Characteristics that distinguish the db approach from the approach of programming with data files

L.O.	4.1
Session	1
Tag	Final Prsntn

File based	DB based
Inconsistency	
Redundancy	

▼ Self-describing nature of data

- the definition of the whole primary database (meta-data) is stored in the database catalog
- DBMS can access diverse db by xtracting the db definitions from the catalog and using these definitions

▼ Insulation between programs and data, and data abstraction

Program-data independence

The structure of data is stored in the DBMS catalog seperately from the access program ⇒ hides details that are not of interest to most db users.

Data abstraction 🕉

- Conceptual representation, without showing how the data is stored or how the operation is implemented

- ***data model*** for conceptual representation: objects, their attribs, their interrelationships

▼ Support of multiple views of the data

View

- subset of the database, or containing virtual data not explicitly stored in the db

Example: the view for user who is interested in accessing and printing students' transcripts is different from the one whose user is interested in checking if students have taken all the prereq of each course

▼ Sharing of data and multi-user transaction processing

concurrency control: ensure several users updating the same data do so in a controlled manner ⇒ correct result.

Example: when several movies go-ers try to reserve a seat in a theatre, make sure that only one ticket buyer can access and make reservation for one seat at a time

OLTP (Online Transaction Processing):

- Online banking
- Online ticket booking systems: flight, movies, …
- Sending a text message
- Order entry
- Add a product to the shopping cart
- An example of the OLTP system is the ATM center. Assume that a couple has a joint account with a bank. One day both simultaneously reach different ATM centers at precisely the same time and want to withdraw the total amount present in their bank account. However, the person that completes the authentication process first will be able to get money. In this case, the OLTP system makes sure that the withdrawn amount will be never drawn more than the amount present in the bank. The key to note here is that OLTP systems are optimized for transactional superiority instead of data analysis.

transaction: executing program or process that includes $\geq$ 1 database operations, such as reading or updating database records {

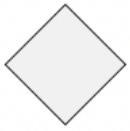
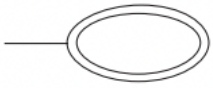
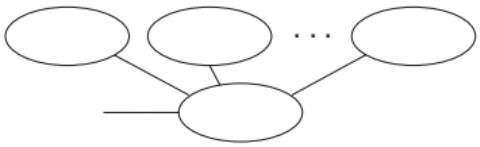
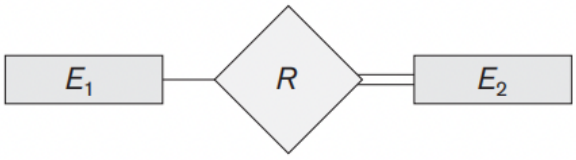
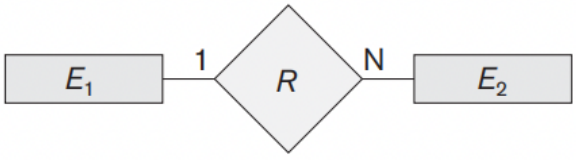
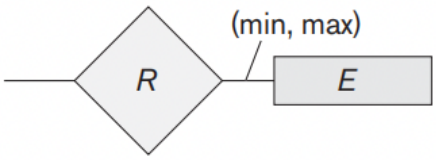
- **atomicity**: By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.
  - **Abort**: If a transaction aborts, changes made to the database are not visible.
  - **Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing rule'.

- **consistency**: This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above. The total amount before and after the transaction must be maintained.
  - Total **before T** occurs = **500 + 200 = 700**.
  - Total **after T** occurs = **400 + 300 = 700**. Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.
- **isolation**: This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
- **durability**: This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

Design a conceptual schema using ER model and its extended version

L.O.	2.1
Session	2 3
Tag	Ass. ClWrk Final Prsntn

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Indentifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of E_2 in R
	Cardinality Ratio 1 : N for E_1:E_2 in R
	Structural Constraint (min, max) on Participation of E in R

Entities and Attributes

Attributes

Composite	- can be divided into subparts - each subpart has independent meaning
Simple (Atomic)	- cannot be divided into subparts
Single valued	example: *Age* is the single-valued attribute of a person
Multi valued	example: *College_degree* is the multi-valued attribute because different people have different number of college degrees they have earned.
Complex	Composite + multi-valued

Key attributes

An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity type in Figure 7.6 because no two companies are allowed to have the same name. For the PERSON entity type, a typical key attribute is Ssn (Social Security number). Sometimes several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a *composite attribute* and designate it as a key attribute of the entity type. Notice that such a composite key must be *minimal*; that is, all component attributes must be included in the composite attribute to have the uniqueness property. Superfluous attributes must not be included in a key. In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure 7.7(a). Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for *every entity set* of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. It is not the property of a particular entity set; rather, it is a constraint on *any entity set* of the entity type at any point in time. This key constraint (and other constraints we discuss later) is derived from the constraints of the miniworld that the database represents.

Some entity types have *more than one* key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR (Figure 7.7) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, neither of which is a key on its own. An entity type may also have *no key*, in which case it is called a *weak entity type* (see Section 7.5).

In our diagrammatic notation, if two attributes are underlined separately, then *each is a key on its own*. Unlike the relational model (see Section 3.2.2), **there is no concept of primary key in the ER model that we present here**; the primary key will be chosen during mapping to a relational schema (see Chapter 9).

Relationship Types

Cardinality Ratios for Binary Relationships

The **cardinality ratio** for a binary relationship specifies the *maximum* number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to (that is, employs) any number of employees,9 but an employee can be related to (work for) only one department. This means that for this particular relationship WORKS_FOR, a particular department entity can be related to any number of employees (N indicates there is no maximum number). On the other hand, an employee can be related to a maximum of one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N. An example of a 1:1 binary relationship is MANAGES (Figure 7.12), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that—at any point in time—an employee can manage one department only and a department can have one manager only. The relation ship type WORKS_ON (Figure 7.13) is of cardinality ratio M:N, because the miniworld rule is that an employee can work on several projects and a project can have several employees.

Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds as shown in Figure 7.2. Notice that in this notation, we can either specify no maximum (N) or a maximum of one (1) on participation. An alternative notation (see Section 7.7.4) allows the designer to specify a specific *maximum number* on participation, such as 4 or 5.

Participation Constraints and Existence Dependencies

The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the *minimum* number of relationship instances that each entity can participate in, and is sometimes called the **minimum cardinality constraint**. There are two types of participation constraints—total and partial—that we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance (Figure 7.9). Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation**, meaning that every entity in *the total set* of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency**. In Figure 7.12 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial**, meaning that *some* or *part of the set of* employee entities are related to some department entity via MANAGES, but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type. In ER diagrams, total participation (or existence dependency) is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* (see Figure 7.2). Notice that in this notation, we can either specify no minimum (partial participation) or a minimum of one (total participation). The alternative notation (see Section 7.7.4) allows the designer to specify a specific *minimum number* on participation in the relationship, such as 4 or 5.

Weak Entity Type

Entity types that do not have key attributes of their own are called **weak entity types**. In contrast,**regular entity types** that do have a key attribute—which include all the examples discussed so far—are called **strong entity types**. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying** or **owner entity type,** and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type. A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship because a weak entity can not be identified without an owner entity. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (License_number) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 7.2). In our example, the attributes of DEPENDENT are Name (the first name of the dependent), Birth_date, Sex, and Relationship (to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, Birth_date, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to *own* the dependent entities that are related to it.

A weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are *related to the same owner entity*. 12 In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key. In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines (see Figure 7.2). The partial key attribute is underlined with a dashed or dotted line.

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. In the preceding example, we could specify a multivalued attribute Dependents for EMPLOYEE, which is a composite attribute with component attributes Name, Birth_date, Sex, and Relationship. The choice of which representation to use is made by the database designer. One criterion that may be used is to choose the weak entity type representation if there are many attributes. If the weak entity participates independently in relationship types other than its identifying relationship type, then it should not be modeled as a complex attribute. In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we illustrate in Section 7.9.

Relationship Types of Degree > 2

Choosing between Binary and Ternary (or higher) Relationships

The ER diagram notation for a ternary relationship type is shown in Figure 7.17(a), which displays the schema for the SUPPLY relationship type that was displayed at the entity set/relationship set or instance level in Figure 7.10. Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where s is a SUPPLIER who is currently supplying a PART p to a PROJECT j. In general, a relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type.

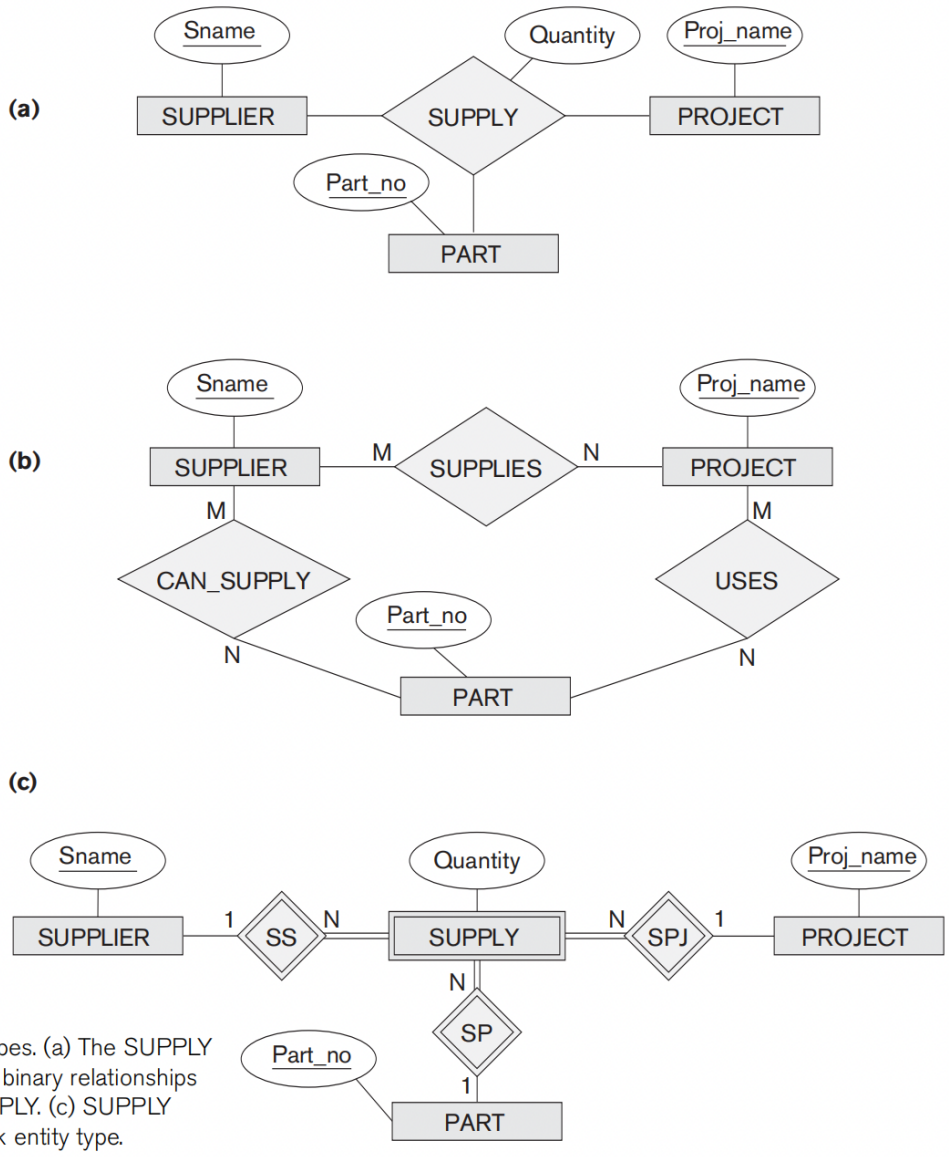


Figure 7.17
Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

Figure 7.17(b) shows an ER diagram for three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents different information than do three binary relationship types. Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (s, p) whenever supplier s *can supply* part p (to any project); USES, between PROJECT and PART, includes an instance (j, p)

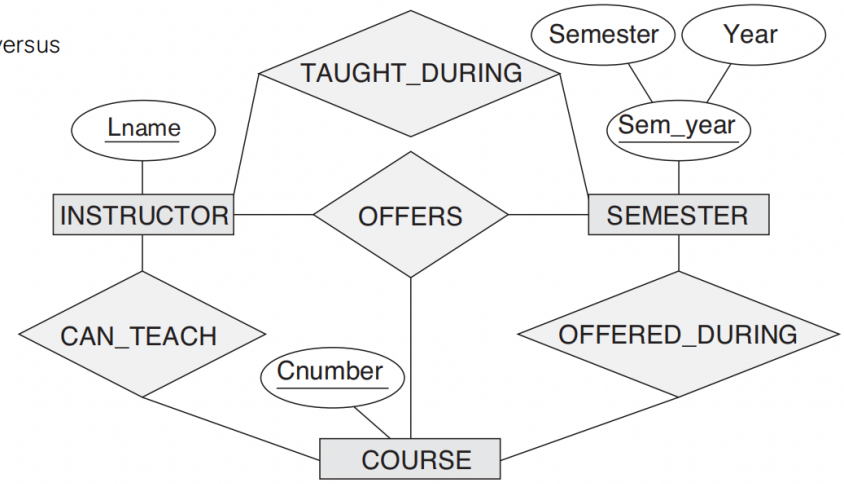
whenever project *j* uses part *p*; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (*s*, *j*) whenever supplier *s* supplies *some part* to project *j*. The existence of three relationship instances (*s*, *p*), (*j*, *p*), and (*s*, *j*) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply that an instance (*s*, *j*, *p*) exists in the ternary relationship SUPPLY, because the *meaning is different*. It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree *n* or should be broken down into several relationship types of smaller degrees. The designer must base this decision on the semantics or meaning of the particular situation being represented. The typical solution is to include the ternary relationship *plus* one or more of the binary relationships, if they represent different meanings and if all are needed by the application.

Some database design tools are based on variations of the ER model that permit only binary relationships. In this case, a ternary relationship such as SUPPLY must be represented as a weak entity type, with no partial key and with three identifying relationships. The three participating entity types SUPPLIER, PART, and PROJECTare together the owner entity types (see Figure 7.17(c)). Hence, an entity in the weak entity type SUPPLY in Figure 7.17(c) is identified by the combination of its three owner entities from SUPPLIER, PART, and PROJECT.

It is also possible to represent the ternary relationship as a regular entity type by introducing an artificial or surrogate key. In this example, a key attribute Supply_id could be used for the supply entity type, converting it into a regular entity type. Three binary N:1 relationships relate SUPPLY to the three participating entity types.

Another example is shown in Figure 7.18. The ternary relationship type OFFERS represents information on instructors offering courses during particular semesters; hence it includes a relationship instance (*i*, *s*, *c*) whenever INSTRUCTOR *i* offers COURSE *c* during SEMESTER *s*. The three binary relationship types shown in Figure 7.18 have the following meanings: CAN_TEACH relates a course to the instructors who *can teach* that course, TAUGHT_DURING relates a semester to the instructors who *taught some course* during that semester, and OFFERED_DURING

Figure 7.18
Another example of ternary versus binary relationship types.



relates a semester to the courses offered during that semester *by any instructor*. These ternary and binary relationships represent different information, but certain constraints should hold among the relationships. For example, a relationship instance (*i*, *s*, *c*) should not exist in OFFERS *unless* an instance (*i*, *s*) exists in TAUGHT_DURING, an instance (*s*,*c*) exists in OFFERED_DURING, and an instance (*i*, *c*) exists in CAN_TEACH. However, the reverse is not always true; we may have instances (*i*, *s*), (*s*, *c*), and (*i*, *c*) in the three binary relationship types with no corresponding instance (*i*, *s*,*c*) in OFFERS. Note that in this example, based on the meanings of the relationships, we can infer the instances of TAUGHT_DURING and OFFERED_DURING from the instances in OFFERS, but we cannot infer the instances of CAN_TEACH; therefore, TAUGHT_DURING and OFFERED_DURING are redundant and can be left out.

Although in general three binary relationships *cannot* replace a ternary relationship, they may do so under certain *additional constraints*. In our example, if the CAN_TEACH relationship is 1:1 (an instructor can teach one course, and a course can be taught by only one instructor), then the ternary relationship OFFERS can be left out because it can be inferred from the three binary relationships CAN_TEACH, TAUGHT_DURING, and OFFERED_DURING. The schema designer must analyze the meaning of each specific situation to decide which of the binary and ternary relationship types are needed.

Notice that it is possible to have a weak entity type with a ternary (or *n*-ary) identifying relationship type. In this case, the weak entity type can have *several* owner entity types. An example is shown in Figure 7.19. This example shows part of a database that keeps track of candidates interviewing for jobs at various companies, and may be part of an employment agency database, for example. In the requirements, a candidate can have multiple interviews with the same company (for example, with different company departments or on separate dates), but a job offer is made based on one of the interviews. Here, INTERVIEW is represented as a weak entity with two owners CANDIDATE and COMPANY, and with the partial key Dept_date. An INTERVIEW entity is uniquely identified by a candidate, a company, and the combination of the date and department of the interview.

