCNF:
     - 2.1. Pick any R' with $A \rightarrow B$ that violates BCNF.
     - 2.2. Decompose R' into $R_1(A, B)$ and $R_2(A, rest)$.
     - 2.3. Compute FDs for R_1 and R_2.
     - 2.4. Compute keys for R_1 and R_2.

- Example:



3.8. 3NF decomposition

- Assures lossless joins and dependency preservation.
- We want a 3NF decomposition of the relation R with set F of FDs for R.
- **Algorithm**:
  1. Find a minimal basis for F, say G
  2. For each FD $X \rightarrow A$ in G, create a relation $R'(X, A)$
  3. If none of the relations of step 2 is a superkey for R, add another relation for a key for R
- Example:
  Given $R(A, B, C, D, E)$ and the following functional dependencies:

  $AB \rightarrow C$

  $C \rightarrow B$

  $A \rightarrow D$

  Decompose R to 3NF.

1. Set of FDs is already a minimal basis
2. $R_1(A, B, C)$
   $R_2(C, B)$
   $R_3(A, D)$
3. R_1 is superkey?
   $\{A, B, C\}^+ = \{A, B, C, D\}$
   No, R_1 is not a superkey.

   R_2 is superkey?
   $\{C, B\}^+ = \{C, B\}$
   No, R_2 is not a superkey.

   R_3 is superkey?
   $\{A, D\}^+ = \{A, D\}$
   No, R_3 is not a superkey.

Therefore, we have to create $R_4(A, B, E)$, $\{A, B, E\}$ is a key for R.

Answer: $R_1(A, B, C)$, $R_2(C, B)$, $R_3(A, D)$ and $R_4(A, B, E)$.

3.9. The chase test for lossless join

- Given a relation and its decomposition, the chase test for lossless join determines if the decomposition ensures lossless joins, thus determining if the decomposition loses information.
- **Algorithm**:
  1. Build the tableau
     - ❏ One line per decomposed relation
     - ❏ A letter per each attribute in the decomposed relation
     - ❏ Subscript the letter with i, if the attribute is not in S_i
  2. For each functional dependency $X \rightarrow Y$, see if the tableau agrees with it, for the same value of X, the Y must be the same. If not, modify it in order to agree. The goal is decrease i.
  3. If the final table has a line without subscripts (i), it is a lossless join.

- Example:
  $S(A, B, C, D)$ was decomposed in $S_1(A, D)$, $S_2(A, C)$ and $S_3(B, C, D)$.
  S has the following functional dependencies: $A \rightarrow B$, $A \rightarrow C$, $CD \rightarrow A$.
  Does this decomposition ensure lossless joins?

(1.)

A	B	C	D

A	B	C	D
a			d
a		c	
	b	c	d

A	B	C	D	
a	b_1	c_1	d	S1 (A, D)
a	b_2	c	d_2	S2 (A, C)
a_3	b	c	d	S3 (B, C, D)

(2.)

A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

A->B

A	B	C	D
a	b_1	c_1	d
a	b_1	c	d_2
a_3	b	c	d

Iterations

A->B tells us that the first two rows must agree in the B attribute, that is, b_1=b_2
From B->C, we know that c_1=c
From CD->A, we know that a=a_3

B->C

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a_3	b	c	d

CD->A

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

(3.)

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

Conclusion: The decomposition ensures lossless joins.

4. SQL - Data Definition Language

4.1. SQL is a...

- **Data definition language:** for declaring database schemas.
- **Data manipulation language:** for querying databases and for modifying the database.

4.2. Types of relations

- **Tables**: are stored in disk.
- **Views**: never stored; are computed at the moment; useful, for example, to control the access to the information.
- **Temporary tables**: we store the tables at that session, but delete them when we close the session; efficient because we are just querying a set of tuples instead of all tuples.

4.3. Data types in SQL and SQLite

- **CHAR(n)**: fixed-length string of up to *n* characters; short strings are padded by trailing blanks to make *n* characters.
    - SQLite: TEXT

- **VARCHAR(n)**: string of up to *n* characters; an endmarker is used.
    - SQLite: TEXT

- **BOOLEAN**: possible values are TRUE, FALSE and UNKNOWN.
    - SQLite: stored as integers
        - INT 0 (false)
        - INT 1 (true)

- **INT/INTEGER**: typical integer values.
    - SQLite: INTEGER

- **SHORTINT**: also integer, but the number of bits permitted may be less
    - SQLite: INTEGER

- **FLOAT/REAL**: typical floating-point numbers
    - SQLite: REAL

- **DECIMAL(n, d)**: values that consist of *n* decimal digits, with the decimal point assumed to be *d* positions from the right.
    - SQLite: REAL

- **DATE**: quoted string of a special form ‘0000-00-00’ (year-month-day)
    - SQLite:
        - TEXT “YYYY-MM-DD”
        - REAL as Julian day numbers “4744 B.C.”
        - INTEGER YYYY-MM-DD
        - built-in date functions to convert between formats

- **TIME**: quoted string of a special form ‘00:00:00.0’ (hours:minutes:seconds.fractions of a second)
    - SQLite:
        - TEXT “HH:MM:SS:SSS”
        - INTEGER 00:00:00
        - built-in time functions to convert between formats

- **BLOB - SQLite**: the value is a binary large object of data, stored exactly as it was input.

