**The Relational Model**

**Database Systems** – Powerful tools for **creating** and **managing** large amounts of data **efficiently** and **allowing** it to persist over long periods of time.

**Database**: a **collection** of **data** organized in a particular way and managed by a DBMS.

**DBMS**: Database Management System – a **collection** of software **programs** that manage a database.

**Data Independence:** Applications and users should be **insulated** from how data is **structured** and **stored**.

**Logical** independence: Protection from **changes** in **logical** structure of data (i.e. attributes/schema)

**Physical** independence: Protection from **changes** in **physical** structure of data (i.e. how it is stored)

**DBMS:**

A DBMS is a set of computer programs that support:

* At least one **data model** (a mathematical abstraction for representing data) to define a **database** and an associated **high-level language** to **create**, **manage** and **query** the database.
* Ability to **store** and **manage** large amounts of **persistent** data **efficiently**.
* **Transaction** management, **concurrency** control.
* **Access control** (limit access of certain data to certain users).
* **Resiliency** (ability to recover from crashes).

**Data Model:**

A **collection** of mathematical **concepts** for describing data – how it is **represented**, **organized** and **structured** in the **database**.

The **relational** **model** of data is the most widely used today.

Every data model has to come with a **data** **language** for defining, updating, manipulating, and retrieving data from its representation.

**Data Sublanguages:**

A data language has two parts (**sublanguages**):

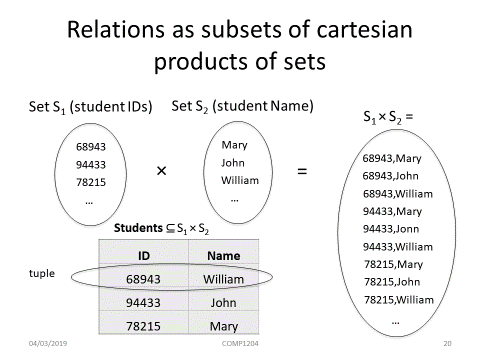
* Data **definition** language (DDL) – syntax for describing “**database** **templates**” in terms of the underlying data model.
* Data **manipulation** language (DML) – for insertion, deletion, update…

**Eg. DDL for XML – XML Schema; DML for XML – XPath/XQuery.**

**Relational Data Model:**

Formally: **a relation is a subset of a Cartesian product of sets.**

Informally: Tables.



**Basic Notions from Discrete Mathematics**

A k-tuple is an **ordered** **sequence** of k objects (need **NOT** be **distinct**) – (a, b, a, a, c) is a 5-tuple.

**If D1 = {0, 1} and D2 = {a, b, c, d} then |D1| \* |D2| = 8**

A **k-ary** **relation** R is a subset of a Cartesian product of k sets. K is called the **arity** of the relation.

* E.g. Unary: R = {0, 2, 4}
* Binary: T = {(a,b): a and b have the same b-day}
* Ternanry: S = {(m,n,s): s = m + n}

**The number of tuples is the number of rows.**

**Properties of relations**

* Each row represents a **k-tuple** of R.
* The ordering of rows is **immaterial** (relations are just sets).
* All rows are **distinct.**
* The ordering of the attributes in **not significant.**
* The significance of each column is conveyed by the **name** we give it.

**Relation Schemas and Relational Database Schemas**

A **k-ary** **relation** schema **R(A1, A2,…,Ak)** is relation **name** and an **ordered sequence** (A1,A2,…,Ak) of k attributes:

* StudentCourses(StudentID, courseID)
* Student(Id, name)
* Courses(ID, name, DeptID)

A k-ary relation **schema** is a “**blueprint**”, a “**template**” for some k-ary relation.

In a **DBMS** implementation the **schema** gives a more precise definition of the **types** (i.e. attribute domain):

* StudentCourses(**int**: StudentID, **int**: courseID)
* Student(**int**: ID, **string**: name)
* Courses(**int**: ID, **string**: name, **int**: DeptID)

An **instance** of **a relation schema** is a **relation** conforming to the schema**.**

A **relational** **database** **schema** is a **set** **of** **relational** **schemas**.

A **relational database instance,** or a **database** of a relational schema is **a set of relations Ri**, each of which is an instance of the relation schema.