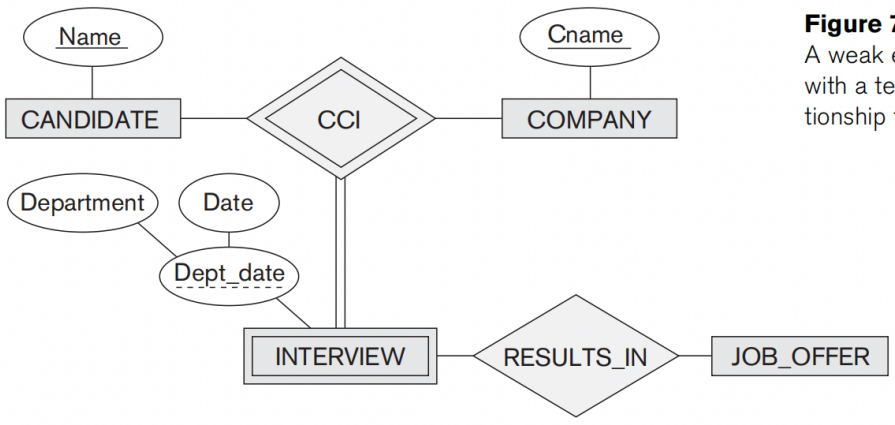


Figure 7.19
A weak entity type INTERVIEW with a ternary identifying relationship type.

Constraints on Ternary (or higher) Relationships

There are two notations for specifying structural constraints on *n*-ary relationships, and they specify different constraints. They should thus *both be used* if it is important to fully specify the structural constraints on a ternary or higher-degree relationship. The first notation is based on the cardinality ratio

notation of binary relationships displayed in Figure 7.2. Here, a 1, M, or N is specified on each participation arc (both M and N symbols stand for *many* or *any number*).15 Let us illustrate this constraint using the SUPPLY relationship in Figure 7.17.

Recall that the relationship set of SUPPLY is a set of relationship instances (*s*, *j*, *p*), where *s* is a SUPPLIER, *j* is a PROJECT, and *p* is a PART. Suppose that the constraint exists that for a particular project-part combination, only one supplier will be used (only one supplier supplies a particular part to a particular project). In this case, we place 1 on the SUPPLIER participation, and M, N on the PROJECT, PART participations in Figure 7.17. This specifies the constraint that a particular (*j*, *p*) combination can appear at most once in the relationship set because each such (PROJECT, PART) combination uniquely determines a single supplier. Hence, any relationship instance (*s*, *j*, *p*) is uniquely identified in the relationship set by its (*j*, *p*) combination, which makes (*j*, *p*) a key for the relationship set. In this notation, the participations that have a 1 specified on them are not required to be part of the identifying key for the relationship set.16 If all three cardinalities are M or N, then the key will be the combination of all three participants.

The second notation is based on the (min, max) notation displayed in Figure 7.15 for binary relationships. A (min, max) on a participation here specifies that each entity is related to at least *min* and at most *max relationship instances* in the relationship set. These constraints have no bearing on determining the key of an *n*-ary relationship, where $n > 2^{17}$ but specify a different type of constraint that places restrictions on how many relationship instances each entity can participate in.

Determine a normal form (1NF, 2NF, 3NF, BCNF) based on its given PK and Functional dependencies

L.O.	2.4
Session	
Tag	Final

key	definition	note
superkey	attrib or {attribs} that uniquely id a record in a table	may contain unnecessary attribs
candidate key	a superkey such that no proper subset is a superkey within the relation. $\{\exists x(\text{Superkey}(x) \wedge \forall y(y \subset x \rightarrow \neg\text{SuperKey}(y)) \rightarrow \text{Candidate key}(x))\}$	is ***unique*** and ***irreducible***
primary key	candidate key that is selected to uniquely id records in a table	
foreign key	attrib, or {attribs} that match candidate key of some (possibly same) relation.	

▼ First Normal Form (1NF)

Data Atomicity: Ensuring that there is only one single instance value per column field

Enforce the Data Atomicity rule and remove duplication of data

Loading... This Course Video Transcript Back-end developers write applications that end-users use to interact with databases. Some common tasks that end-users carry out using these applications include

https://www.coursera.org/learn/intro-to-databases-back-end-development/lecture/YEH1G/first-normal-form-1nf

▼ Second Normal Form (2NF)

Loading... This Course Video Transcript Back-end developers write applications that end-users use to interact with databases. Some common tasks that end-users carry out using these applications include

https://www.coursera.org/learn/intro-to-databases-back-end-development/lecture/dtFfc/second-normal-form-2nf

- Is in 1NF

- Every non-prime attribute is fully functional dependent on the primary key: $\exists y \forall x (\text{PrimaryKey}(y) \land \text{NonPKey}(x) \implies \text{FullFunctDep}(x, y))$

Full Functional dependency: B is fully dependent on A if B is not functionally dependent on any proper subset of A

Partial dependency: B is partially dependent on A when there is some attribute that can be removed from A and yet the dependency still holds.

▼ Third Normal Form (3NF)

https://www.coursera.org/learn/intro-to-databases-back-end-development/lecture/ctzx9/third-normal-form-3nf

- Is in 1NF
- Is in 2NF
- No non-primary-key attribute is transitively dependent on the primary key: $\forall y (\text{PrimaryKey}(y) \implies \forall x (\text{NonPKey}(x) \implies \neg \text{TransDep}(x, y)))$

Transitive dependency: $\forall A \forall B \forall C (A \to B \land B \to C \land (B \nrightarrow A \lor C \nrightarrow A) \implies A \to C)$

▼ Advanced Normalization

👾 Basic concepts of Advanced Normalization

(1)	**Reflexivity:**	If B is a subset of A, then A → B
(2)	**Augmentation:**	If A → B, then A,C → B,C
(3)	**Transitivity:**	If A → B and B → C, then A → C
(4)	**Self-determination:**	A → A
(5)	**Decomposition:**	If A → B,C, then A → B and A → C
(6)	**Union:**	If A → B and A → C, then A → B,C
(7)	**Composition:**	If A → B and C → D then A,C → B,D

▼ Boyce-Codd Normal Form (BCNF)

Loading... This Course Video Transcript Database Management Essentials provides the foundation you need for a career in database development, data warehousing, or business intelligence, as well as

https://www.coursera.org/lecture/database-management/normal-forms-video-lecture-kcKrg

- More strict than 3NF
- Quite similar to 3NF
- A relation is in **Boyce-Codd Normal Form (*BCNF*)** if and only if *every determinant is a candidate key*.

❓ **NOTE:**
BCNF is a stronger form of 3NF, such that every relation in BCNF is also in 3NF. However, *a relation in 3NF is not necessarily in BCNF*.

✅ A relation with **two attributes** will **gurantee to be a BCNF**

Determine a normal form ≡ highest normal form

BCNF	if every determinant is a superkey
3NF	check for transitive dependency: either left hand side is superkey || right hand side is prime attribute
2NF	check for partial dependency: every non-prime attribute must be fully functionally dependent on the primary key

$$\text{Proper subset of one Candidate} \cup \text{Proper subset of another Candidate} = \text{Proper subset of Candidate}$$

https://www.youtube.com/watch?v=rQib0A1oUfg&list=PLdo5W4Nhv31b33kF46f9aFjoJPOkdlsRc&index=14

https://www.youtube.com/watch?v=M9_JPdlcCHQ&list=PLdo5W4Nhv31b33kF46f9aFjoJPOkdlsRc&index=15

Decompose a Normal form

https://www.youtube.com/watch?v=Q2Fyi7lT14E&list=PLTpNwHSD94utlrWtWLQ2frBFoheiQEAvU&index=14&t=5400s

In general:

- If a form is already BCNF → no need to do anything
- Else try to decompose to **next higher normal form**.
  - **Step 1:** Determine candidate keys
  - **Step 2:** Determine its current highest normal form
  - **Step 3:** Check and resolve violated condition to make it satisfies higher normal form (splitting relations into tables)

Develop a relational schema from a conceptual schema which is designed using the ER model and its extended versions.

L.O.	2.2
Session	4 5
Tag	Ass. ClWrk Final Prsntn

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

▼ ER → Relational

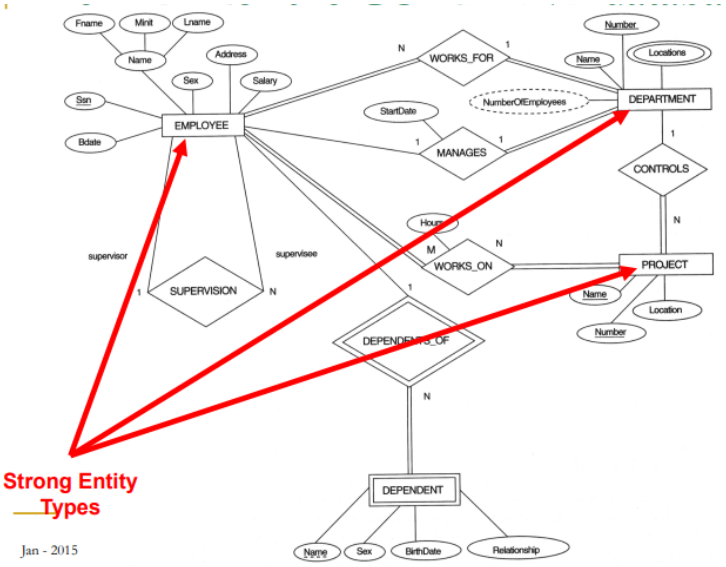
Reference Link:

https://www.youtube.com/watch?v=CZTkgMoqVss

▼ Step 1: Mapping regular entity types

Include all simple attributes of E, and only the simple component attrib of a composite attrib.

Choose 1 of the key attrib as PK

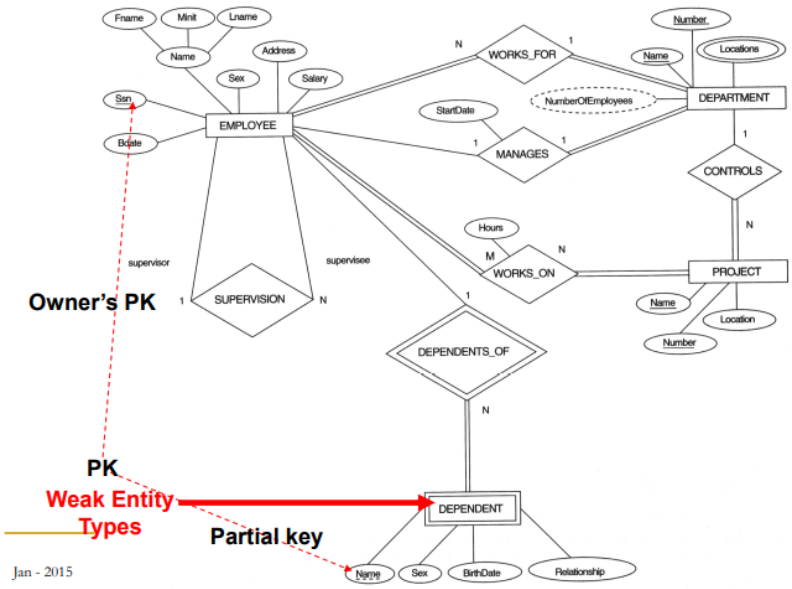


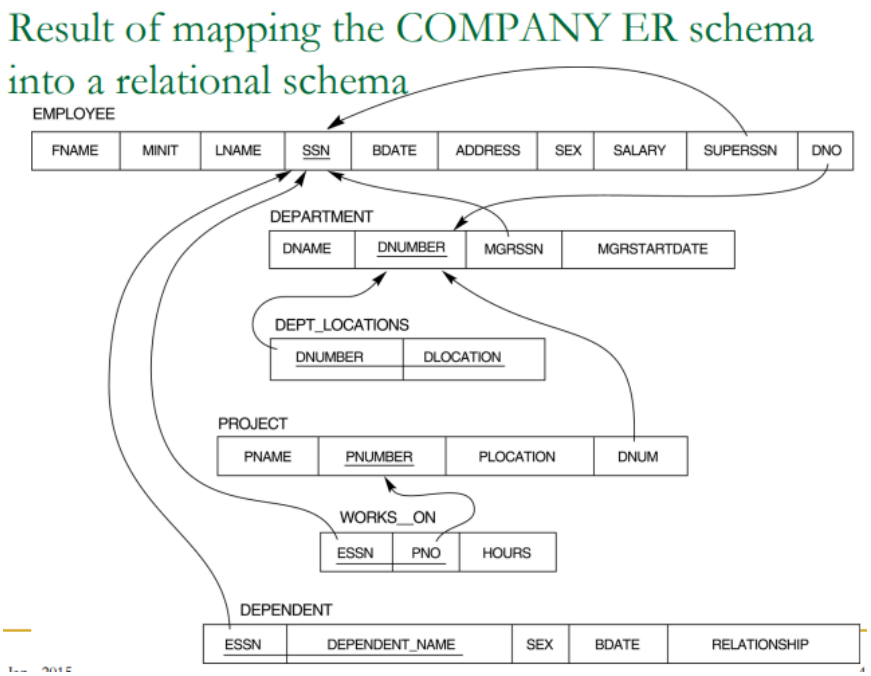
▼ Step 2: Mapping weak entity types

like step 1

include PK of owners as FK

PK: pk of owners + partial key of weak





▼ Step 3: Mapping 1:1 Relationship Types

Foreign key approach (Recomm. 😎)

1. **Foreign key approach:** Choose one of the relations—*S*, say—and include as a foreign key in *S* the primary key of *T*. It is better to choose an entity type with *total participation* in *R* in the role of *S*. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type *R* as attributes of *S*.

- Pick 1 of 2 in the relationship and include the PK of the one ***not total*** as FK of the **one with total participation**
- Include all simple attribs (**simple components of composite attributes**) of the 1:1

Merged relation approach

2. **Merged relation approach:** An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when *both participations are total,* as this would indicate that the two tables will have the exact same number of tuples at all times.

Both are total participation ⇒ **exact number of tuples at all times**

Relationship relation approach (Cross-reference)

3. **Cross-reference or relationship relation approach:** The third option is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we will see, this approach is required for binary M:N relationships. The relation R is called a **relationship relation** (or sometimes a **lookup table**), because each

tuple in R represents a relationship instance that relates one tuple from S with one tuple from T. The relation R will include the primary key attributes of S and T as foreign keys to S and T. The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R. The drawback is having an extra relation, and requiring extra join operations when combining related tuples from the tables.

⇒ Required for M:N relationship

▼ Step 4: Mapping of Binary 1:N

Foreign key approach (Recomm. 😎)

Include the PK of the 1 side into the N side as FK

Include simple attribs (or simple component of composite attribs) of the relationship

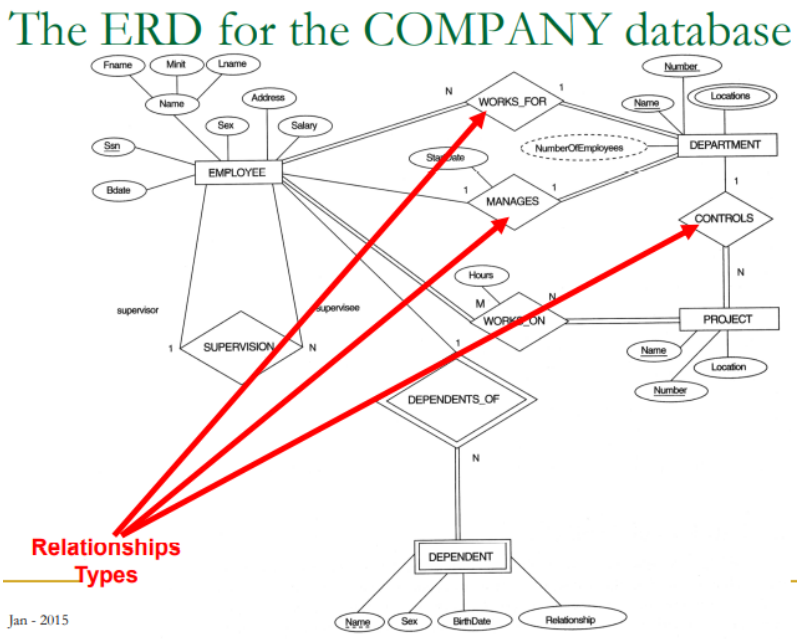
Relationship relation approach (Cross-reference)

Create a separate relation R(En, One)