4.4. Type affinities

The **type affinity** is the recommended type for data stored, in case we want to convert from SQLite to another database engine.

There are 5 types of affinities:

- TEXT
- NUMERIC
- INTEGER
- REAL
- BLOB

Rules should be assessed by this order	Declared type in SQLite	Type affinity in case we want to convert to other database engine
1	INT	INTEGER
2	CHAR, CLOB, TEXT	TEXT
3	BLOB or no type specified at all	BLOB
4	REAL, FLOA, DOUB	REAL
5	otherwise (if unrecognizable type specified)	NUMERIC

4.5. Table declaration

```
DROP TABLE IF EXISTS <table_name>;
CREATE TABLE <table_name>
(
    <column_name><data_type>,
    <column_name><data_type>,
    ...
    <column_name><data_type>
);
```

4.6. Modifying schemas

- To remove an entire table and all its tuples:

```
DROP TABLE <table_name>;
```

- To modify the schema of an existing relation, adding and/or dropping attributes/columns:

```
ALTER TABLE <table_name> ADD <column_name><data_type>;
```

```
ALTER TABLE <table_name> DROP <column_name>;
```

4.7. Default values

- For each column we can define its default value. That value is either NULL or a constant.
- We can define a default value either in a table declaration or by modifying schemas.

```
CREATE TABLE <table_name>
(
    <column_name><data_type> DEFAULT <default_value>,
    ...
    <column_name><data_type>
);
```

4.8. Constraints

4.8.1. Non-null constraint

- Defines that a column cannot have NULL values

```
CREATE TABLE <table_name>
(
    ...
    <column_name><data_type> NOT NULL,
    ...
);
```

4.8.2. Primary key constraint

- NULL values are not allowed and we can define one, and only one primary key for a table
- Two types of declaration:
    - one attribute is the key

```
CREATE TABLE <table_name>
(
    <column_name><data_type> PRIMARY KEY,
    ...
    <column_name><data_type>
);
```

    - set of attributes form a key

```
CREATE TABLE <table_name>
(
    <column_name_1><data_type>,
    ...
    <column_name_2><data_type>,
    PRIMARY KEY(<column_name_1>, <column_name_2>)
);
```

4.8.3. Unique key constraint

- NULL values are allowed
- Two types of declaration:
  - one attribute is the key

```
CREATE TABLE <table_name>
(
    <column_name><data_type> UNIQUE,
    ...
    <column_name><data_type>
);
```

  - set of attributes form a key

```
CREATE TABLE <table_name>
(
    <column_name_1><data_type>,
    ...
    <column_name_2><data_type>,
    UNIQUE(<column_name_1>, <column_name_2>)
);
```

4.8.4. Check constraint

- Checked whenever we insert or update a tuple.
- Two types of check constraints:
  - **attribute-based constraint**: check one attribute

```
CREATE TABLE <table_name>
(
    <column_name><data_type> CHECK <expression>,
    ...
    <column_name><data_type>
);
```

  - **tuple-based constraint**: various attributes

```
CREATE TABLE <table_name>
(
    <column_name_1><data_type>,
    ...
    <column_name_2><data_type>
    CHECK(<expression involving more than one column>)
);
```

- It is best practice to name all constraints.

```
CREATE TABLE <table_name>
(
    <column_name><data_type> CONSTRAINT <constraint_name><expression>,
    ...
);
```

- It is possible to modify constraints.

```
ALTER TABLE <table_name> ADD CONSTRAINT <constraint_name><expression>;

ALTER TABLE <table_name> DROP CONSTRAINT <constraint_name>;
```

4.8.5. Foreign-key constraint

- we can declare an attribute or various attributes of one relation to be a foreign key, referencing some attribute(s) of a second relation.
- foreign key constraints are disabled by default. To enable:

```
PRAGMA foreign_keys = ON;
```

- Two ways of declaring a foreign key:
    - one attribute is a foreign-key

```
CREATE TABLE <table_A>
(
    <column_A><data_type> PRIMARY KEY,
    <column_B><data_type>,
    ...
    <column_C><data_type>
);
CREATE TABLE <table_B>
(
    <column_X><data_type> PRIMARY KEY,
    <column_Y><data_type>,
    ...
    <column_Z><data_type> REFERENCES <table_a>(<column_A>)
);
```