**A relation schema is essentially the schema for a table. In a relational database (what people typically mean when they say database) each take can be referred to as a "relation" . Hence a relational schema is the design for the table. It includes none of the actual data, but is like a blueprint or design for the table, so describes what columns are on the table and the data types. It may show basic table constraints ( e.g. if a column can be null) but not how it relates to other tables.**

**That is where the database schema comes in. The database schema describes how the tables (relations) connect and are built. So this will sore where there are one to one, one to many or other joins between tables, but will not show details about how the individual tables are designed.**

**You could say that a database schema is made up of lots of relation schema and shows how they work together.**

**Keys**

A set of **attributes** forms a key for a relation if we do not allow **two different tuples** in a relation instance **to have the same values** in all the attributes of the key.

**Relational Algebra**

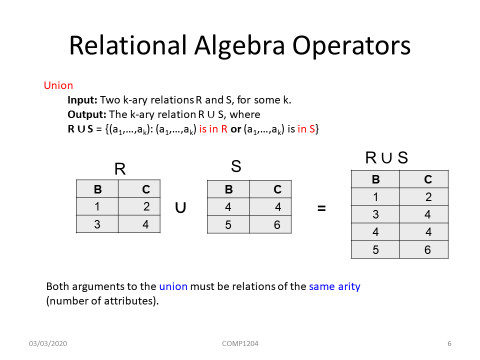
**Procedural vs Declarative**

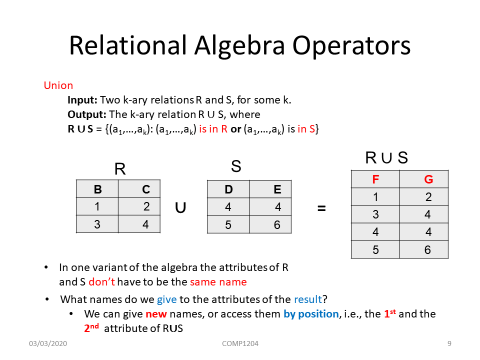
* **Procedural** (programming) languages or definitions specify **how** the task is to be accomplished (sequence of operations) (e.g. Java).
* **Declarative** languages specify what has to be accomplished (as opposed to “how”) (e.g. definition of the FD closure: X+ = {Ai: F ⊧ X → Ai})

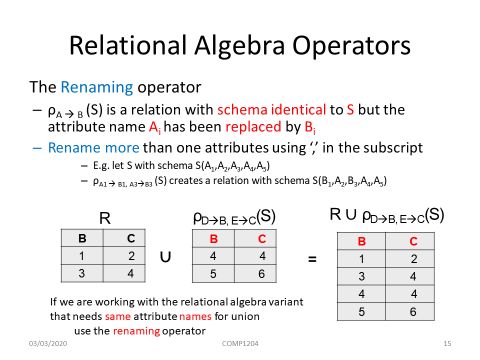
**Relational Algebra**

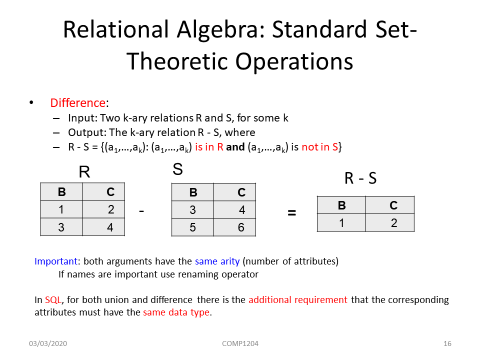
* Is a procedural query language.
* **Algebra**:
  + **Operators** --- symbols denoting procedures that construct new values from given values
  + **Operands** --- variables or values from which new values can be constructed
* **Relational**:
  + Operands are **relations** or variables that represent relations.
  + Allows us to do common operations on relations.
* This results in a **query language**.

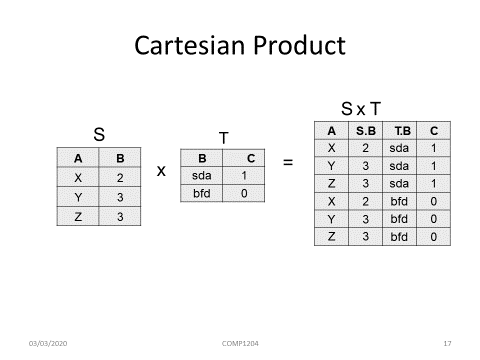
**Relational Algebra Operators**

* **Group I** 
  + Union
  + Difference
  + Cartesian Product
* **Group II** 
  + Projection
  + Selection
* **Depending on the variant** of relational algebra we might need:
  + Renaming operator
* Relational Algebra consists of all expressions obtained by combining these five (or six) basic operations in syntactically correct ways.