PK = En

> 禁 Avoid when many tuples ⇒ NULL values

Mandatory mapping

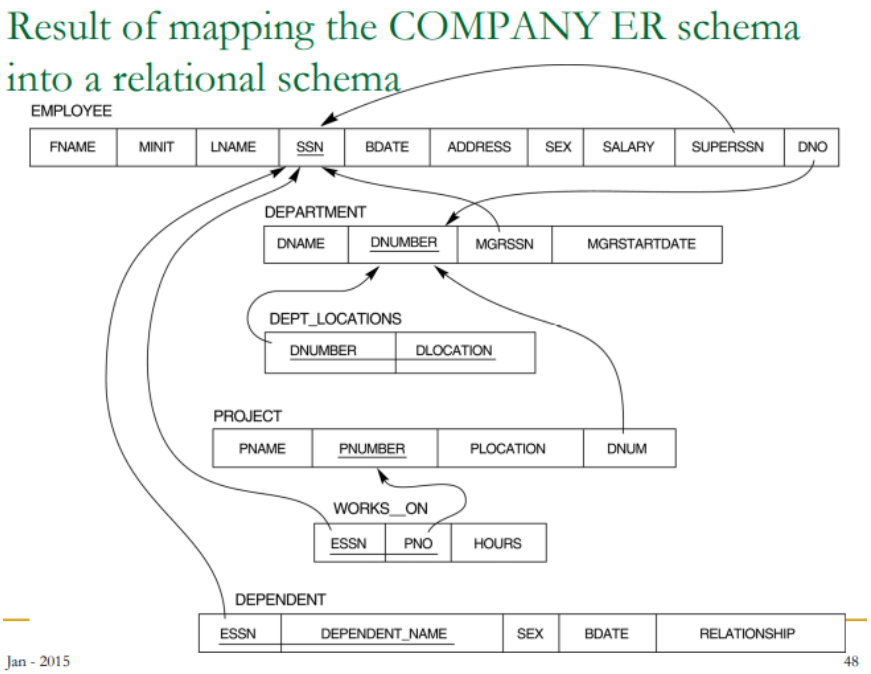


Mandatory membership class

❑ For two entity types E1 and E2: **If E2 is a mandatory member of an N:1** (or 1:1) relationship with E1, then the relation for E2 will include the prime attributes of E1 as a foreign key to represent the relationship

❑ **1:1 relationship:** If the membership class for E1 and E2 are both mandatory, a foreign key can be used in either relation

❑ **N:1 relationship:** If the membership class of E2, which is at the N-side of the relationship, is optional (i.e. partial), then the above guideline is not applicable

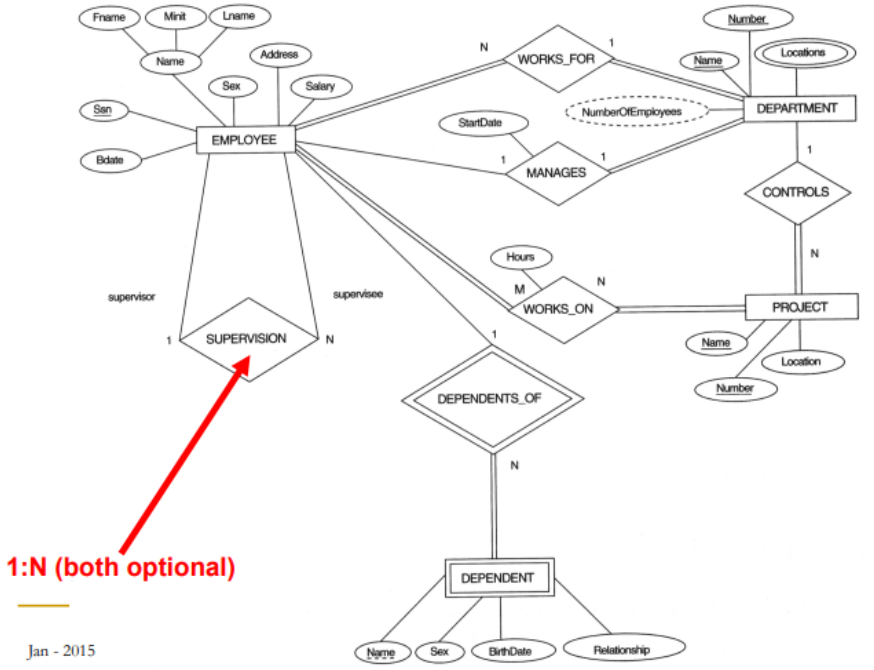


Optional mapping

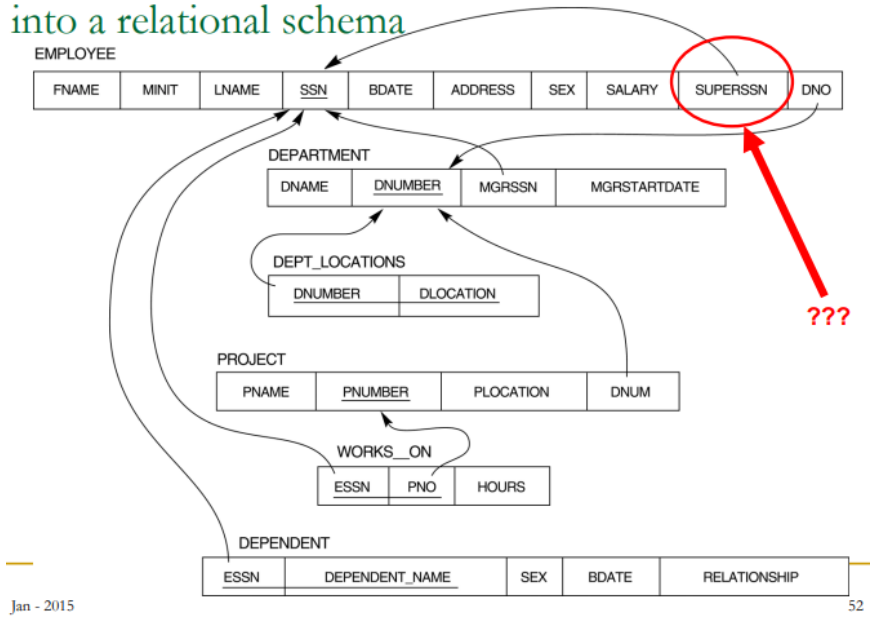
Optional membership classes

❑ **If entity type E2 is an optional member of the N:1 relationship with entity type E1** (i.e. E2 is at the Nside of the relationship), then the relationship is usually represented by a new relation containing the prime attributes of E1 and E2, together with any attributes of the relationship. The key of the entity type at the N-side (i.e. E2) will become the key of the new relation

❑ **If both entity types in a 1:1 relationship have the optional membership**, a new relation is created which contains the prime attributes of both entity types, together with any attributes of the relationship. The prime attribute(s) of either entity type will be the key of the new relation.



Result of mapping the COMPANY ER schema into a relational schema

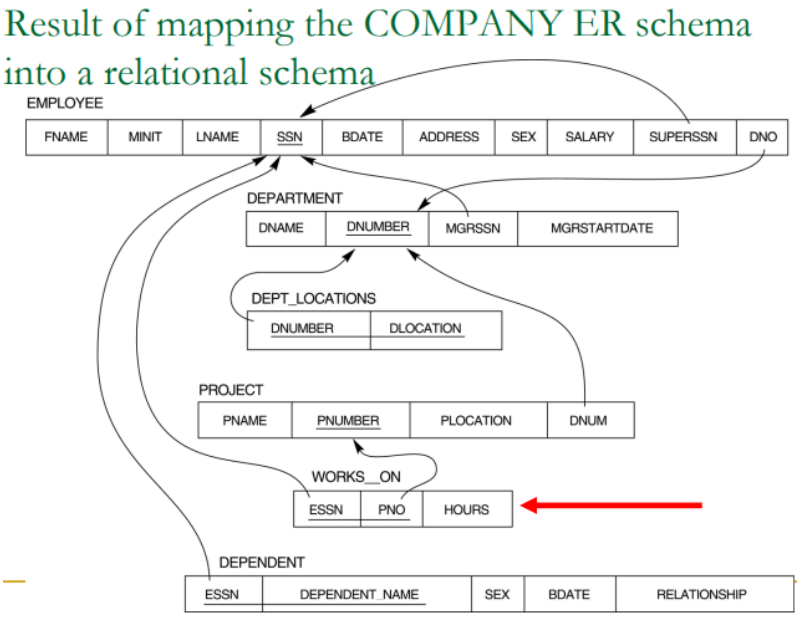


▼ Step 5: Mapping of M:N

Create a separate relation S(Em, En)

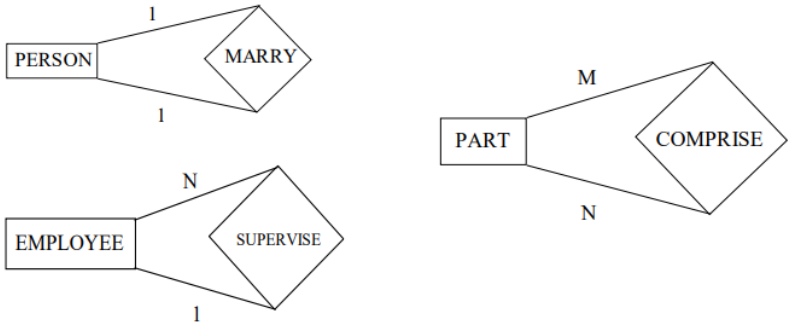
Include simple attribs… of the relationship

PK = Em + En



Transformation of recursive/involuted relationships

- Relationship among different instances of the same entity
- The name(s) of the prime attribute(s) needs to be changed to reflect the role each entity plays in the relationship



▼ Step 6: Mapping of multivalued attribs

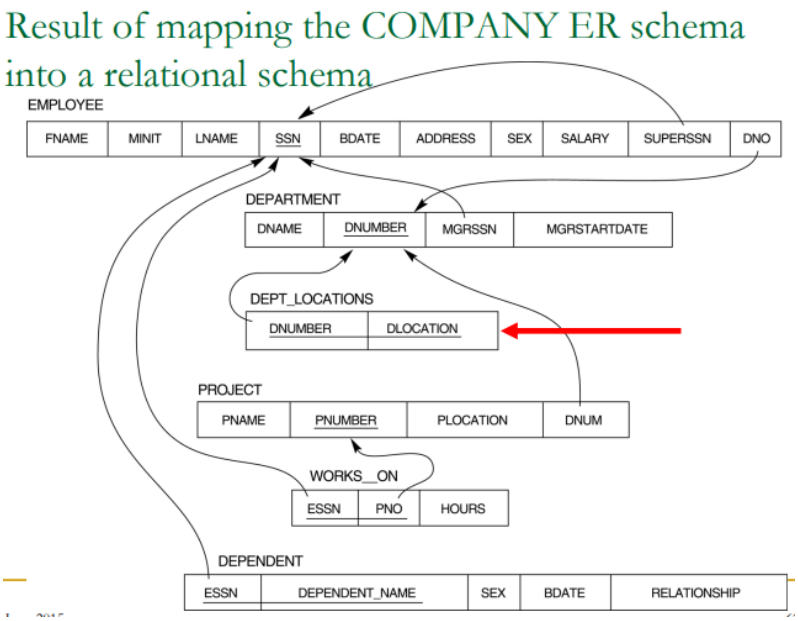
For each multival attrib A, new relation R.

If multival attrib is composite, include its components

Else include it as is.

Include PK of `relation` that represents entity or relationship as FK in R.

PK = FK + A



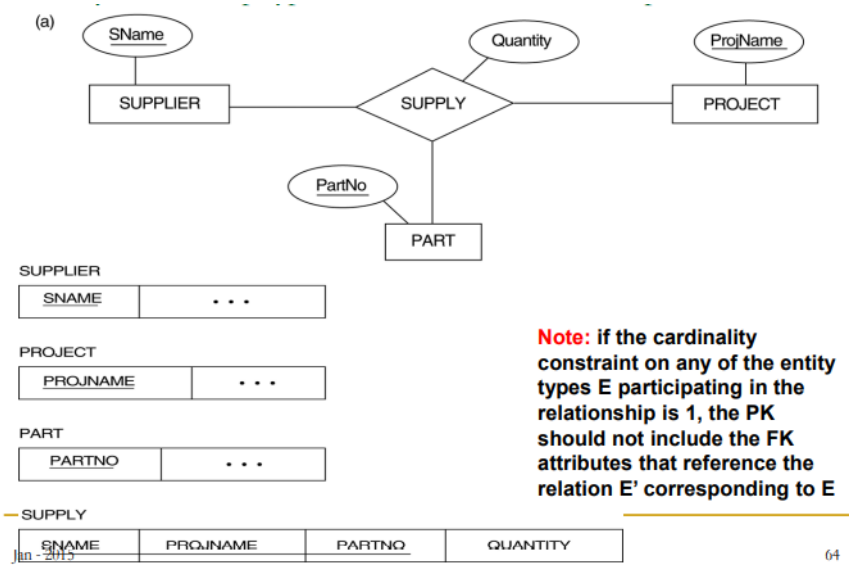
▼ Step 7: Mapping n-ary relationship types

For each n-ary relationships (n > 2), create a new relation S

Include PKs of participating entities as FK

Include simple attribs (or components of composite)… of the ***relationship***. (not relation)

If any cardinality constraint == 1, don't include it in the PK of S



▼ EER → Relational

2 options for multiple-relation approach: 8A and 8B

2 options for single-relation approach: 8C and 8D

C	superclass
K	C's primary key
$A_1, ...A_n$	C's attribs

$S_1, ...S_n$	C's subclasses
T	type attribute (could be `boolean`);

▼ Step 8: Options for Mapping Specialization or Generalization

▼ Option A:

> 🤑 works for any constraints: disjoin, overlapping, total, partial, you name it

L = C.relation;

L(K, C.attributes);

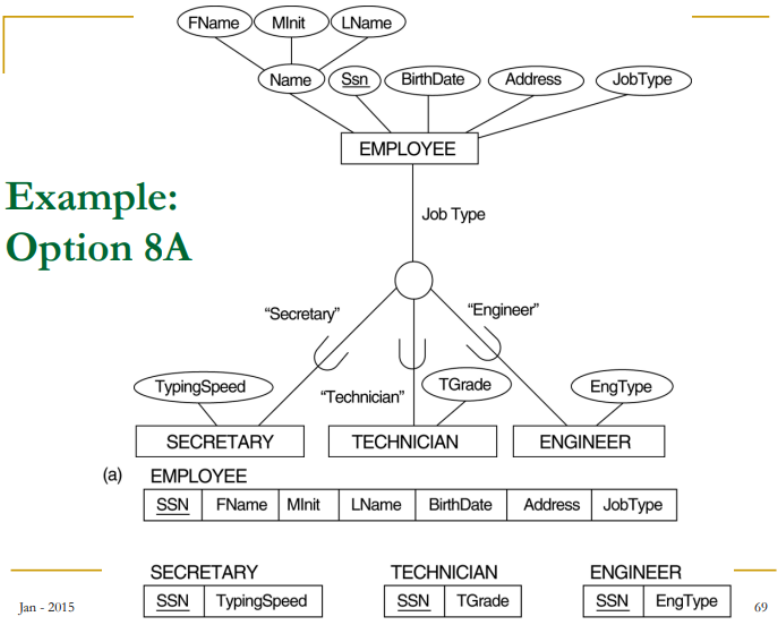
L.primary_key = K;

for each S_i:

  L_i = S_i.relation;

  L_i(K, C.attributes, S_i.attributes);

  L_i.primary_key = K;



▼ Option B:

> 🤑 works well only for (disjoint and total) constraints

for each S_i:

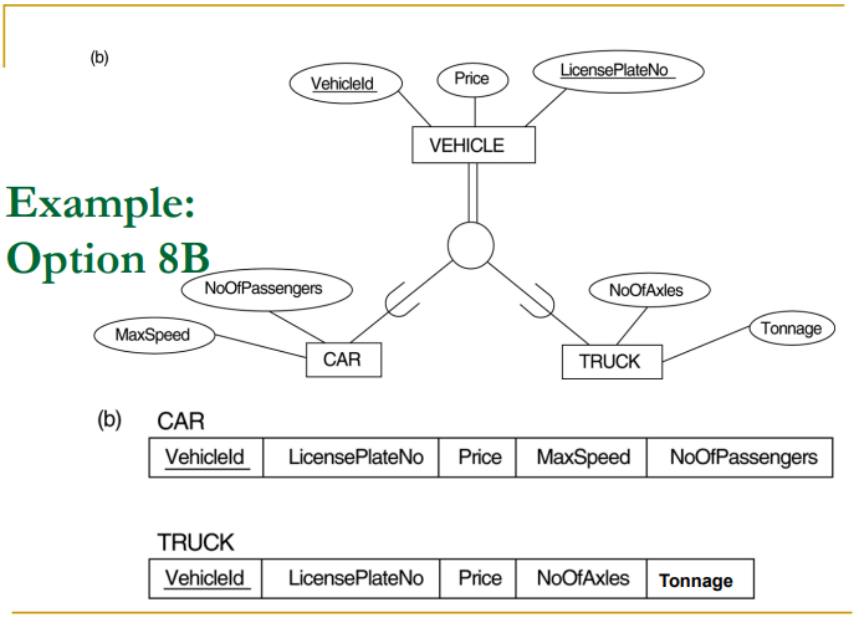
  L_i = S_i.relation;

  L_i(K, C.attributes, S_i.attributes);

L_i.primary_key = K;

💔 If not disjoint, redundant values for inherited attributes

💔 If not total, entity not belonging to any subclasses is lost



▼ Option C:

🤑 works for disjoint

L = C.relation;

L(K, C.attributes);

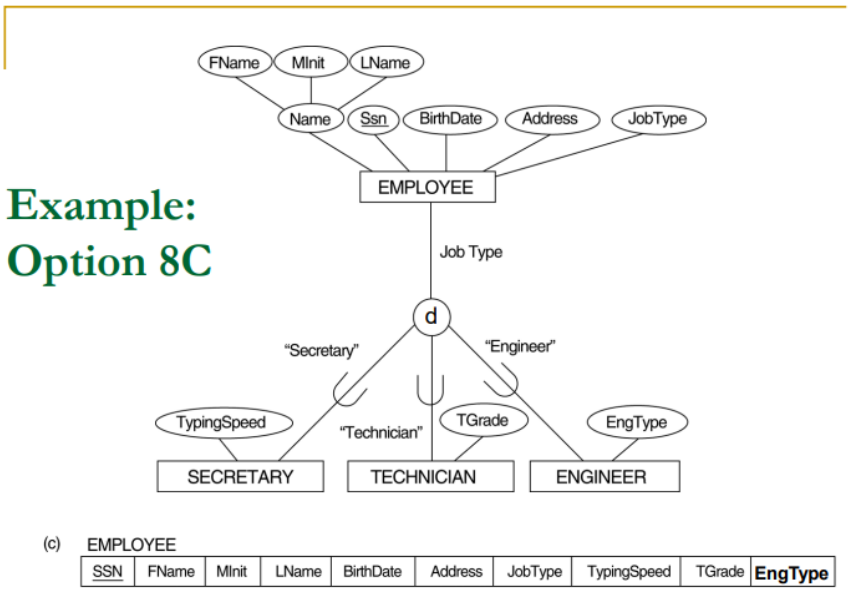
for each S_i:

  L.append(S_i.attributes);

L.append(Type); //(*This variable type is used to specify which type the subclass belongs to*)

L.primary_key = K;

💔 may generate huge number of NULL values ⇒ NOT RECOMMENDED if there are many subclass's specific attributes ⇒ a lot of NULL values!