We can omit this (red) if the referenced column is the primary key

- set of attributes are foreign-keys

```
CREATE TABLE <table_A>
(
    <column_A><data_type> PRIMARY KEY,
    ...
    <column_B><data_type>
);
CREATE TABLE <table_B>
(
    <column_X><data_type> PRIMARY KEY,
    ...
    <column_Y><data_type>,
    FOREIGN KEY(<column_X>, <column_Y>) REFERENCES <table_A>
(<column_A>, <column_B>)
);
```

4.9. Referential integrity

```
CREATE TABLE <table_B>
(
    <column_X><data_type> PRIMARY KEY,
    <column_Y><data_type>,
    ...
    <column_Z><data_type> REFERENCES <table_a>(<column_A>)
        ON DELETE <action>
        ON UPDATE <action>
);
```

- A table (the referencing table) can refer to a column (or columns) in another table (the referenced table) by using a foreign key.
- On inserting (or deleting) a row into the referencing table, the DBMS checks if the entered key value exists in the referenced table. Performs the update (or removal) according to the action specified after ON UPDATE/ON DELETE.
- Actions:
    - **RESTRICT**: default, prohibit operation on a parent key when there are child keys mapped to it.
    - **CASCADE**: whenever rows in the parent table are deleted (or updated), the respective rows of the child table with a matching foreign key column will be deleted (or updated) as well.
    - **SET NULL**: the value of the affected child attribute is changed to NULL.
    - **SET DEFAULT**: child key columns are set to the default value.

5. Relational Algebra

- Formal language that operates on relations and produces relations as a result.
- When a DBMS (Database Management System) processes queries, the first thing that happens to a SQL query is that it gets translated into relational algebra or a very similar internal representation.
- **Queries**: expression of relational algebra.
- Relational Algebra eliminates duplicates; SQL does not eliminate duplicates.
- (PPR-USD) **Projection**, **product**, **renaming**, **union**, **selection** and **difference** - form an independent set, none of which can be written in terms of the other five.
- **Aggregation operators**: cnt, sum, avg, max, min

5.1. Selection

$$\sigma_{condition}(Relation)$$

- Returns all tuples from Relation which satisfy a condition.
- Operators to use in condition: $=, <, \leq, >, \geq, <>$ (diferente)

5.2. Projection

$$\pi_{A_1, A_2, ..., A_n}(Relation)$$

- Picks only the columns A_1, A_2, ..., A_n of $Relation$.

5.3. Cartesian product or Cross-product

$$R \; x \; S$$

- Returns one tuple for every combination of tuples from R and S relations.

- Example:

Relation *R*

A	B
1	2
3	4

Relation *S*

B	C	D
2	5	6
4	7	8
9	10	11

Relation *R x S*

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

- Notation to disambiguate attribute B:
  - **R.B** → attribute B from relation R
  - **S.B** → attribute B from relation S

5.4. Natural join

$$R \bowtie S$$

- $R \bowtie S = \pi_{R,S}(\sigma_C(R \; x \; S))$
- We pair only those tuples from R and S that agree in whatever attributes are common to the schemas of R and S.
- Basically, is a cross-product enforcing equality on all attributes with the same name.

- Example:

Relation R

A	B
1	2
3	4

Relation S

B	C	D
2	5	6
4	7	8
9	10	11

Relation $R \bowtie S$

A	B	C	D
1	2	5	6
3	4	7	8

Dangling tuple: a tuple that fails to pair with any tuple of the other relation

5.5. Theta-join

$$R \bowtie_C S$$

- $R \bowtie_C S = \sigma_C (R \times S)$

1. Take the product of R and S
2. Select from the product only those tuples that satisfy the condition C.

5.6. Semijoin

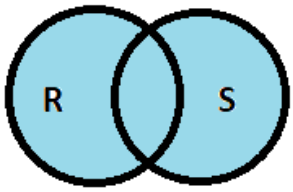
$$R \ltimes S$$

- $R \ltimes S = \pi_{A_1, ..., A_n}(R \bowtie S)$, where A_1, ..., A_n are attributes in R.
- Returns the tuples in R for which there is a tuple in S that is equal to their common attribute names.

5.7. Union

$$R \cup S$$

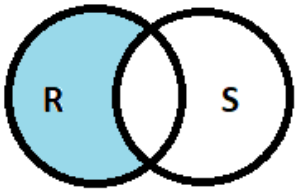
- An element appears only once in the union even if it is present in both R and S, because Relational Algebra eliminates duplicates.



5.8. Difference

$$R - S$$

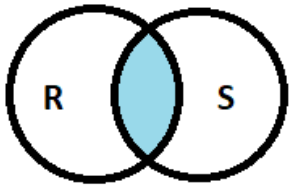
- $R - S \neq S - R$
- Set of elements that are in R but not in S.