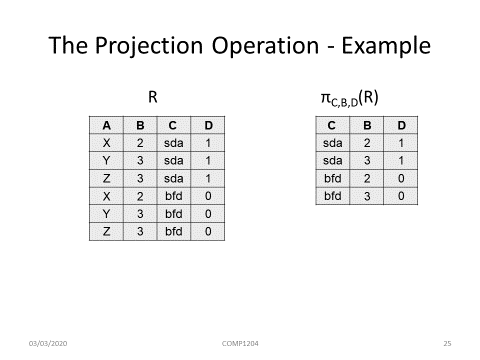
**Algebraic Laws**

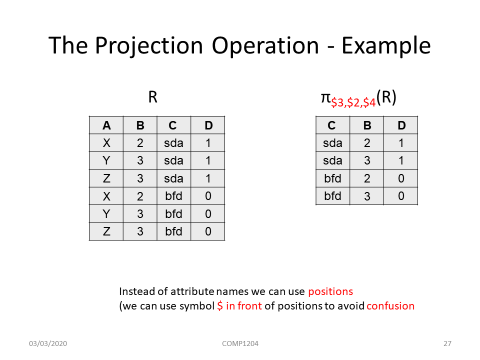
**Q:** Why are **algebraic laws important**?

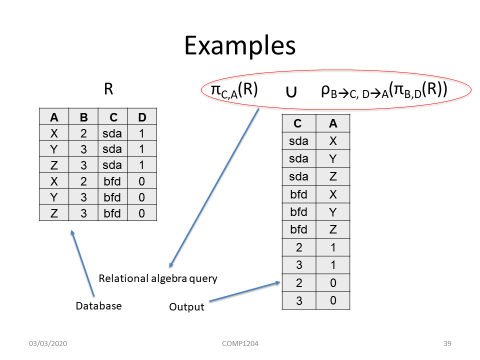
**A:** They are **important** in **query** **processing** and **optimization** to **transform** a query to a **less** **costly** equivalent one. Applying correct algebraic laws ensures the **correctness** of the transformations.

**The projection operation**

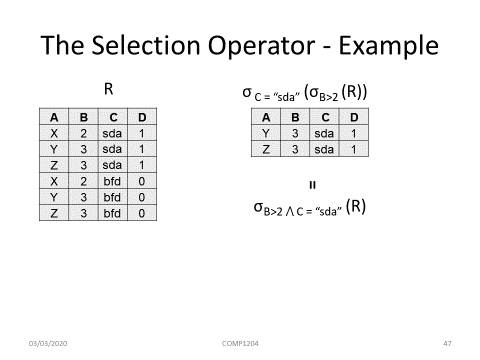
* Given a table R, we want to:
  + **Rearrange** the order of the columns.
  + **Suppress** some columns.







**The selection operator**

* Selection is unary operation of the form
  + **σΘ(R),**
    - R is a relation and **Θ** is a condition that can be applied as a test to each row of R
* Selection returns the subset of **R** consisting of all rows that satisfy **Θ**.

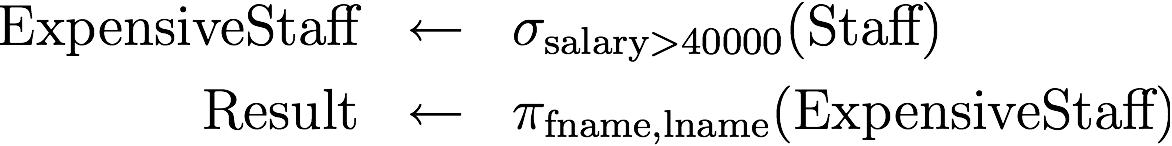
**Algebraic laws for the selection operation**

* **σΘ1(σΘ2(R)) = σΘ2(σΘ1(R))**
* **σΘ1(σΘ2(R)) = σΘ1⋀Θ2(R)**
* **σΘ(R⨉S ) = σΘ(R) ⨉ S if Θ mentions only attributes of R**