▼ **Option D:**

> 🤑 works for overlapping

L = C.relation;

L(K, C.attributes);

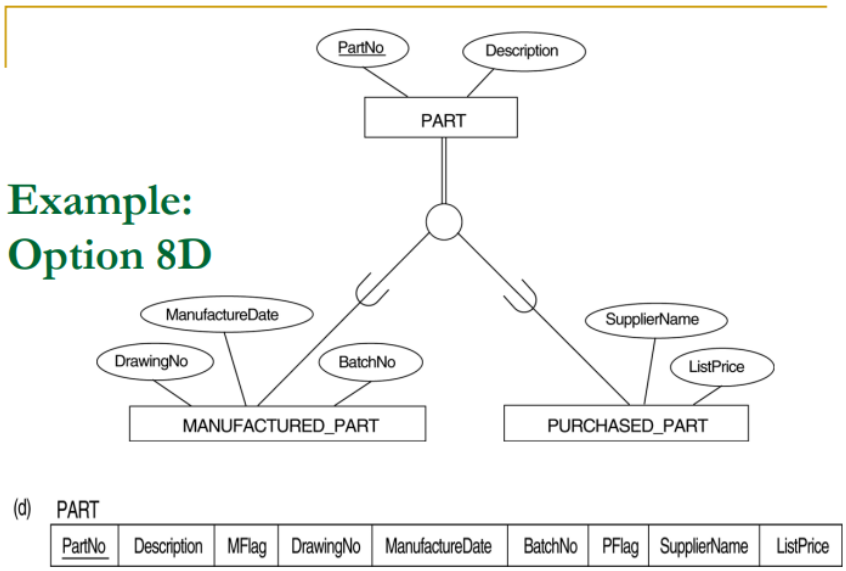
for each S_i:

  L.append(S_i.attributes);

  L.append(<`boolean`> Type_i);

L.primary_key = K;

> 💔 may generate huge number of redundant and NULL values ⇒ NOT RECOMMENDED if there are many subclass's specific attributes ⇒ a lot of redundant and NULL values!

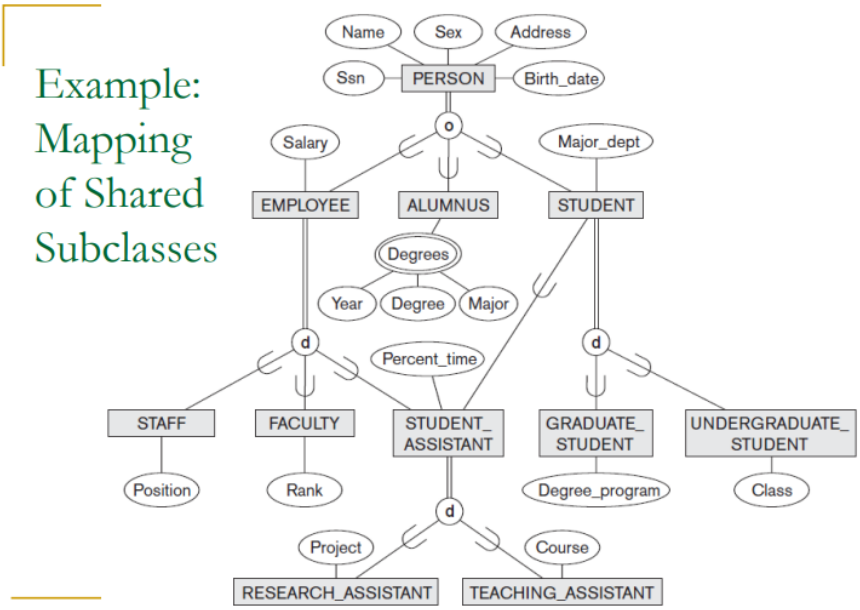


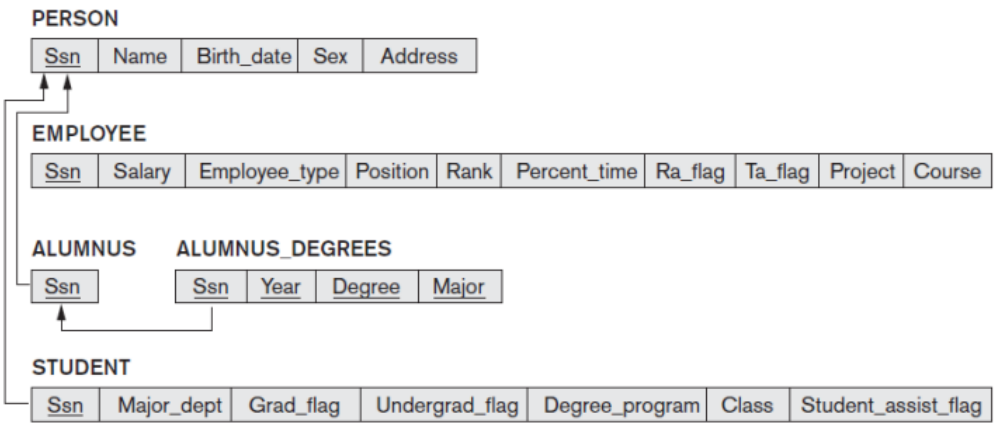
▼ Mapping of Shared Subclasses (Multiple Inheritance)

Apply the same method at step 8, just for more than 1 class.

❑ A shared subclass, such as STUDENT_ASSISTANT, is a subclass of several classes, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category.

❑ We can apply any of the options discussed in Step 8 to a shared subclass, subject to the restriction discussed in Step 8 of the mapping algorithm. Below both 8C and 8D are used for the shared class STUDENT_ASSISTANT.

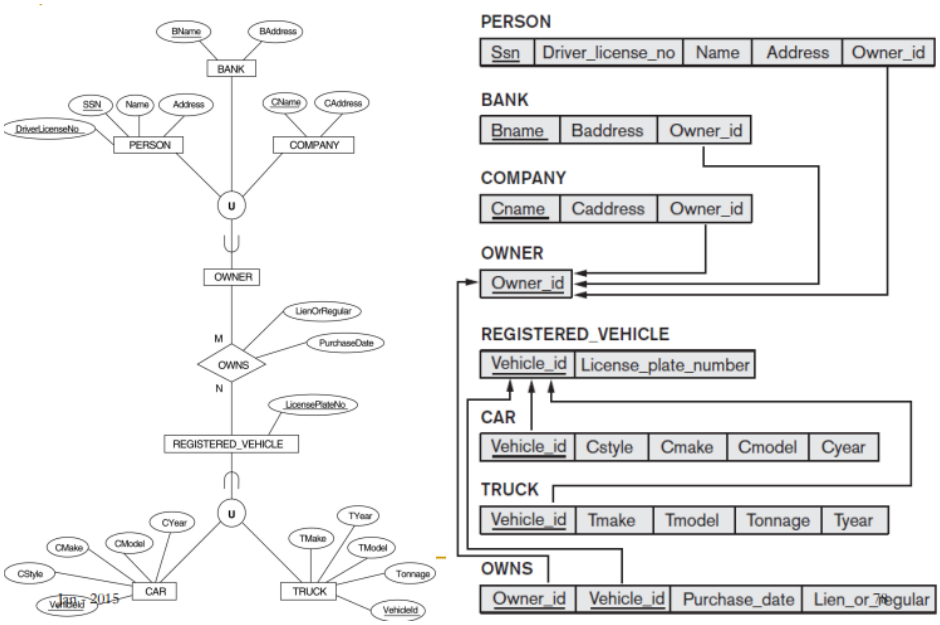




▼ Step 9: Mapping of Categories (Union Types)

We add another step to the mapping procedure — step 9 — to handle categories. *A category (or union type) is a subclass of the union of two or more superclasses that can have different keys because they can be of different entity types.*

For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called **a surrogate key**, when creating a relation to correspond to the union type. The keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the relation.



Evaluate appropriateness and effectiveness of database approaches, data models and DBMS for developing an information system

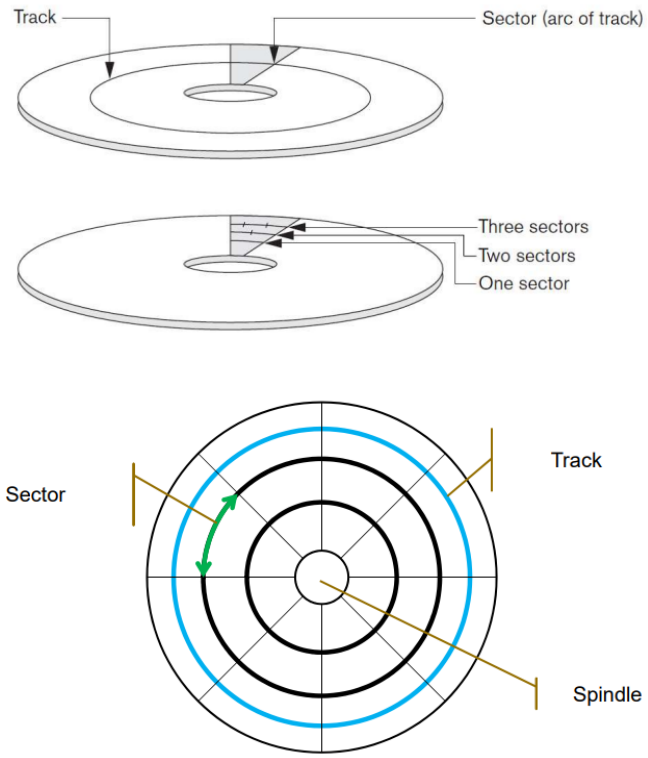
L.O.	4.3
Session	
Tag	Final

DATA STORAGE, INDEXING STRUCTURES FOR FILES

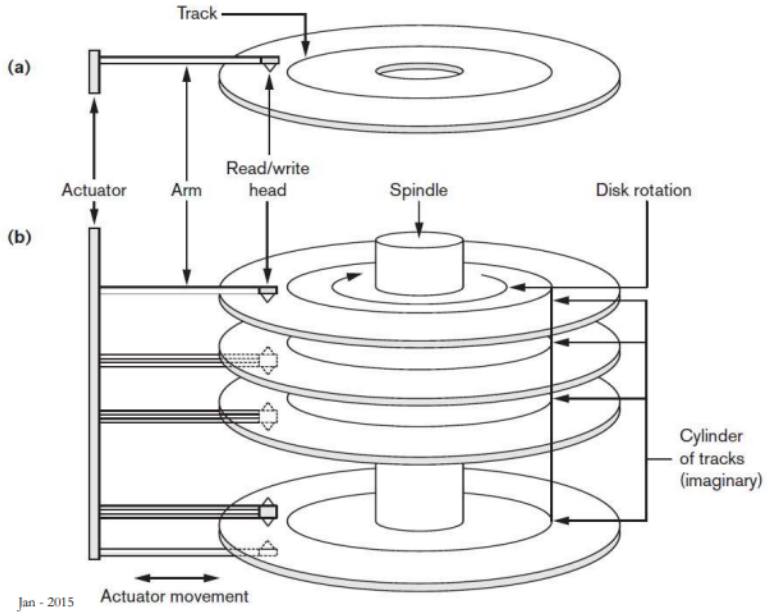
Data Storage

Disk Storage Devices

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A **disk pack** contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular **tracks** on each disk surface.
  - Track capacities vary typically from 4 to 50 Kbytes



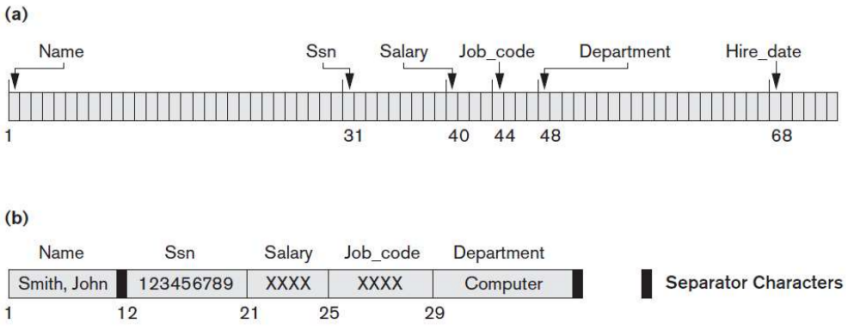
- A track is divided into smaller **blocks** or sectors.
  - because a track usually contains a large amount of information.
- A track is divided into blocks.
  - The block size B is fixed for each system.
    - Typical block sizes range from $B = 512$ bytes to $B = 4096$ bytes.
    - Whole blocks are transferred between disk and main memory for processing.
- A read-write head moves to the track that contains the block to be transferred.
  - Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
  - a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)
  - the track number or surface number (within the cylinder)
  - and block number (within track).
- Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) **rd**.
- Double buffering can be used to speed up the transfer of contiguous disk blocks.



▼ Files of Records

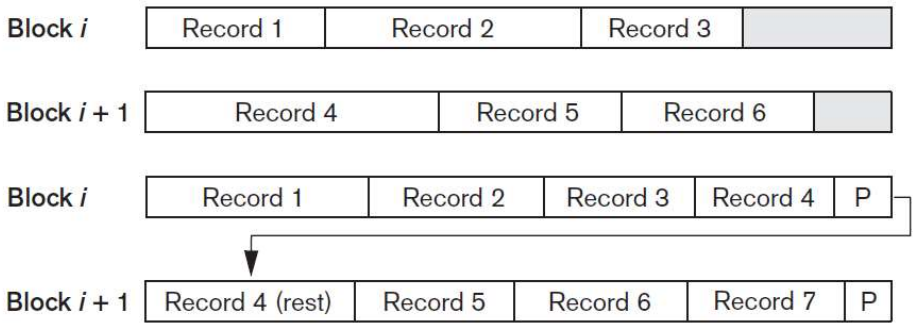
Records

- Fixed and variable length records.
- Records contain fields which have values of a particular type.
  - E.g., amount, date, time, age.
- Fields themselves may be fixed length or variable length.
- Variable length fields can be mixed into one record:
  - Separator characters or length fields are needed so that the record can be “parsed”.



Blocking

- **Blocking:** refers to storing a number of records in one block on the disk
- **Blocking factor (bfr):** refers to the number of records per block.
- There may be empty space in a block if an integral number of records do not fit in one block.
- **Spanned Records:** refer to records that exceed the size of one or more blocks and hence span a number of blocks.



Files of Records

- A **file** is a sequence of records, where each record is a collection of data values (or data items).
- A **file descriptor** (or **file header**) includes information that describes the file, such as the field names and their data types, and the addresses of the file blocks on disk.
- Records are stored on disk blocks.
- The **blocking factor bfr** for a file is the (average) number of file records stored in a disk block.
- A file can have **fixed-length records** or **variable length records**.
- File records can be **unspanned** or **spanned**:
  - **Unspanned**: no record can span two blocks
  - **Spanned**: a record can be stored in more than one block
- The physical disk blocks that are allocated to hold the records of a file can be contiguous, linked, or indexed.
- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.
- Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**.
  - Usually spanned blocking is used with such files.