5.9. Intersection

$$R \cap S$$

- $R \cap S = R - (R - S) = R \bowtie S$



5.10. Division

$$R / S$$

- $R\,(a_1,\ ...,\ a_n,\ b_1,\ ...,\ b_n)$
- $S\,(b_1,\ ...,\ b_m)$
- $R / S = \pi_{a_1,\ ...,\ a_n}(R) - \pi_{a_1,\ ...,\ a_n}[(\pi_{a_1,...,\ a_n}(R)\ x\ S) - R]$
- Algorithm:
    1. Reorder the columns in R so the last ones are the ones in S.
    2. Order R by the first columns
    3. Each R sub-tuple is part of the result if the sub-tuple of the last columns contains the divisor (S).

- Example:

R

A	C	B	D
a	c	b	d
a	e	b	f
b	e	c	f
e	c	d	d
e	e	d	f
a	d	b	e

/

S

C	D
c	d
e	f

= ?

1. Reorder the columns in R so the last ones are the ones in S.

R

A	B	C	D
a	b	c	d
a	b	e	f
b	c	e	f
e	d	c	d
e	d	e	f
a	b	d	e

/

S

C	D
c	d
e	f

= ?

2. Order R by the first columns

R

A	B	C	D
a	b	c	d
a	b	e	f
a	b	d	e
b	c	e	f
e	d	c	d
e	d	e	f

/

S

C	D
c	d
e	f

= ?

3. Each R sub-tuple is part of the result if the sub-tuple of the last columns contains the divisor (S).

R

A	B	C	D
a	b	c	d
a	b	e	f
a	b	d	e
b	c	e	f
e	d	c	d
e	d	e	f

/

S

C	D
c	d
e	f

=

A	B
a	b
e	d

5.11. Renaming

$$\rho_{R(A_1, ..., A_n)}(E)$$

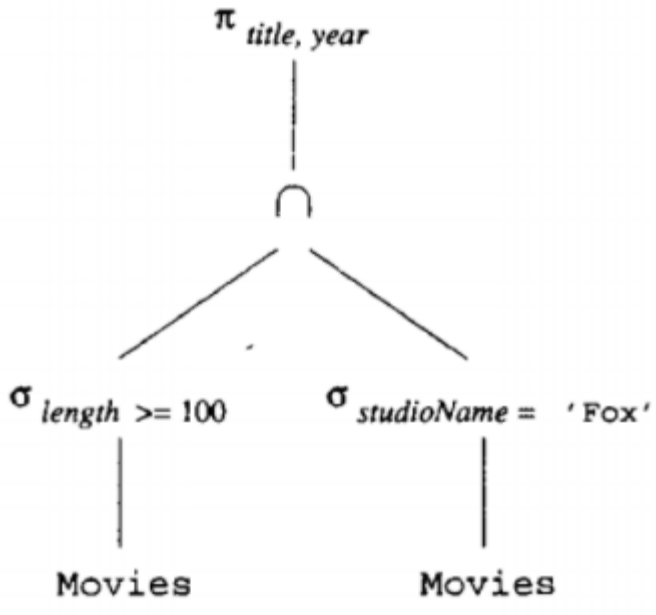
- $R(A_1, ..., A_n) \rightarrow$ new changed attributes names and/or relation name
- $E \rightarrow$ old relation name

- changes the schema, not the instance
- to only change the relation name: $\rho_R(E)$
- to only change attribute names: $\rho_{A_1, ..., A_n}(E)$

5.12. Expression trees

- Are evaluated bottom-up by applying the operator at an interior node to the arguments, which are the results of its children.
- Example:

$$\pi_{title,year}\left(\sigma_{length \geq 100}(\texttt{Movies}) \cap \sigma_{studioName='\texttt{Fox}'}(\texttt{Movies})\right)$$



6. SQL - Data Manipulation Language

- When a query is run over relations, the result is a relation.

6.1. Simple query

```
SELECT X
FROM Y
WHERE C;
```

- **SELECT**: tells which attributes of the tuples matching the condition are produced as part of the answer
    - If X = *, then the entire tuple is produced (all attributes)
- **FROM**: gives the relation or relations to which the query refers
- **WHERE**: is a condition. Tuples must satisfy the condition in order to match the query.

6.2. Projection

```
SELECT attribute_1, attribute_2, ... attribute_n
FROM Y
WHERE C;
```

- We can project the relation produced by a SQL query onto some of its attributes, listing some of the attributes of the relation Y.

6.3. Rename attributes

```
SELECT attribute_1 AS attr1, attribute_2 AS attr2, ...
FROM Y
WHERE C;
```

- **AS:** to produce a relation with column headers different from the attribute of the relation Y.
- We can, in the SELECT clause, use an expression in place of attribute. example:

```
SELECT num*2 AS doubleNumber
```

6.4. Rename tables

```
SELECT X
FROM aLetter AS A, letterB AS B
WHERE C;
```

6.5. Distinct

```
SELECT DISTINCT attribute_1, attribute_2
FROM Y
WHERE C;
```

- Used to return only distinct (different) values. So, attribute_1 won't have duplicate values. But, attribute_2 can have duplicate values, because it does not have the DISTINCT keyword associated.