**Decomposing Complex Operations**

* We can combine the relational algebra operations to build expressions of arbitrary complexity.
* For the sake of legibility, we can name intermediate expressions using the assignment operator (🡨)

latex-image-1

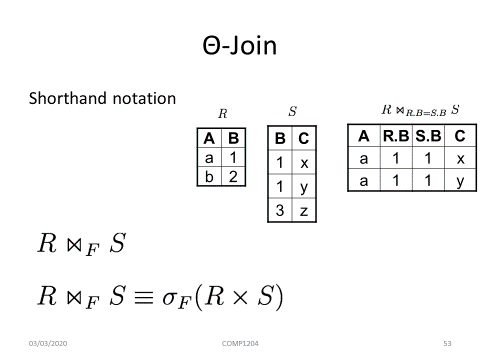


**Derived Relational Algebra Operations**

* Some operations have **special names** or can be **derived** by **other** relational algebra operations.
* E.g. **Intersection**
* **R∩S = R – (R – S) = S – (S – R) (example slide 51)**

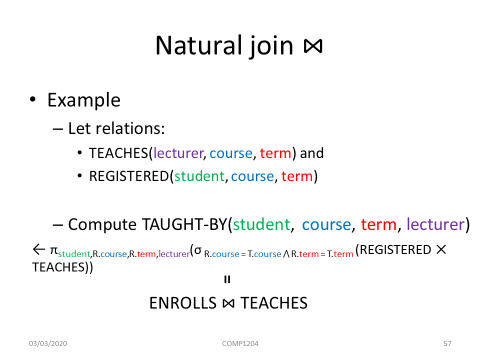
**Θ-Join**

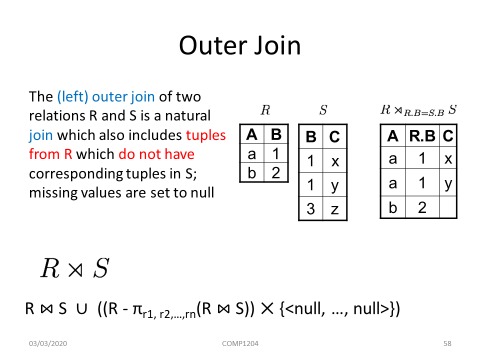
* A **special query** of the form σΘ(R⨉S) when we **combine relations**.

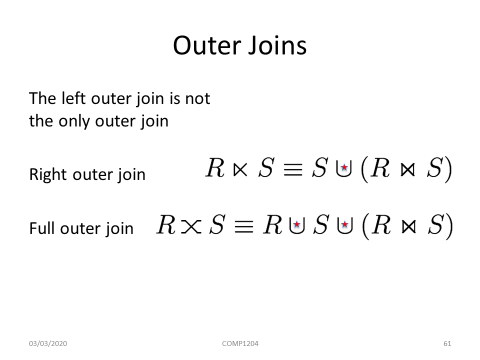
****

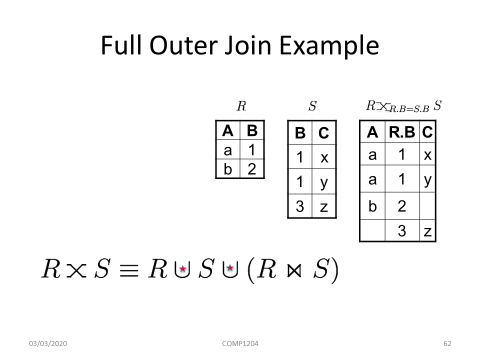
**Equijoins and Natural Joins**

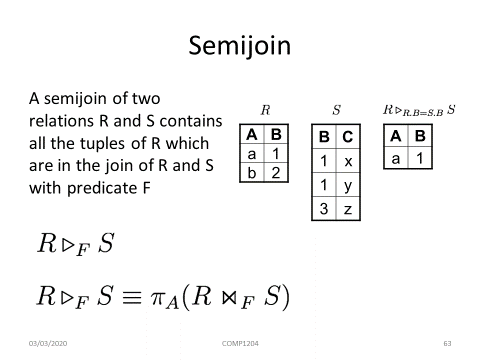
* An **equijoin** is a theta join in which the only predicate operator used is **equality (=)**.
* A **natural join** is an **equijoin** over all the **common attributes** of the joined relations; one occurrence of each common attribute is removed from the resulting relation.