▼ Operations on Files

- **OPEN:** Reads the file for access, and associates a pointer that will refer to a current file record at each point in time.
- **FIND:** Searches for the first file record that satisfies a certain condition, and makes it the current file record.
- **FINDNEXT:** Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
- **READ:** Reads the current file record into a program variable.
- **INSERT:** Inserts a new record into the file, and makes it the current file record.
- **DELETE:** Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.

- **MODIFY:** Changes the values of some fields of the current file record.
- **CLOSE:** Terminates access to the file.
- **REORGANIZE:** Reorganizes the file records. For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- **READ_ORDERED:** Read the file blocks in order of a specific field of the file.

▼ Unordered Files & Ordered Files & Hashed Files

Unordered files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
  - This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records.

Ordered files

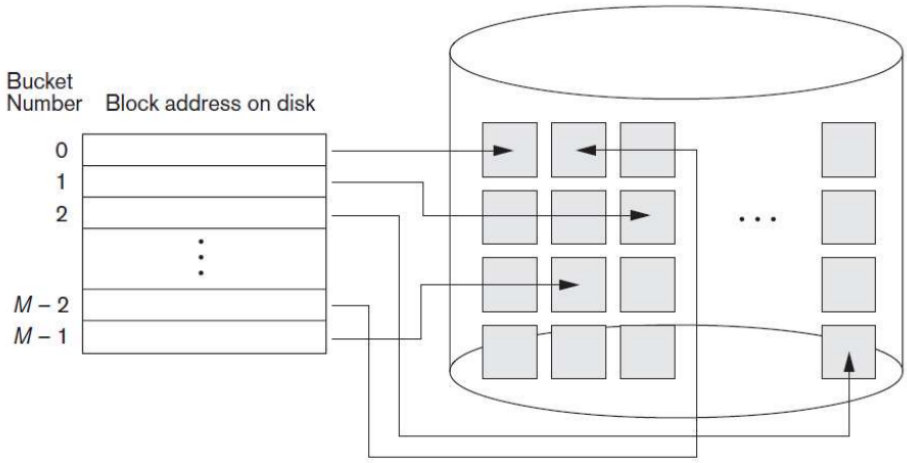
- Also called a **sequential** file.
- File records are kept sorted by the values of an ordering field.
- Insertion is expensive: records must be inserted in the correct order.
  - It is common to keep a separate unordered overflow (or transaction) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its ordering field value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

Table 17.2 Average Access Times for a File of *b* Blocks under Basic File Organizations

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

Hashed files

- Hashing for disk files is called **External Hashing**.
- The file blocks are divided into M equal-sized **buckets**, numbered $\text{bucket}_0, \text{bucket}_1, ..., \text{bucket}_{M-1}$.
  - Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the **hash key** of the file.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the **hashing function**.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together



- There are numerous methods for **collision resolution**, including the following (**DSA**):
  - **Open addressing:** Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.

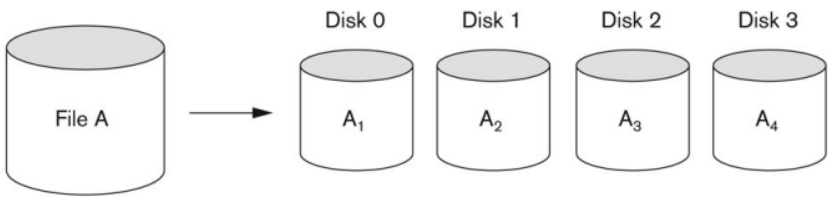
	0	1	2	3	4	5	6
h(K) = K mod 7		1		3	11		6
Insert 8		1	8	3	11		6
Insert 15		1	8	3	11	15	6
Insert 13	13	1	8	3	11	15	6

  - **Chaining**:

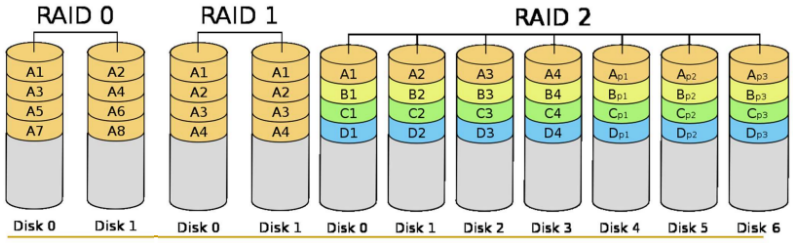
        - Various overflow locations are kept: extending the array with a number of overflow positions.
        - A pointer field is added to each record location.
        - A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
    - **Multiple hashing:**
        - The program applies a second hash function if the first results in a collision.
        - If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.
- To reduce overflow records, a hash file is typically kept $70 - 80\%$ full.
- The hash function h should distribute the records uniformly among the buckets; otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of static external hashing:
    - Fixed number of buckets M is a problem if the number of records in the file grows or shrinks.
    - Ordered access on the hash key is quite inefficient (requires sorting the records).

▼ RAID Technology

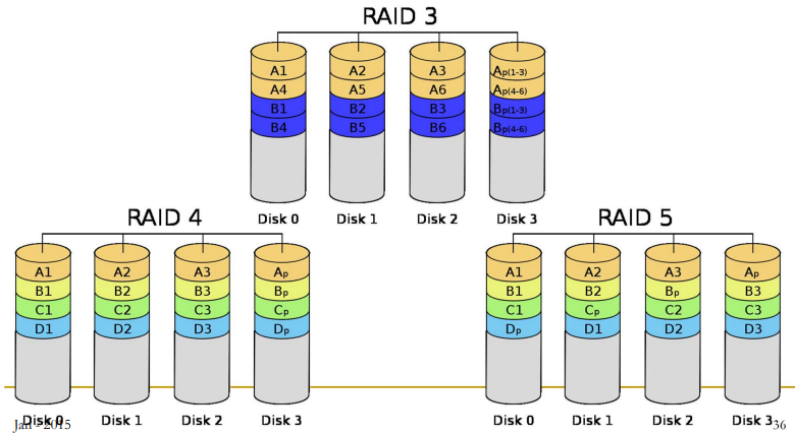
- Secondary storage technology must take steps to keep up in performance and reliability with processor technology.
- A major advance in secondary storage technology is represented by the development of RAID, which originally stood for **Redundant Arrays of Inexpensive Disks**
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.
- A natural solution is a large array of small independent disks acting as a single higher-performance logical disk.
- A concept called **data striping** is used, which utilizes *parallelism* to improve disk performance.
- Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk.



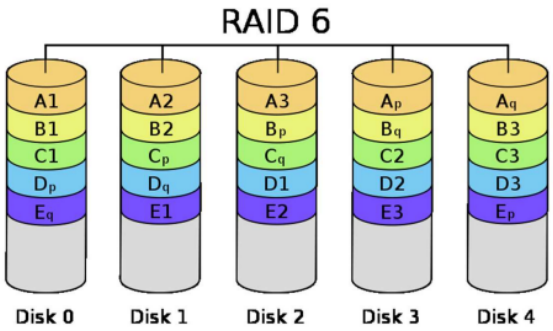
- Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.
  - **Raid level 0** has no redundant data and hence has the best write performance.
  - **Raid level 1** uses mirrored disks.
  - **Raid level 2** uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.



  - **Raid level 3** uses a single parity disk relying on the disk controller to figure out which disk has failed.
  - **Raid levels 4 and 5** use block-level data striping, with level 5 distributing data and parity information across all disks.



  - **Raid level 6** applies the so-called P + Q redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.



- Different raid organizations are being used under different situations:
  - Raid level 1 (mirrored disks)is the easiest for rebuild of a disk from other disks.
    - It is used for critical applications like logs.
  - Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
  - Raid level 3 (single parity disks relying on the disk controller to figure out which disk has failed) and level 5 (block-level data striping) are preferred for large volume storage, with level 3 giving higher transfer rates.
  - Most popular uses of the RAID technology currently are: Level 0 (with striping), Level 1 (with mirroring) and Level 5 with an extra drive for parity.
  - Design decisions for RAID include – level of RAID, number of disks, choice of parity schemes, and grouping of disks for block-level striping.

▼ Storage Area Networks

- The demand for higher storage has risen considerably in recent times.
- Organizations have a need to move from a static fixed data center oriented operation to a more flexible and dynamic infrastructure for information processing.
- Thus they are moving to a concept of Storage Area Networks (SANs).
  - In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.
- This allows storage systems to be placed at longer distances from the servers and provide different performance and connectivity options.
- Advantages of SANs are:
  - Flexible many-to-many connectivity among servers and storage devices using fiber channel hubs and switches.
  - Up to 10km separation between a server and a storage system using appropriate fiber optic cables.

    - Better isolation capabilities allowing nondisruptive addition of new peripherals and servers.
  - SANs face the problem of combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware.

Indexing Structures for Files

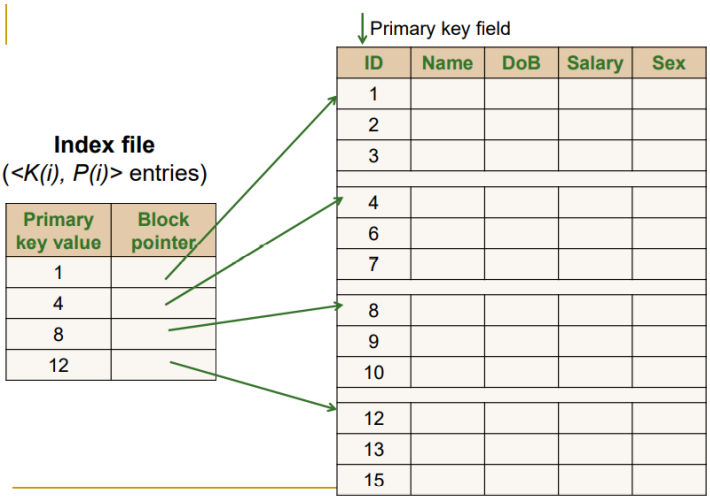
Index as access path

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries **<field value, pointer to record>**, which is ordered by field value
- The index is called an **access path** on the field.
- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.
- A binary search on the index yields a pointer to the file record.
- Indexes can also be characterized as dense or sparse:
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - A **sparse** (or **nondense**) **index**, on the other hand, has index entries for only some of the search values.

Types of Single-level Ordered Indexes

Primary Index

- Defined on an **ordered data file**.
  - The data file is ordered on a **key field**.
- One index entry for each block in the data file
  - First record in the block, which is called the *block anchor.*
- A similar scheme can use the last record in a block.



Number of index entries?

ANS: Number of blocks in data file.

Dense of Nondense?

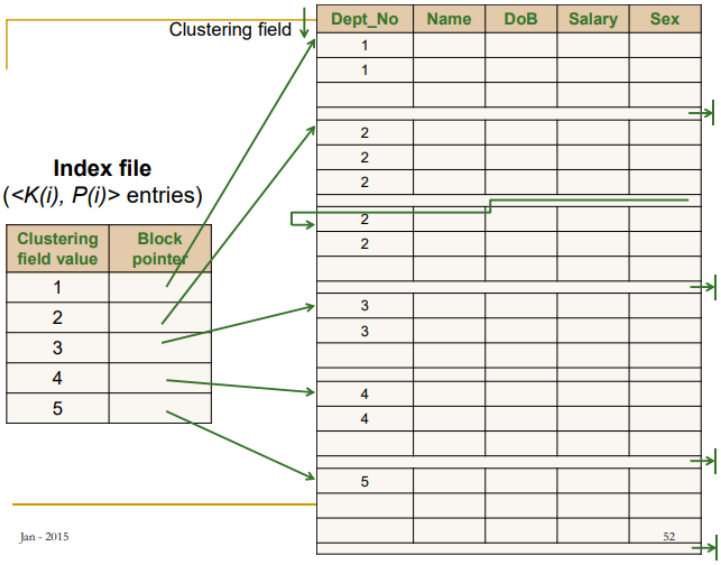
ANS: Nondense

- An ordered file with r = 300,000 records
- A disk with block size B = 4,096 bytes
- File records are of fxed size and are unspanned, with record length R = 100 bytes
- $bfr = \lfloor (B/R) \rfloor = \lfloor (4{,}096/100) \rfloor = 40$ records per block
- $b = \lceil (r/bfr) \rceil = \lceil (300{,}000/40) \rceil = 7{,}500$ blocks
- A binary search on the data fle would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 7{,}500) \rceil = 13$ block accesses

- Suppose that
  - The ordering key feld of the fle is V = 9 bytes long
  - A block pointer is P = 6 bytes long
- Construct a primary index
  - The size of each index entry is $R_i = (9 + 6) = 15$ bytes
  - The blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor = 273$ entries per block
  - The total number of index entries r_i is equal to the number of blocks in the data fle, which is 7,500
  - The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (7{,}500/273) \rceil = 28$ blocks
- Searching
  - Binary search on the index fle would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 28) \rceil = 5$ block accesses
  - A total of 5 + 1 = 6 block accesses

▼ Clustering Index

- Defined on an **ordered data file**.
  - The data file is ordered on a **non-key field.**
- One index entry **each distinct value** of the field.
  - The index entry points to the **first data block** that contains records with that field value

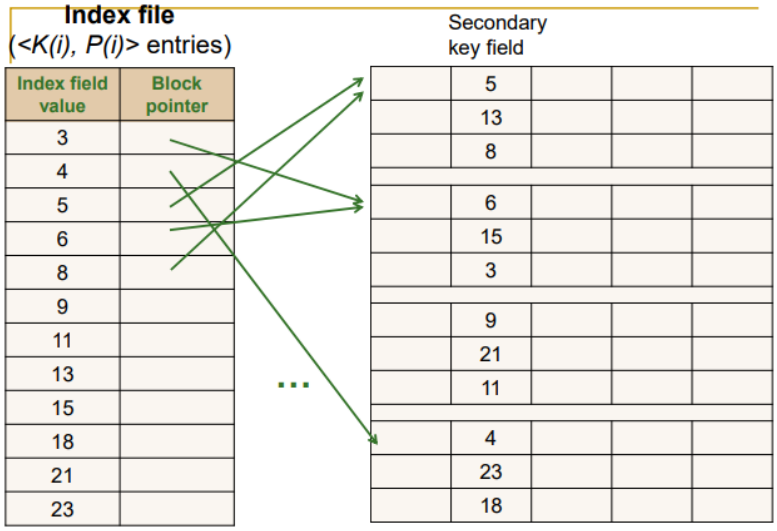


- At most **one primary index or one clustering index but not both.**

- r = 300,000 records, B = 4,096 bytes
- The file is ordered by the attribute Zipcode and there are 1,000 zip codes in the file
- R_i = 5-byte Zipcode and 6-byte block pointer
- $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/11) \rfloor = 372$ index entries per block
- r_i = 1000 index entries of the clustering index
- $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (1{,}000/372) \rceil = 3$ blocks
- A binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 3) \rceil = 2$ block accesses

▼ Secondary index

- A secondary index provides a secondary means of accessing a file.
  - The data file is **unordered** on indexing field.
- Indexing field:
  - secondary key (unique value)
  - nonkey (duplicate valu
- The index is an ordered file with two fields.
  - The first field: indexing field.
  - The second field: block pointer or record pointer.
- There can be **many** secondary indexes for the same file.



SECONDARY INDEX ON KEY FIELD

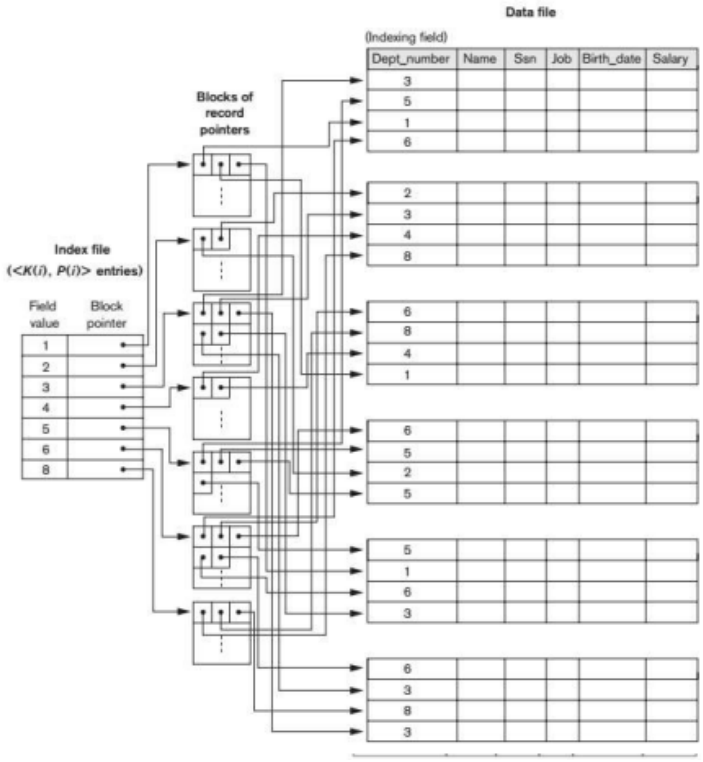
Number of index entries?

ANS: Number of record in data file

Dense of Nondense?