6.6. Condition C

- Six common comparison operators: $=$, $<>$ (different), $<$, $>$, $<=$, $>=$.
- The values that may be compared include constants and attributes of the relations Y.
- We may also apply the usual arithmetic operators ($+$, $*$, …) to numeric values before we compare them.
- We may apply the concatenation operator || to strings.
- Strings in SQL are denoted by surrounding them with single quotes. 'like this'
- Numeric constants, integers and reals are also allowed, and SQL uses the common notation for reals such as -12.34 or $1.234E45$ $(1.23 * 10^{45})$.
- The result of a comparison is a boolean value: either TRUE or FALSE.
- Boolean values may be combined by the logical operators AND, OR and NOT.
- Two strings are equal if they are the same sequence of characters.
- When we compare strings by one of the "less than" operators, such as $<$ or $>=$, we are asking whether one precedes the other in lexicographic order. (dictionary order, or alphabetically).

- `S LIKE P`
  - provides the capability to compare strings on the basis of a simple pattern match.
  - S is a string and P is a pattern, that is, a string with the optional use of the special characters % and _
    - **%**: any sequence of 0 or more characters.
    - **_**: any one character.

- Two consecutive apostrophes in a string represent a single apostrophe and do not end the string.
    - ‘ I’’m studying for BDAD ’
- $<$ on dates means that the first date is earlier than the second.
- $<$ on times means that the first is earlier, within the same day, than the second.
- When we operate a NULL value and any value, including another NULL, using an arithmetic operator like $*$ or $+$, the result is NULL.
- When we compare a NULL value and any value, including another NULL, using a comparison operator like $=$ or $>$, the result is UNKNOWN.

- `x IS NULL`
    - to ask if x has the value NULL

- `ORDER BY <list_of_attributes>`
    - to get output in sorted order
    - the order is by default ascending (ASC)
    - append keyword DESC to an attribute to get descending order.

6.7. Disambiguating attributes

- Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. SQL solves this problem by allowing us to place a relation name and a dot in front of an attribute. Thus $R.A$ refers to the attribute A of relation R.
- The relation name, followed by a dot, is permissible even in situations where there is no ambiguity.

6.8. Subqueries

- Is a query that is part of another.
- Subqueries must be parenthesized.
- Subqueries can have subqueries, and so on, down as many levels as we wish.
- An atomic value that can appear as one component of a tuple is referred to as a scalar. We can use a subquery as if it were a constant. It may appear in a WHERE clause any place we would expect to find a constant or an attribute representing a component of a tuple.

6.8.1. Union ($\cup$)

```
(subquery1)
UNION
(subquery2);
```

- By default, in SQL, the union operator eliminates duplicates.
- `UNION ALL` to allow duplicates.

6.8.2. Intersection ($\cap$)

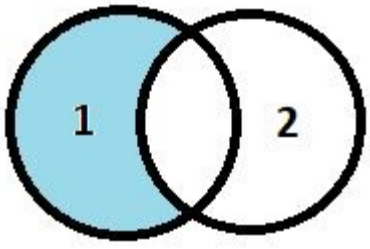
```
(subquery1)
INTERSECT
(subquery2);
```

- By default, in SQL, the intersect operator eliminates duplicates.
- `INTERSECT ALL` to allow duplicates.

6.8.3. Exception ($-$)

```
(subquery1)
EXCEPT
(subquery2);
```

- By default, in SQL, the intersect operator eliminates duplicates.
- `EXCEPT ALL` to allow duplicates.



6.8.4. Condition C involving subqueries

- `EXISTS R` is TRUE if and only if relation R is not empty.
- `S IN R` is TRUE if and only if S is equal to one of the values in R.
- `S NOT IN R` is TRUE if and only if S is equal to no value in R.
- `S > ALL R` is TRUE if and only if S is greater than every value in unary relation R. Similarly, the $>$ operator could be replaced by any of the other

five comparison operators with the meaning that S stands in the stated relationship to every tuple in R.

- `S > ANY R` is TRUE if and only if S is greater than at least one value in unary relation R. Similarly, the $>$ operator could be replaced by any of the other five comparison operators with the meaning that S stands in the stated relationship to at least one tuple in R.
- The EXISTS, ALL and ANY operators can be negated by putting NOT in front of the entire expression.
- The ALL and ANY operators are not supported by SQLite.

6.9. Join expressions

- Produce relations.
- May be used as subqueries in the FROM clause.

6.9.1. Cross join / Cartesian product

```
R CROSS JOIN S
```

- Returns every combination of tuples from R and S relations.

6.9.2. Inner joins

6.9.2.1. Theta-join

```
R JOIN S ON (condition)
```

- Returns the same as the product of $R\ x\ S$, but followed by a selection for whatever condition follows on ON.