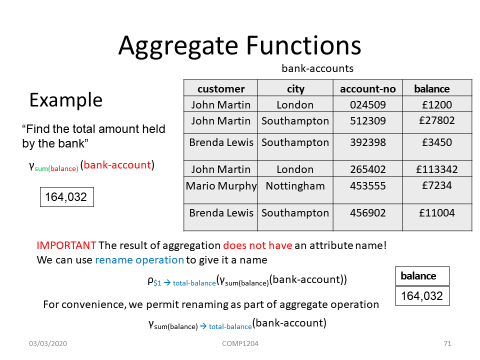


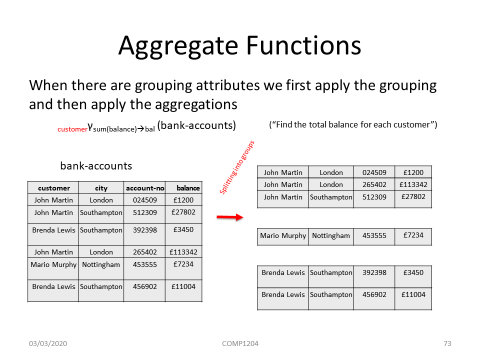


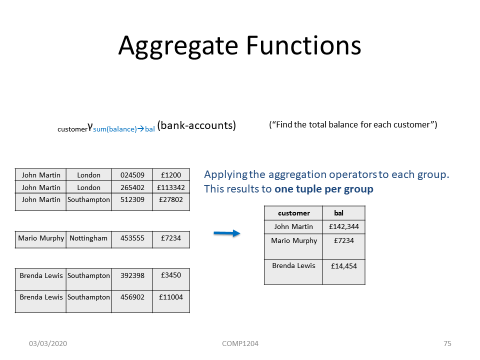


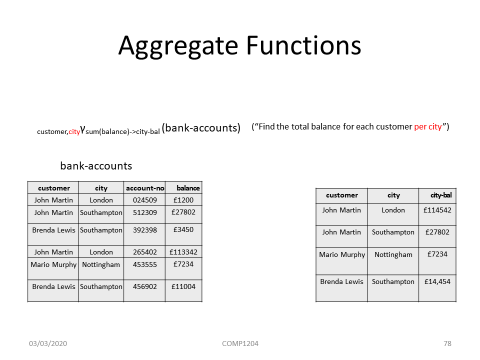
**Aggregate Functions**

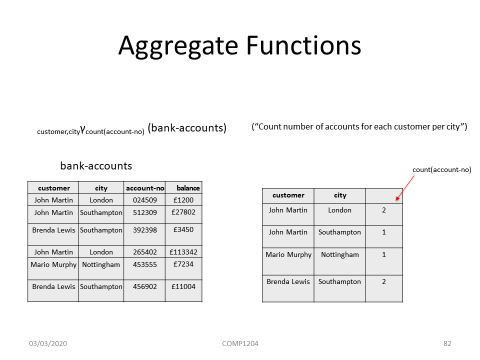
* We use a special **unary operator γ** to perform grouping and aggregate operations.
* <at1, at2, … , atm> γ <op1(atr1), op2(atr2), … , opn(atrn)>(Relation-name)
* <at1, at2, … , atm> is a list of all attributes of the relation (might be empty)
* <op1(atr1), op2(atr2), … , opn(atrn)> is a list of operators, which can be one of the following:
  + avg
  + min
  + max
  + sum
  + count
* <op1(atr1), op2(atr2), … , opn(atrn)> cannot be empty!
* **THE MOST IMPORTANT SLIDES ARE HERE BUT CHECK OUT END OF PRESENTATION FOR MORE.**