ANS: Dense

- Structure of Secondary index on nonkey field?
  - **Option 1:** include **duplicate index entries** with the same K(i) value - one for each record
  - **Option 2:** keep a **list of pointers** <P(i, 1), ..., P(i, k)> in the index entry for K(i).
  - **Option 3:**



- More commonly used.
- One entry for each distinct - index field value + an **extra level of indirection** to handle the multiple pointers.

SECONDARY INDEX ON NON-KEY FIELD

Number of index entries?

ANS: Number of record in data file or number of distinct index field values

Dense of Nondense?

ANS: Dense/nondense

EXAMPLE 3

- r = 300,000, R = 100 bytes, B = 4,096 bytes, b = 7,500 blocks
- A non-ordering key feld of the file that is V = 9 bytes long
- A linear search on the file would require $b/2$ = 7,500/2 = 3,750 block accesses on the average
- A secondary index
  - A block pointer is P = 6 bytes long, so each index entry is R_i = (9 + 6) = 15 bytes
  - $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor$ = 273 index entries per block
  - $b_i = \lceil (r/bfr_i) \rceil = \lceil (300{,}000/273) \rceil$ = 1,099 blocks
  - A binary search on this secondary index needs $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 1{,}099) \rceil$ = 11 block accesses
  - Total block accesses: 11 + 1 = 12

Summary

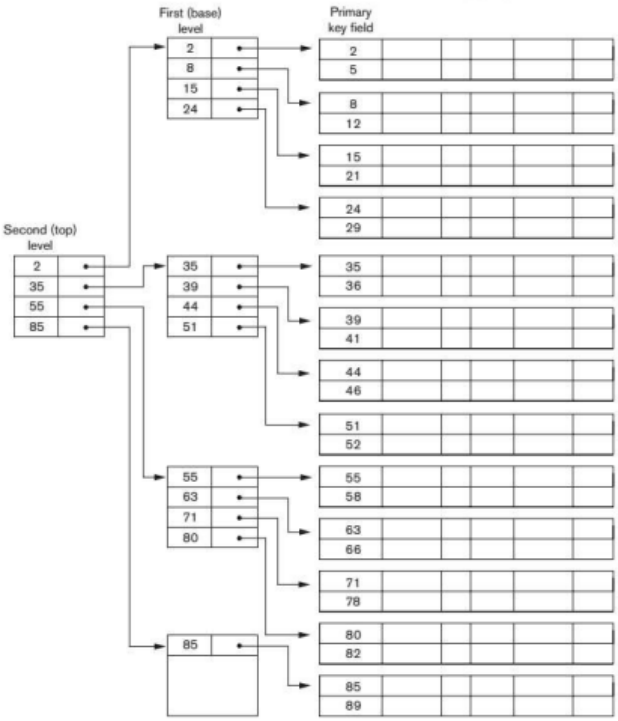
Table 18.2 Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

▼ Multi-Level Indexes

- Because a single-level index is an ordered file, we can **create a primary index to the index itself**.
  - The original index file is called the first-level index and the index to the index is called the second-level index.
- We can repeat the process, creating a third, fourth,..., top level **until all entries of the top level fit in one disk block**.
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of more than one disk block.



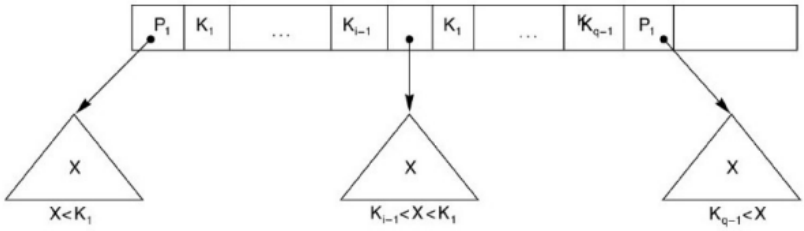
From example 3: bfr_i = 273 index entries per block (fo) and b_1 = 1,099 blocks

- $b_2 = \lceil (b_1/fo) \rceil = \lceil (1{,}099/273) \rceil = 5$ blocks
- $b_3 = \lceil (b_2/fo) \rceil = \lceil (5/273) \rceil = 1$ block

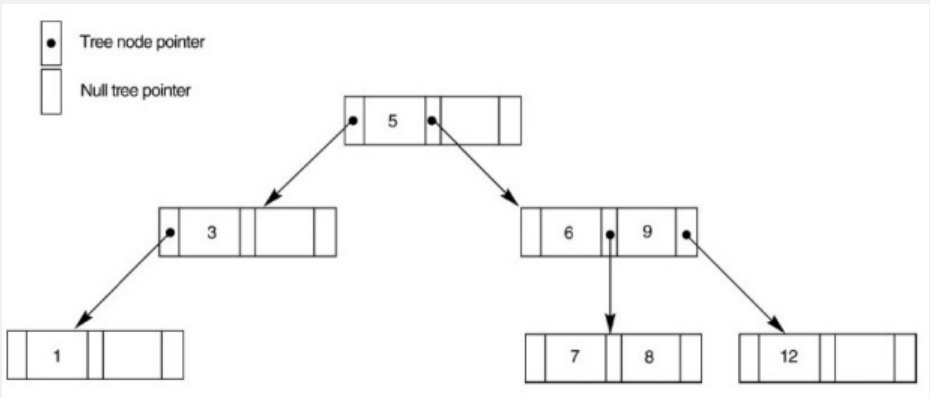
Top level of the index: t = 3

Total block accesses: t + 1 = 3 + 1 = 4 block accesses

- Such a multi-level index is a form of *search tree*.
- However, insertion and deletion of new index entries is a severe problem because every level of the index is an **ordered file**.



✅ Order $p = 3$



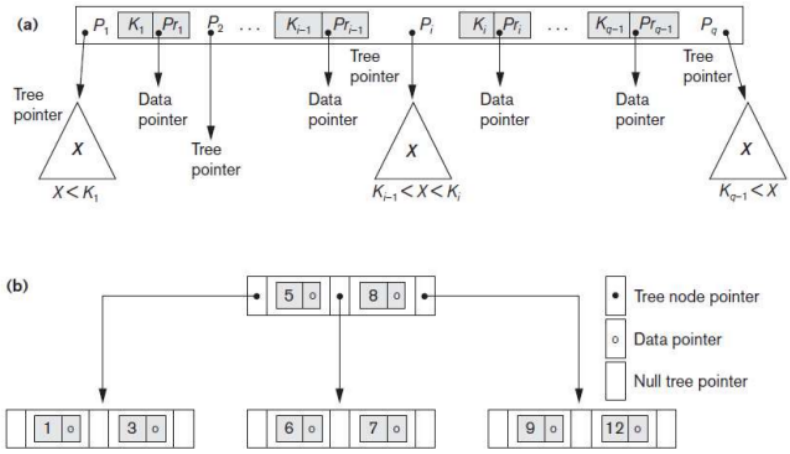
▼ Dynamic Multilevel Indexes Using BTrees and B+-Trees

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem.
  - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block.
- Each node is kept between half-full and completely full.

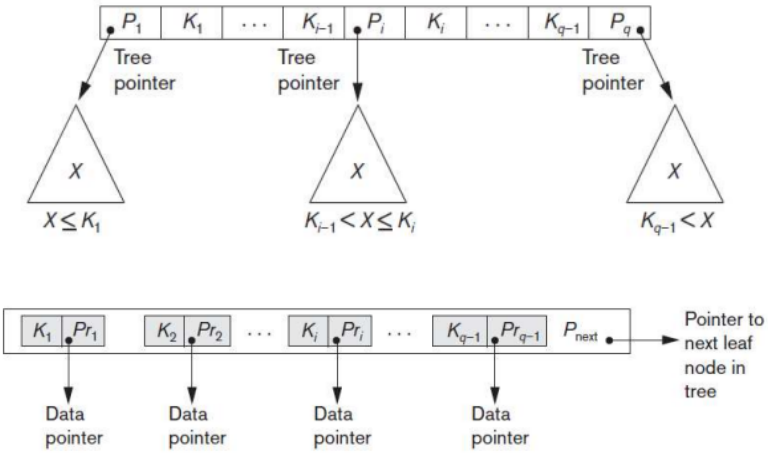
- An insertion into a node that is not full is quite efficient.
  - If a node is full, the insertion causes a split into two nodes.
- Splitting may propagate to other tree levels.
- A deletion is quite efficient if a node does not become less than half full.
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes.

Difference between B-Tree and B+-Tree

- In a B-Tree, pointers to data records exist at all levels of the tree.

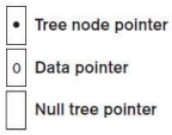


- In a B+-Tree, all pointers to data records exist at the leaf-level nodes.



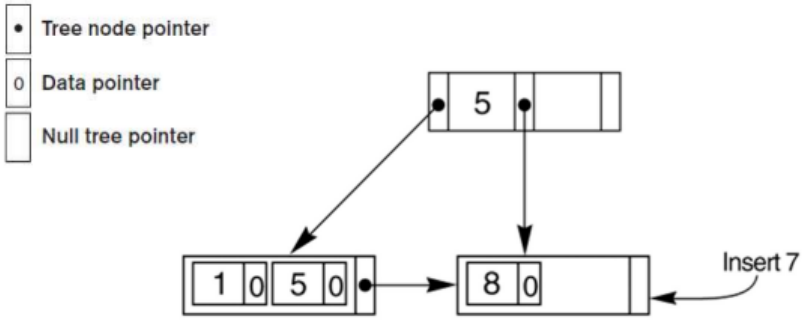
- A B+-Tree can have less levels (or higher capacity of search values) than the corresponding B-tree

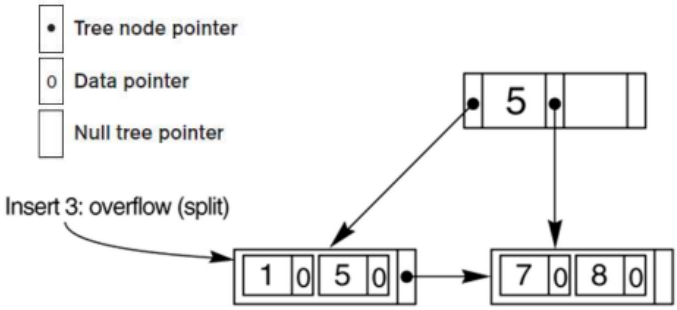
▼ Insertion of B+-Tree

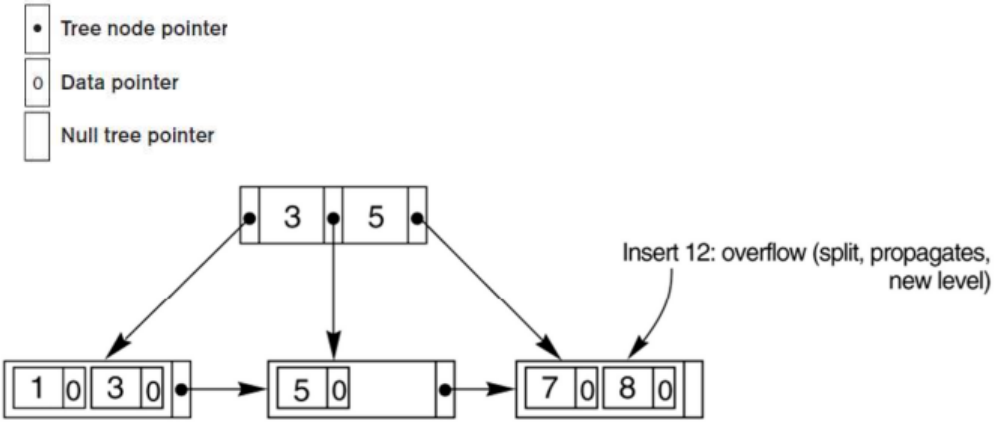


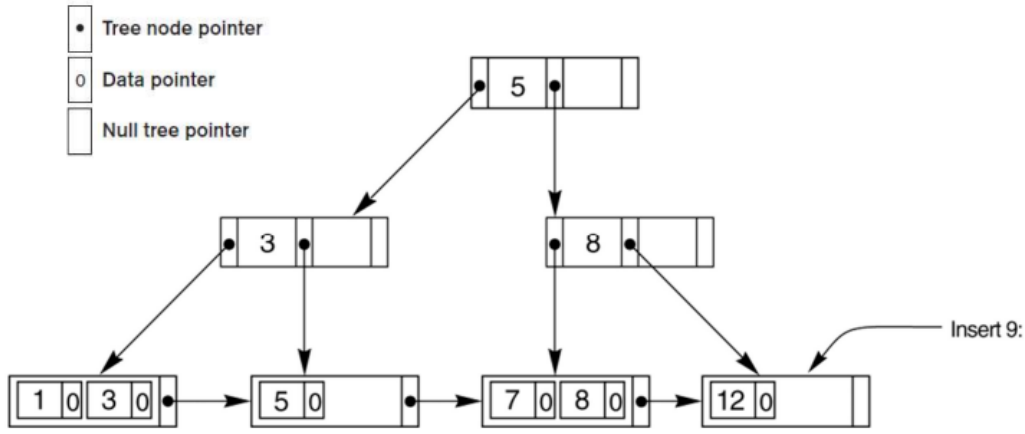
p = 3 and p_{leaf} = 2

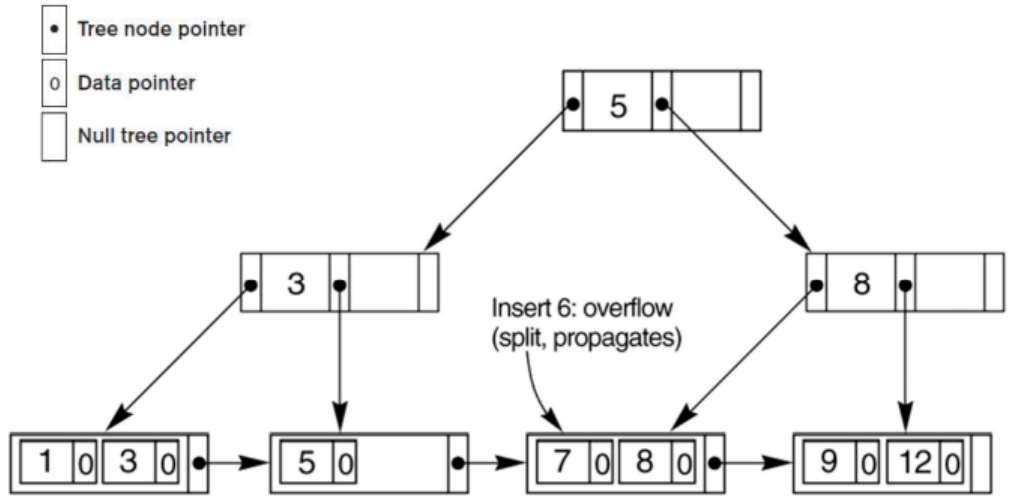
Insertion Sequence: *8, 5,* 1, 7, 3, 12, 9, 6