6.9.2.2. Natural join

```
R NATURAL JOIN S
```

- The result will be a relation whose schema includes the attributes from R and S with the same name plus all the other attributes. Only returns those who have a match.

6.9.2.3. Using

```
R JOIN S USING (attrs)
```

- Returns the same as the NATURAL JOIN, but explicitly listing the attributes to be equated.

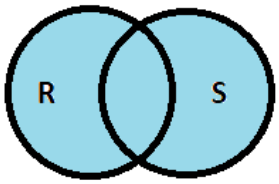
6.9.3. Outer joins

- Combines tuples as in theta-join but when they don't match they are added to the result with NULL values.

6.9.3.1. Full outerjoin

```
R NATURAL FULL OUTER JOIN S
```

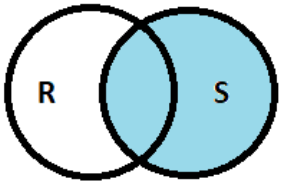
- To include unmatched tuples from both sides of a join.



6.9.3.2. Right outerjoin

```
R NATURAL RIGHT OUTER JOIN S
```

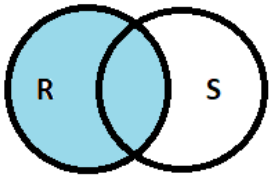
- Takes any tuples on the right side and if they don't have a match on a tuple from the left, it is still added to the result and padded with NULL values.



6.9.3.3. Left outerjoin

```
R NATURAL LEFT OUTER JOIN S
```

- Takes any tuples on the left side and if they don't have a match on a tuple from the right, it is still added to the result and padded with NULL values.



6.10. Aggregation operators

- There's five aggregation operators: SUM, AVG, MIN, MAX and COUNT.
- Except COUNT, all aggregation operators apply to a single attribute/column.
- Use these operators in the SELECT clause.

- `COUNT(*)`: counts all the tuples in the relation that is constructed from the FROM clause and WHERE clause.

- `COUNT(DISTINCT x)`: counts the number of distinct values in columns x.

- `COUNT(A)`: counts the number of tuples with non-NULL values for attribute A.

6.11. Group by

```
SELECT S
FROM R
WHERE C
GROUP BY a1, ..., ak; //list of grouping attributes
```

- Groups tuples.
- Only used in conjunction with aggregation operators.
- **Algorithm:**
  1. Compute the FROM and WHERE clauses.
  2. Group by the attributes in the GROUP BY clause. "Group" in the sense that all tuples are grouped together first by *a1*, then *a2*, …, then *ak*.
  3. Compute the SELECT clause: grouped attributes and aggregates.

6.12. Notes about GROUP BY, aggregation operators and NULL's

- The value NULL is ignored in any aggregation. It does not contribute to a sum, average, or count of an attribute, nor can it be a minimum or maximum in its column.
- NULL is treated as an ordinary value when forming groups. That is, we can have a group in which one or more of the grouping attributes are assigned the value NULL.
- When we perform any aggregation (except count) over an empty bag of values, the result is NULL.

6.13. Having

```
SELECT S
FROM R
WHERE C1
GROUP BY a1, ..., ak
HAVING C2;
```

- *C1* is any condition on the attributes *S.*
- *C2* is any condition on the aggregate expressions.
- The HAVING clause applies conditions to the results of aggregate functions.
- If we use a HAVING clause without a GROUP BY clause, the HAVING condition applies to all rows that satisfy the search condition. In other words, all rows that satisfy the search condition make up a single group.

7. Views

- **Relations** that are defined with a CREATE TABLE statement actually exist in the database. That is, a SQL system stores tables in some physical organization. They are persistent, in the sense that they can be expected to exist indefinitely and not to change unless they are explicitly told to change by a SQL modification statement.
- **Virtual views** is another class of SQL relations that do not exist physically. They are not stored in the database, but can be queried as if they existed.
- It is useful to have views/tables that are not stored to hide some data from some users. For these users, views are the only access to the database they have.
- A view may be queried exactly as if it were a stored table.
- It is also possible to write queries involving both views and base tables (relations).

7.1. View declaration

```
CREATE VIEW <view_name>
AS <query>;
```

7.2. Renaming attributes

```
CREATE VIEW Vname(newLetterA, newLetterE)
AS
    SELECT a, e
    FROM A, B;
```

- `newLetterA` refers to the attribute `a`
- `newLetterE` refers to the attribute `e`

7.3. View removal

```
DROP VIEW Vname;
```

- This statement deletes the definition of the view, so we may no longer make queries or issue modification commands involving this view.

8. Triggers

- **event-condition-action rules**
- When event occurs, check condition; if true, do action
- Triggers are only awakened when certain events, specified by the database programmer, occur.
- Once awakened by its triggering event, the trigger tests a condition. If the condition does not hold, then nothing else associated with the trigger happens in response to this event.
- If the condition of the trigger is satisfied, the action associated with the trigger is performed by the DBMS.
- Triggers are used to enforce constraints.