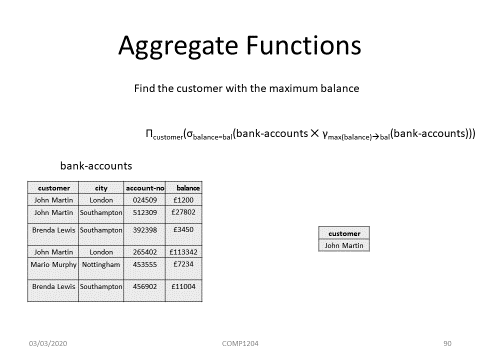








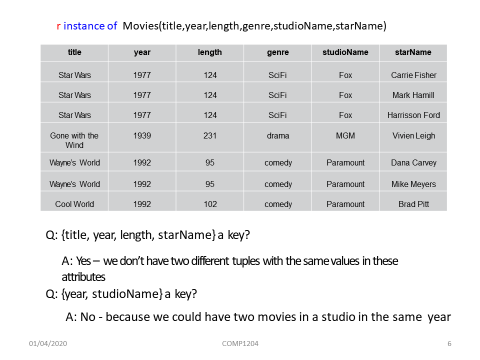


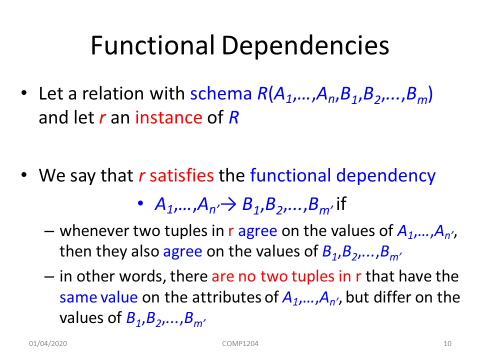


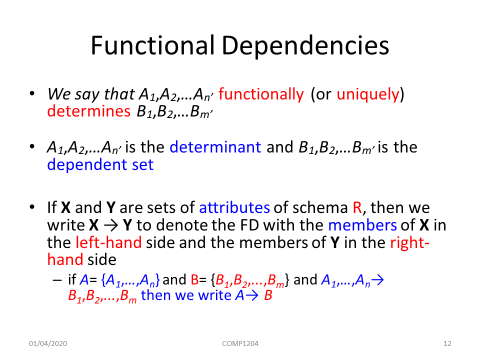
**Keys and Functional Dependencies**

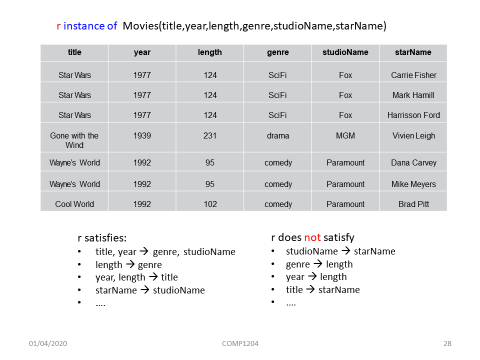
**Keys**

A set of **attributes** forms a key for a relation if we do **not** **allow** **two** **different** **tuples** in a relation instance to have the **same** **values** in all the attributes of the key.



****





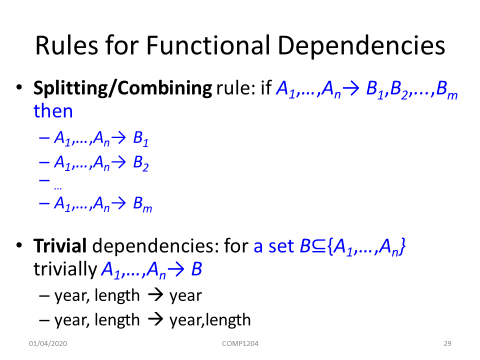
How to explain it:

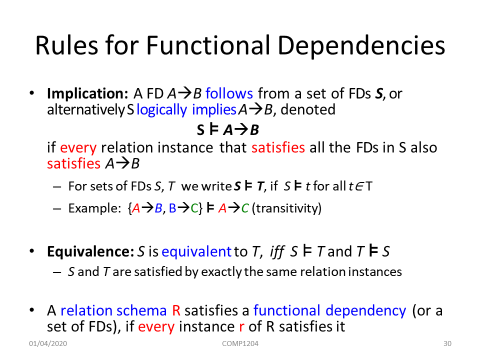
Eg. length -> genre

It is true because when you say a certain length you can say only 1 genre.

EG. genre -> length

It is false because when you say a certain genre there is more than 1 length.





**Keys**

A **super key** is a set of one or more attributes (columns), which can uniquely identify a row in a table.

A **candidate key** is selected from the set of super keys. The only thing we take care while selecting a candidate key is that it should not have any redundant attributes. That’s the reason they are also termed as **minimal super key**.

**All candidate keys are super keys**.

How to choose candidate keys from the set of super keys? We look for those keys from which we cannot remove any fields.

**Closure algorithms - used for computing keys!**