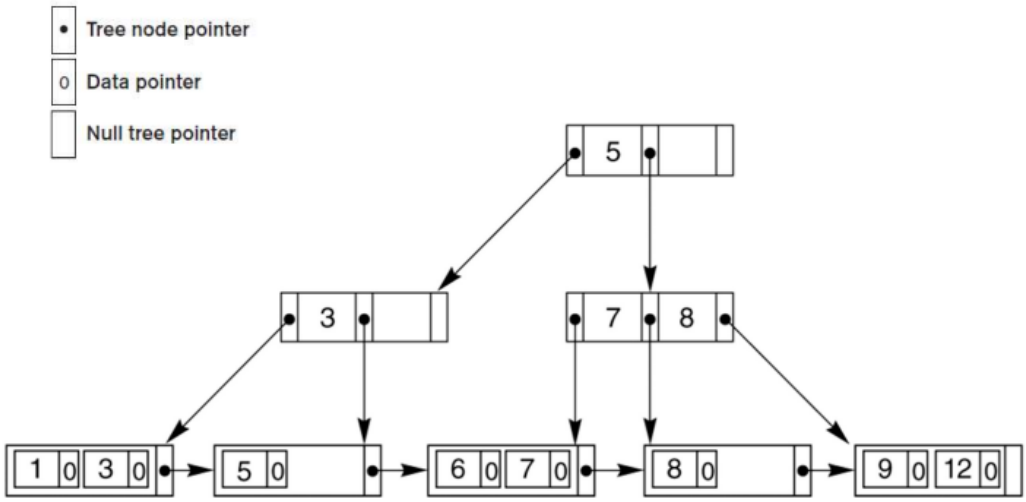










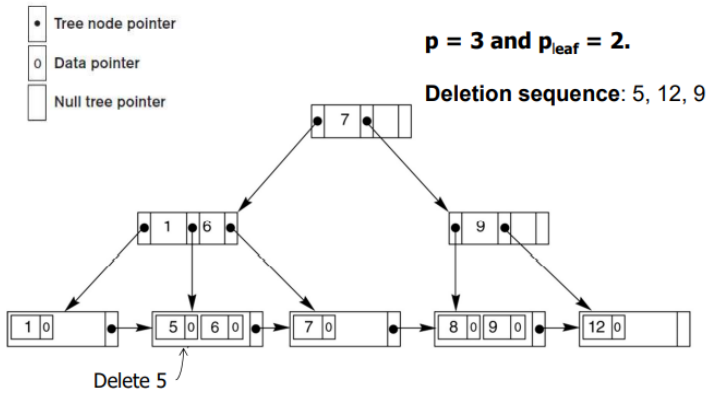


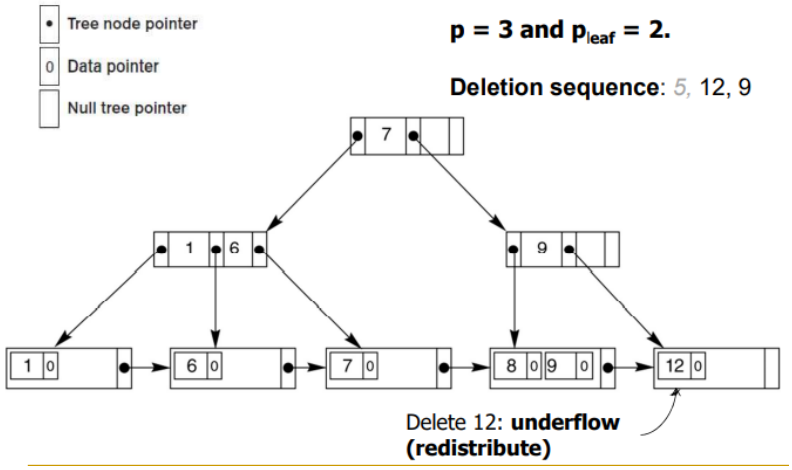
▼ Deletion of B+-Tree

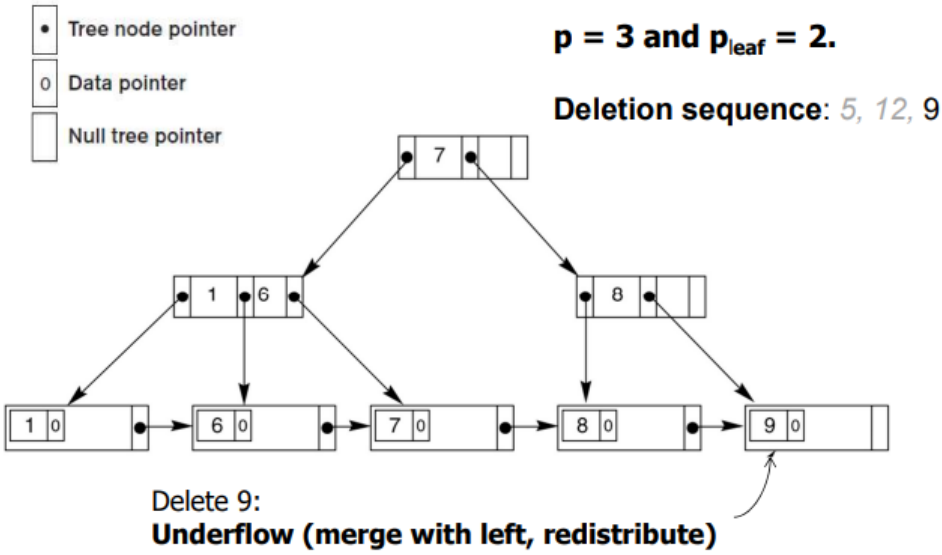
- Start at root, find leaf L where entry belongs.
- Remove the entry:
  - If L is at least half-full, done!
  - If L has fewer entries than it should,

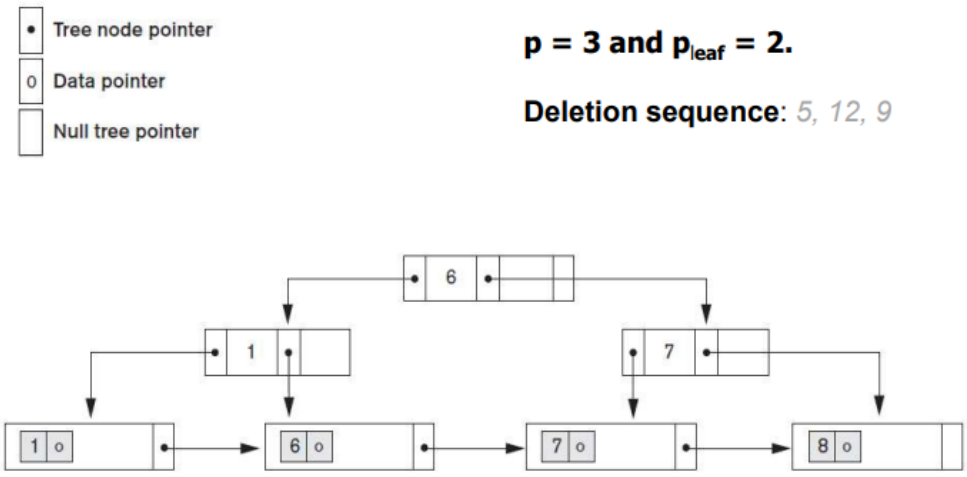
- Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
- If re-distribution fails, merge L and sibling.

- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.









Indexes in ORACLE

- **B-tree indexes**: Standard index type
  - Index-organized tables: The data is itself the index.
  - **Reverse key indexes:** -=he bytes of the index key are reversed. For example, 103 is stored as 301. The reversal of bytes spreads out inserts into the index
    over many blocks.
  - **Descending indexes:** This type of index stores data on a particular column or columns in descending order.
  - **B-tree cluster indexes:** is used to index a table cluster key. Instead of pointing to a row, the key points to the block that contains rows related to the cluster key.
- Bitmap and bitmap join indexes: an index entry uses a bitmap to point to multiple rows. A bitmap join index is a bitmap index for the join of two or more tables.
- Function-based indexes:
  - Includes columns that are either transformed by a function, such as the **UPPER function**, or included in an expression.
  - B-tree or bitmap indexes can be function-based.
- Application domain indexes: customized index specific to an application.
- Simple create index syntax:

```
CREATE [ UNIQUE | BITMAP ] INDEX [schema.] <index_name>
ON [schema.] <table_name> (column [ ASC | DESC ] [ , column [ASC | DESC ] ] ...)
[REVERSE];
```

```
CREATE INDEX ord_customer_ix ON ORDERS (customer_id);

CREATE INDEX emp_name_dpt_ix ON HR.EMPLOYEES(last_name ASC, department_id DESC);

CREATE BITMAP INDEX emp_gender_idx ON EMPLOYEES (Sex);
```

```
CREATE BITMAP INDEX emp_bm_idx
ON EMPLOYEES (JOBS.job_title)
FROM EMPLOYEES, JOBS
WHERE EMPLOYEES.job_id = JOBS.job_id;
```

Explain basic concepts: data, db, db sys, architecture of a db sys and components of a db sys

L.O.	1.1
Session	1
Tag	Final Prsntn

Data

recorded facts, unorganized, and without context

Database

collection of logically related data with the description of the data

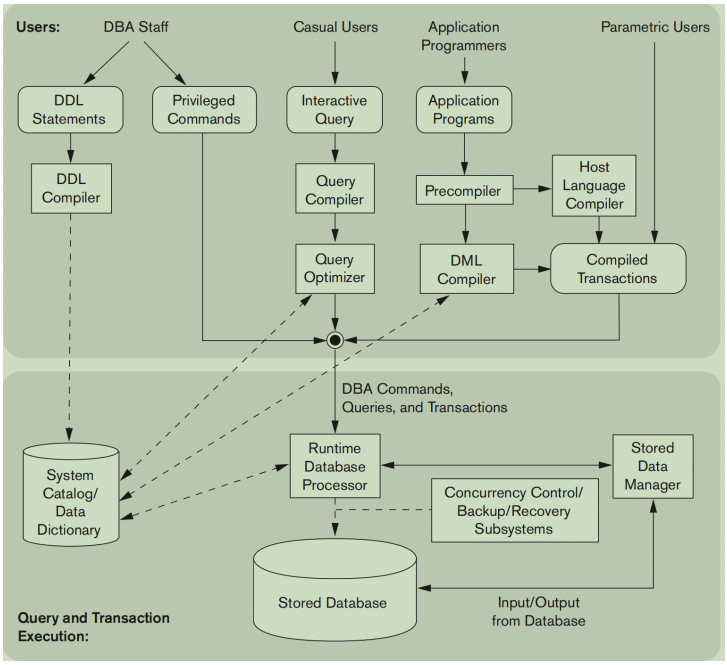
Database systems

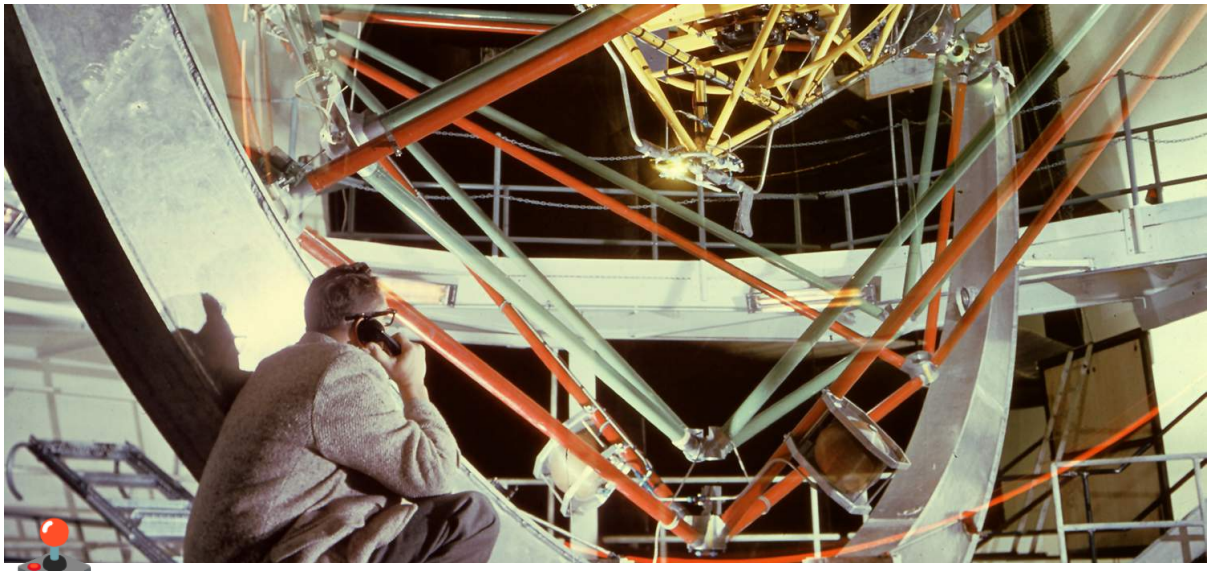
create and maintain Database

Architecture of a database system

External → Conceptual → Internal

Components of a database system





Explain db design methodologies, ER model, data normalization

L.O.	1.3
Session	2 3
Tag	Ass. ClWrk Final Prsntn

▼ DB Design methodologies

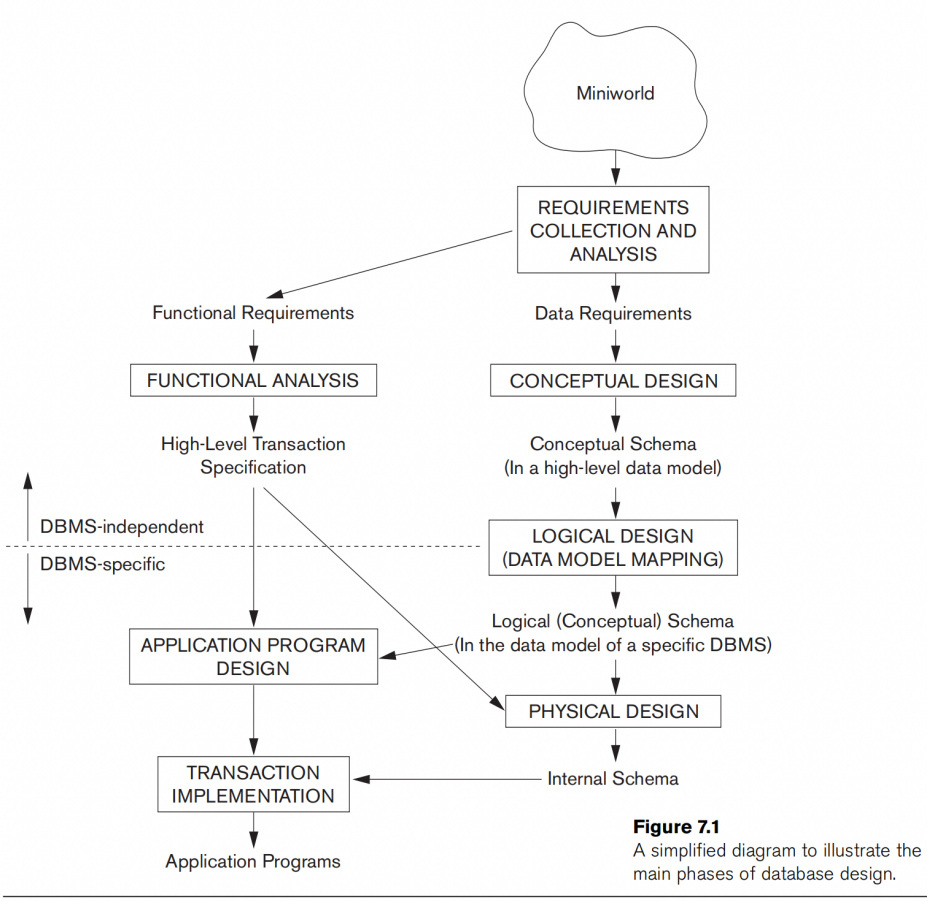


Figure 7.1
A simplified diagram to illustrate the main phases of database design.

Phase	Summary	Elaboration
Conceptual	Entities and relationships among them	lets you know the entities and the relation between them. The entities & relations are defined here.
Logical	Schema (ER, Relational,…)	provides details about the data to the physical phase. The logical phase gives ER Diagram, data dictionary, schema, etc that acts as a source for the physical design process.
Physical	DBMS-specifics	designer decides on how the database will be implemented
Implementation		

▼ ER model

💽 Basic concepts of ER Modeling

Assume the ternary relationship has participating entities A, B and C (for degree > 3 it gets pretty hairy).

The way to read the relationship is to **always isolate 2 out of the 3 participating entities and see how they relate towards the third one**. And you need to do this *for all possible pairs*.

More precisely: the 2 entities that you pair each time, need to be considered as **"one of" for each one of them** and the question to answer is ***"how many of the third one can correspond to this pair?"***

Abstract example

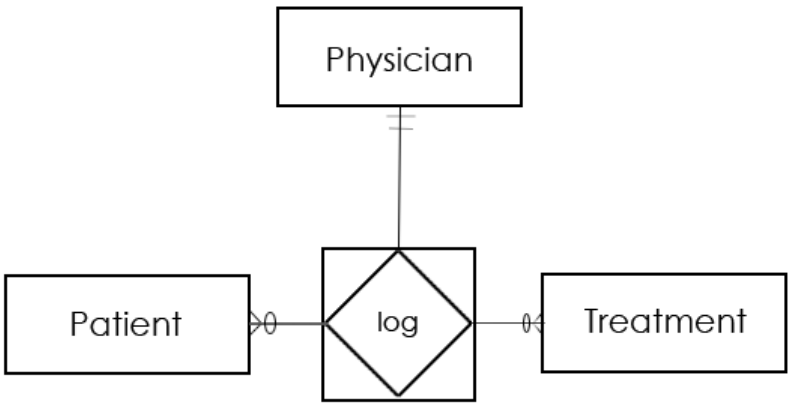
`"One of A and one of B can {have/associate with/belong to} X? of C"`. We need to use our knowledge of our business model to answer if `X?` should be `1` or `N`. This is the cardinality to assign to the ternary relationship on the edge that connects the ternary relationship with the entity `C`.

This phrase has to be reformed for all possible combinations (not permutations, since the order of pairing doesn't matter). So to answer the question `How many pairs are there?`, simple math dictates that the possible ways to combine 3 things in groups of 2 is:

`3!/(2!*(3-2)!) = 3`.