- **row-level triggers**: includes “FOR EACH ROW” statement
- **statement-level triggers**: does not include “FOR EACH ROW” statement

8.1. Trigger declaration

```
CREATE TRIGGER name
BEFORE/AFTER/INSTEAD OF events
[referencing variables]
[FOR EACH ROW]
WHEN (condition)
action;
```

8.2. Events

```
-> INSERT ON T

-> DELETE ON T

-> UPDATE [C1, ..., Cn] ON T
```

T: relation
C: attribute
[C1, …, Cn]: optional

8.3. Referencing variables

- References the data that was modified; references all the tuples that were modified and not all tuples of that table.
- **old/new row as var**: refer to the specified tuple
- **old/new table as var**: refer to the set of all the affected tuples

	row-level trigger	statement-level trigger	for deletes	for inserts	for updates
`old row as var`	✓	✓	✓		✓
`new row as var`	✓	✓		✓	✓
`old table as var`		✓	✓		✓
`new table as var`		✓		✓	✓

8.4. [FOR EACH ROW]

- Optional clause
- States that the trigger is activated once for each modified tuple
- The trigger is activated
    - x times if clause is present (action is executed x times, but the trigger is always executed once)
    - once if clause is not present

8.5. Condition

- Tests the condition, if it is true, action will be performed
- Optional (if the WHEN clause is missing, then the action is executed whenever the trigger is awakened)

8.6. Action

- Set of simple SQL statements + begin and end bracket

8.7. Example

R.A references S.B, cascaded delete

```
Create Trigger Cascade
After Delete On S
Referencing Old Row As O
For Each Row
[ no condition ]
Delete From R Where A = O.B
```

```
Create Trigger Cascade
After Delete On S
Referencing Old Table As OT
[ For Each Row ]
[ no condition ]
Delete From R Where A in (select B from OT)
```

9. Indexes

- Is a specialized type of “materialized view”.
- Is a stored data structure whose sole purpose is to speed up the access to specified tuples of one of the stored relations.
- Users don’t access indexes.
- Indexes are used underneath by the query execution engine.

9.1. Motivation

- When relations are very large, it becomes expensive to scan all the tuples of a relation to find those (perhaps very few) tuples that match a given condition. With indexes we don’t have to scan the entire table.

9.2. Syntax

```
CREATE INDEX indexName ON T(A);

CREATE INDEX indexName ON T(A1, A2, ... An);

CREATE UNIQUE INDEX indexName ON T(A);

DROP INDEX indexName;
```

9.3. Factors to consider when choosing indexes

- The existence of an index on an attribute may greatly speed up the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.
- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming. Each modification on a relation R forces us to change any index on one or more of the modified attributes of R.
- Often, the most useful index we can put on a relation is an index on its key.
- Choosing indexes: evaluate the benefit of each of the candidate indexes. If at least one provides a positive benefit (i.e, it reduces the average execution time of queries), then choose that index.

9.4. Data structures

- **Balanced trees**: for equalities or inequalities conditions; logarithmic runtime.
- **Hash tables**: only for equality conditions; constant runtime

10. Transactions

- **Transaction** is a collection of one or more operations on the database that must be executed atomically; that is, either all operations are performed or none are.
- Each transaction begins with a specific task and ends when all tasks in the group successfully complete. If any of the tasks fail, the transaction fails. Therefore, a transaction has only two results: success or failure.

10.1. Commands

- The following commands are used to control transactions. It is important to note that these statements cannot be used while creating tables and are only used with the DML commands such as INSERT, UPDATE and DELETE.

- `START TRANSACTION`
  - Is used to mark the beginning of a transaction.

- `START TRANSACTION READ ONLY`
  - must be executed before the transaction begins. Read-only transactions access the same data to run in parallel, but they are not allowed to run in parallel with a transaction that wrote the same data.

- `START TRANSACTION READ WRITE`
  - This option is the default.

- `COMMIT`
  - Causes the transaction to end successfully.

- `ROLLBACK`
  - Causes the transaction to abort. Any changes made in response to the SQL statements of the transaction are undone.

10.2. ACID properties

10.2.1. Atomicity

- Certain combinations of database operations need to be done atomically; that is, either they are both done or neither is done.
- A simple solution is to have all changes to the database done in a local workspace, and only after all work is done do we commit the changes to the database, whereupon all changes become part of the database and visible to other operations.

10.2.2. Consistency

- Ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.

10.2.3. Isolation

- Per transaction.
- It does not affect the behaviour of any other transaction.
- Specific to reads.
- Serializability: operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions.

10.2.4. Durability

- Guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure.

10.3. Isolation levels

- **Dirty data:** data written by a transaction that has not yet been committed.
- **Dirty read**: is a read of dirty data written by another transaction.
- **Phantom tuples:** tuples that result from insertions into the database while our transaction is executing.

- increased concurrency + decrease overhead = increased performance
- weaker consistency

weak

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Commited	Not allowed	Allowed	Allowed
Repeatable Read	Not allowed	Not allowed	Allowed
Serializable (default)	Not allowed	Not allowed	Not allowed

strong

- decreased concurrency + increased overhead = decreased performance
- strong consistency

10.3.1. Read uncommitted

```
SET TRANSACTION [READ WRITE (default) | READ ONLY]
    ISOLATION LEVEL READ UNCOMMITTED;
```

- A transaction may perform dirty reads.

10.3.2. Read committed

```
SET TRANSACTION [READ WRITE (default) | READ ONLY]
    ISOLATION LEVEL READ COMMITTED;
```

- A transaction may not perform dirty reads.

10.3.3. Repeatable read

```
SET TRANSACTION [READ WRITE (default) | READ ONLY]
    ISOLATION LEVEL REPEATABLE READ;
```

- If a tuple is retrieved the first time, then we can be sure that the identical tuple will be retrieved again if the query is repeated. However, it is also possible that a second or subsequent execution of the same query will retrieve phantom tuples.

10.3.4. Serializable

```
SET TRANSACTION [READ WRITE (default) | READ ONLY]
    ISOLATION LEVEL SERIALIZABLE;
```

- A certain transaction must be serializable with respect to other transactions. That is, these transactions must behave as if they were run serially - one at a time, with no overlap.

11. NoSQL Databases

- Describes databases that store and manipulate data in other formats than tabular relations, i.e. non-relational databases.
- Needed for situations with massive volumes, flexible data structures and where scalability and availability are more important.

- **When should NoSQL be used:**
  - when huge amounts of data need to be stored and retrieved,
  - the relationship between the data you store is not that important,
  - the data is changing over time and is not structured,
  - support of constraints and joins is not required at database level,
  - the data is growing continuously and you need to scale the database regularly to handle the data.

- **Advantages:**
  - High scalability: **sharding** - partitioning of data and placing it on multiple machines (servers) in such a way that the order of the data is preserved.
  - High availability: **auto replication** feature in NoSQL databases makes it highly available because in case of any failure, data replicates itself to the previous consistent state.

- As the data volumes or number of parallel transactions increase, capacity can be increased by:
  - **Vertical scaling**: extending storage capacity and/or CPU power of the database server (Not easy to implement).
  - **Horizontal scaling**: multiple DBMS servers being arranged in a cluster (Easy to implement)
- NoSQL databases aim at near linear horizontal scalability (sharding).

11.1. Types of NoSQL databases

- **Key-value stores**
  - Stores data as (key, value) pairs.

- **Tuple and document stores**
  - **Tuple store** stores a unique key together with a vector of data.
  - **Document store** stores a collection of attributes that are labeled and unordered.

- **Column-oriented databases**
    - Stores data tables as sections of columns of data.
    - All values of a column are placed together on disk.
    - Null values do not take up storage space anymore.

- **Graph based databases**
    - Apply graph theory to the storage of information of records.
    - One-to-one, one-to-many, and many-to-many structures can easily be modeled in a graph.

11.2. CAP theorem

- CAP theorem states that a distributed computer system cannot guarantee the following three properties at the same time:
    - **Consistency**: all nodes see the same data at the same time.
    - **Availability**: guarantees that every request receives a response indicating a success or failure result.
    - **Partition tolerance**: the system continues to work even if nodes go down or are added.

11.3. BASE principle

- Most NoSQL databases follow the BASE principle:
    - **Basically available**: NoSQL databases adhere to the availability guarantee of the CAP theorem.
    - **Soft state**: the system can change over time, even without receiving input.
    - **Eventual consistency**: the system will become consistent over time.

11.4. NoSQL versus relational databases

	Relational databases	NoSQL databases
Data paradigm	Relational tables	Key-value (tuple) based Document based Column based Graph based XML, object based Others: time series, probabilistic, etc.
Distribution	Single-node and distributed	Mainly distributed
Scalability	Vertical scaling, harder to scale horizontally	Easy to scale horizontally, easy data replication
Openness	Closed and open source	Mainly open source
Schema role	Schema-driven	Mainly schema-free or flexible schema
Query language	SQL as query language	No or simple querying facilities, or special-purpose languages
Transaction mechanism	ACID: Atomicity, Consistency, Isolation, Durability	BASE: Basically available, Soft state, Eventual consistency
Feature set	Many features (triggers, views, stored procedures, etc.)	Simple API
Data volume	Capable of handling normal-sized data sets	Capable of handling huge amounts of data and/or very high frequencies of read/write requests

	RDBMS	NoSQL
Relational	Yes	No
SQL	Yes	No
Column stores	No	Yes
Scalability	Limited	Yes
Eventually consistent	Yes	Yes
BASE	No	Yes
Big volumes of data	No	Yes
Schema-less	No	Yes