So all the possible phrases to answer using our business model are:

- `One of A and one of B can {have/associate with/belong to} ?X? of C`
- `One of A and one of C can {have/associate with/belong to} ?Y? of B`
- `One of B and one of C can {have/associate with/belong to} ?Z? of A`



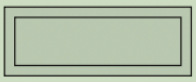
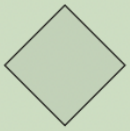
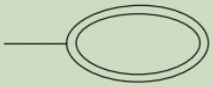
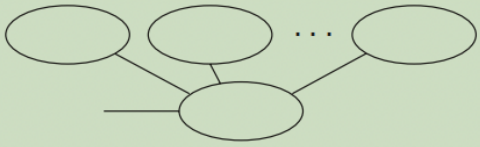
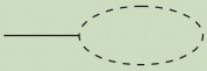
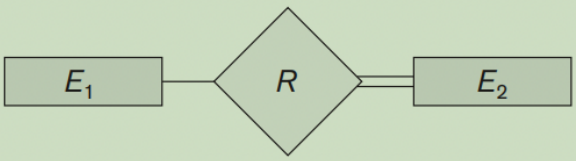
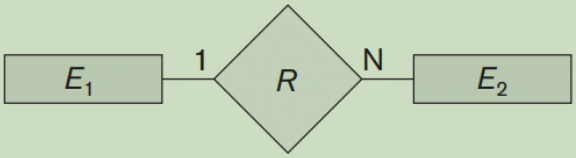
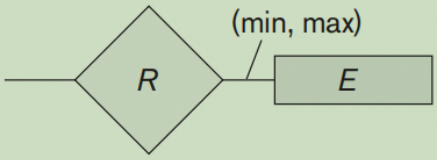
The realities of our business model that led to this image are:

- `1 Physician with 1 specific Patient can log M Treatments`
- `1 Physician logs 1 specific Treatment for N Patients`
- `1 Patient is logged 1 specific Treatment by 1 Physician`

So the ternary relationship `log` is an M-N-1 relationship between the participating entities `Treatment-Patient-Physician` (in this order).

Problems

Fan 🍃	Chasm ⛰️
Where a model represents a relationship between entity types, but the **pathway** between certain instances is **ambiguous**	Where a model represents a relationship btween entity types, but the **pathway** between certain instances **does not exist**

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Indentifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of E_2 in R
	Cardinality Ratio 1 : N for E_1:E_2 in R
	Structural Constraint (min, max) on Participation of E in R

▼ EER (Enhanced ER)

💎 Basic concepts of EER (Enhanced Entity-Relational Modelling)

Specialization	Generalization
Identify distinguishing characteristics of members of an entity	Identify common characteristics
top-down	bottom-up

Particip. Constraints

Mandatory	Optional
Every member of superclass must be member of a subclass	Member of a superclass need not be a member of a subclass

Disjoint Constraints

Disjoint	Overlapping
Entity instance can be member of ***one*** subclass only.	Entity instance can be member of $\geq$ ***one*** subclasses

	shared subclass	category
superclass	intersection $\cap$	union $\cup$

▼ Data Normalization

😎 Basic concepts of Normalization

The normalization process aims to minimize data duplications, avoid errors during data modifications and simplify data queries from the database. The three fundamental normalization forms are known as:

▼ First Normal Form (1NF)

> **Data Atomicity**: Ensuring that there is only one single instance value per column field

Enforce the Data Atomicity rule and remove duplication of data

Loading... This Course Video Transcript Back-end developers write applications that end-users use to interact with databases. Some common tasks that end-users carry out using

https://www.coursera.org/learn/intro-to-databases-back-end-development/lecture/YEH1G/first-normal-form-1nf

▼ Second Normal Form (2NF)

Loading... This Course Video Transcript Back-end developers write applications that end-users use to interact with databases. Some common tasks that end-users carry out using

https://www.coursera.org/learn/intro-to-databases-back-end-development/lecture/dtFfc/second-normal-form-2nf

- Is in 1NF
- Every non-primary-key attribute is fully functional dependent on the primary key: $\exists y \forall x (\text{PrimaryKey}(y) \wedge \text{NonPKey}(x) \implies \text{FullFunctDep}(x, y))$

Full Functional dependency: B is fully dependent on A if B is not functionally dependent on any proper subset of A

Partial dependency: B is partially dependent on A when there is some attribute that can be removed from A and yet the dependency still holds.

▼ Third Normal Form (3NF)

https://www.coursera.org/learn/intro-to-databases-back-end-development/lecture/ctzx9/third-normal-form-3nf

- Is in 1NF
- Is in 2NF
- No non-primary-key attribute is transitively dependent on the primary key: $\forall y (\text{PrimaryKey}(y) \implies \forall x (\text{NonPKey}(x) \implies \neg \text{TransDep}(x, y)))$

Transitive dependency: $\forall A \forall B \forall C (A \to B \wedge B \to C \wedge (B \nrightarrow A \vee C \nrightarrow A) \implies A \to C)$

▼ Advanced Normalization

👾 Basic concepts of Advanced Normalization

(1)	**Reflexivity:**	If B is a subset of A, then A → B
(2)	**Augmentation:**	If A → B, then A,C → B,C
(3)	**Transitivity:**	If A → B and B → C, then A → C
(4)	**Self-determination:**	A → A
(5)	**Decomposition:**	If A → B,C, then A → B and A → C
(6)	**Union:**	If A → B and A → C, then A → B,C
(7)	**Composition:**	If A → B and C → D then A,C → B,D

▼ Boyce-Codd Normal Form (BCNF)

Loading... This Course Video Transcript Database Management Essentials provides the foundation you need for a career in database development, data warehousing, or business

https://www.coursera.org/lecture/database-management/normal-forms-video-lecture-kcKrg

- More strict than 3NF
- Quite similar to 3NF
- A relation is in **Boyce-Codd Normal Form (*BCNF*)** if and only if *every determinant is a candidate key*.

> ❓ **NOTE:**
> BCNF is a stronger form of 3NF, such that every relation in BCNF is also in 3NF. However, *a relation in 3NF is not necessarily in BCNF*.

> ✅ A relation with **two attributes** will **gurantee to be a BCNF**

Explain the concept of data independence and its importance in a db sys

L.O.	4.2
Session	1
Tag	Final Prsntn

Data Independence

Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.5(a) should not be affected by changing the GRADE_REPORT file (or record type) shown in Figure 1.2 into the one shown in Figure 1.6(a). Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before. Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
A database that stores student and course information.

TRANSCRIPT

Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

(a)

COURSE_PREREQUISITES

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

(b)

Figure 1.5
Two views derived from the database in Figure 1.2. (a) The TRANSCRIPT view. (b) The COURSE_PREREQUISITES view.

Figure 1.6
Redundant storage of Student_name and Course_name in GRADE_REPORT. (a) Consistent data. (b) Inconsistent record.

(a)

GRADE_REPORT

Student_number	Student_name	Section_identifier	Course_number	Grade
17	Smith	112	MATH2410	B
17	Smith	119	CS1310	C
8	Brown	85	MATH2410	A
8	Brown	92	CS1310	A
8	Brown	102	CS3320	B
8	Brown	135	CS3380	A

Physical data independence is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creat ing additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval speed of section records (Figure 1.2) by semester and year should not require a query such as *list all sections offered in fall 2008* to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

Importance

- With the help of physical data independence, we can change the storage and file structure of the database without affecting the application program ⇒ saves resources.
- Improve security of the database.
- Helps us modify data without affecting the application program.

Explain the relational data model, relational algebra, and SQL

L.O.	1.2
Session	1 4 5
Tag	Final Prsntn

Relational Data Model

▼ Characteristics of relations

Ordering of values within a tuple is important

if a tuple is a set of `(<attrib>,<value>)` ⇒ ordering is not important

Each value in a tuple is atomic ⚛ (cannot be divided into component values)

⭐ Interpretation of a relation:

- the relation schema can be represented as a declaration (blueprint)
- each tuple can be interpreted as a *fact*

▼ What is the relational model?

The relational model is built around three main concepts which are:

- Data
- Relationships
- Constraints.

It describes a database as “a collection of inter-related relations (or tables)”. It is still a dominant model used for data storage and retrieval. In essence, it is a way of organizing or storing data in a database. SQL is the language that’s used to retrieve data from a relational database.

▼ Relational Model Constraints

key	definition	note
superkey	attrib or {attribs} that uniquely id a record in a table	may contain unnecessary attribs
candidate key	a superkey such that no proper subset is a superkey within the relation. $\{\exists x(\text{Superkey}(x) \wedge \forall y(y \subset x \rightarrow \neg \text{SuperKey}(y)) \rightarrow \text{Candidate key}(x))\}$	is ***unique*** and ***irreducible***

key	definition	note
primary key	candidate key that is selected to uniquely id records in a table	
foreign key	attrib, or {attribs} that match candidate key of some (possibly same) relation.	

▼ Discuss the differences between the candidate keys and the primary key of a relation. Explain what is meant by a foreign key. How do foreign keys of relations relate to candidate keys? Give examples to illustrate your answer.

	candidate key	primary key
Difference	- there can be many. - some attributes in candidate key can contain ***NULL*** values	there is only one and only one

▼ Define the two principal integrity rules for the relational model. Discuss why it is desirable to enforce these rules.

Entity integrity: In a base relation, no attribute of a primary key can be null

Referential integrity: A foreign key must match a candidate key value of some tuple in the home relation OR be wholly *NULL*. Apart from relational integrity, integrity constraints include *required data*, *domain*, *multiplicity constraints*; other called **general constraints**.

Why is it desirable to enforce these?

It is desirable to enforce the entity integrity constraint because the primary key is used to uniquely identify table records. We cannot identify records by NULL values.

It is desirable to enforce the referential integrity constraint because: firstly, it ensures the changes made to data in one record be reflected in other related records; secondly, this constraint ensures the consistency between data records in our database. For example, suppose we have 2 relations `Staff(sid, name, branchno)` and `Branch(branchno, name)` with `Staff.branchno` as the foreign key to `Branch.branchno`. We cannot create a staff record with `branchno` B305 unless there is already a branch record with `brancho` B305. However, we should be able to create a staff record with NULL `branchno` to account for the situation where a new recruit has not been assigned a branch.

Relational Algebra

👨‍🔬 Theory of Relational Algebra

🎣 Theory of Relational Calculus

sigma = commutative

project ≠ commutative and will remove duplicate tuples

type compatible: same attr, same domain

grouping first, aggregate function later

include non-aggregate attr in group by

left join: include all unmatched rows in the left and put null for each unmatched in the right

---

▼ List the operations of relational algebra and the purpose of each.

OPERATION	NOTATION	FUNCTION
Selection	$\sigma_{predicate}(R)$	Produces a relation that contains only those tuples of R that satisfy the specified *predicate*.
Projection	$\Pi_{a_1,\ldots,a_n}(R)$	Produces a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.
Union	$R \cup S$	Produces a relation that contains all the tuples of R, or S, or both R and S, duplicate tuples being eliminated. R and S must be union-compatible.
Set difference	$R - S$	Produces a relation that contains all the tuples in R that are not in S. R and S must be union-compatible.
Intersection	$R \cap S$	Produces a relation that contains all the tuples in both R and S. R and S must be union-compatible.
Cartesian product	$R \times S$	Produces a relation that is the concatenation of every tuple of relation R with every tuple of relation S.
Theta join	$R \bowtie_F S$	Produces a relation that contains tuples satisfying the predicate *F* from the Cartesian product of R and S.
Equijoin	$R \bowtie_F S$	Produces a relation that contains tuples satisfying the predicate *F* (which contains only equality comparisons) from the Cartesian product of R and S.
Natural join	$R \bowtie S$	An Equijoin of the two relations R and S over all common attributes *x*. One occurrence of each common attribute is eliminated.
(Left) Outer join	$R \mathbin{⟕} S$	A join in which tuples from R that do not have matching values in the common attributes of S are also included in the result relation.
Semijoin	$R \triangleright_F S$	Produces a relation that contains the tuples of R that participate in the join of R with S satisfying the predicate *F*.
Division	$R \div S$	Produces a relation that consists of the set of tuples from R defined over the attributes C that match the combination of **every** tuple in S, where C is the set of attributes that are in R but not in S.
Aggregate	$_{AL}(R)$	Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list. AL contains one or more (<aggregate_function>, <attribute>) pairs.
Grouping	$_{GA\ AL}(R)$	Groups the tuples of relation R by the grouping attributes, GA, and then applies the aggregate function list AL to define a new relation. AL contains one or more (<aggregate_function>, <attribute>) pairs. The resulting relation contains the grouping attributes, GA, along with the results of each of the aggregate functions.

	Purpose	**Note**
Selection	choose a subset of tuples (rows) that satisfies a **selection condition**	
Projection	select certain columns and discard other columns	

	Purpose	Note
Cartesian product	for each of tuple in A, match it with each of tuple in B. Used to select all matching tuples from both A and B	
Union	The result of this operation is a relation including all rows in either A or B or both A and B	⚠️union-compatible
Set Difference	when we want tuples that only relation A has	⚠️union-compatible

▼ What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?

Union compatibility: when 2 relations share the same number of attributes and each pair or corresponding attributes share the same domain

Because UNION, INTERSECTION, DIFFERENCE are all binary set operations, they require their operand relations to be union-compatible. In fact, these operations require direct comparison between tuples

▼ Discuss some types of queries for which renaming of attributes is necessary in order to specify the query unambiguously.

When a query has an NATURAL JOIN operation than renaming foreign key attribute is necessary, if the name is not already same in both relations, for operation to get executed. In EQUIJOIN after the operation is performed there are two attributes that have same values for all tuples. These are attributes which have been checked in condition. In NATURAL JOIN one of them has been removed only single attribute is there.

▼ Discuss the various types of *inner join* operations?

	Representation	Definition	Note
Θ-join	$A \bowtie_F B = \sigma_F(A \times B)$	- a relation whose tuples satisfy condition F from the Cartesian product of A and B: $F \equiv A.a_i\ \theta\ B.b_i$ ☝ $\theta \in \{>, \geq, <, \leq, =, \neq\}$	
Equijoin	$A \bowtie_{A.a_i = B.b_i} B$	- a relation whose tuples satisfy condition $A.a_i = B.b_i$	particular type of Θ-join
Natural join	$A \bowtie B$	Equijoin over all common attributes x. One occurence of each common attribute is eliminated from the result.	
Semijoin	$A \rhd_F B$	$\Pi_{(\text{all attributes in A})}(A \bowtie_F B)$	

▼ SQL

▼ Some of The Most Important SQL Commands

- `SELECT` - extracts data from a database
- `UPDATE` - updates data in a database

- `DELETE` - deletes data from a database
- `INSERT INTO` - inserts new data into a database
- `CREATE DATABASE` - creates a new database
- `ALTER DATABASE` - modifies a database
- `CREATE TABLE` - creates a new table
- `ALTER TABLE` - modifies a table
- `DROP TABLE` - deletes a table
- `CREATE INDEX` - creates an index (search key)
- `DROP INDEX` - deletes an index

▼ The SELECT Statement

> The `SELECT` statement is used to select data from a database. The data returned is stored in a result table, called the result-set.

```
SELECT [column1, column2, ...]
FROM [table_name];
```

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table:

```
SELECT * FROM [table_name];
```

Select all the *different* values from a table name:

```
SELECT DISTINCT [column1, column2,...] FROM [table_name];
```

▼ The WHERE Statement

The `WHERE` clause is used to filter records. It is used to extract only those records that fulfill a specified condition.

```
SELECT [column1, column2, ...]
FROM [table_name]
WHERE [condition];
```

Operators in The WHERE Clause

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal

Operator	Description
<=	Less than or equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

▼ THE AND, OR and NOT OPERATORS

> The `WHERE` clause can be combined with `AND`, `OR`, and `NOT` operators.
>
> The `AND` and `OR` operators are used to filter records based on more than one condition:
>
> - The `AND` operator displays a record if all the conditions separated by `AND` are TRUE.
> - The `OR` operator displays a record if any of the conditions separated by `OR` is TRUE.
>
> The `NOT` operator displays a record if the condition(s) is NOT TRUE.

AND Syntax

```
SELECT [column1, column2, ...]
FROM [table_name]
WHERE [condition1] AND [condition2] AND [condition3 ...];
```

OR Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

NOT Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

▼ The ORDERED BY Keyword

> The `ORDER BY` keyword is used to sort the result-set in ascending or descending order.
>
> The `ORDER BY` keyword sorts the records in ascending order by default. To sort the records in descending order, use the `DESC` keyword.

ORDER BY Syntax

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

ORDER BY Several Columns Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName:

```
SELECT * FROM Customers
ORDER BY Country, CustomerName;
```

▼ SQL INSERT INTO Statement

It is possible to write the `INSERT INTO` statement in two ways:

1. **Specify both the column names and the values to be inserted:**

```
INSERT INTO table_name (column1, column2, column3, ...)
 VALUES (value1, value2, value3, ...);
```

2. **If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query**. However, make sure the order of the values is in the same order as the columns in the table. Here, the `INSERT INTO` syntax would be as follows:

```
INSERT INTO table_name
 VALUES (value1, value2, value3, ...);
```

▼ SQL NULL Operators

> The `IS NULL` operator is used to test for empty values (NULL values).

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

> The `IS NOT NULL` operator is used to test for non-null values.

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

▼ THE UPDATE Statement

> The `UPDATE` statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

▼ The DELETE Statement

> The `DELETE` statement is used to delete existing records in a table.

```
DELETE FROM table_name WHERE condition;
```

🏆

Identify a functional dependencies of a relation and define a key of a relation with given functional dependencies

L.O.	2.3
Session	
Tag	Final

▼ Id a functional dependencies of a relation

Functional dependency: if A and B are R.attributes, B is functionally dependent on A if each value of A is associated with exactly one value of B. A and B may each consist of one or more attributes.

> 🧸 however, for a value B, there can be many different value A's

Determinant: $A \rightarrow B$, so A is the determinant of B

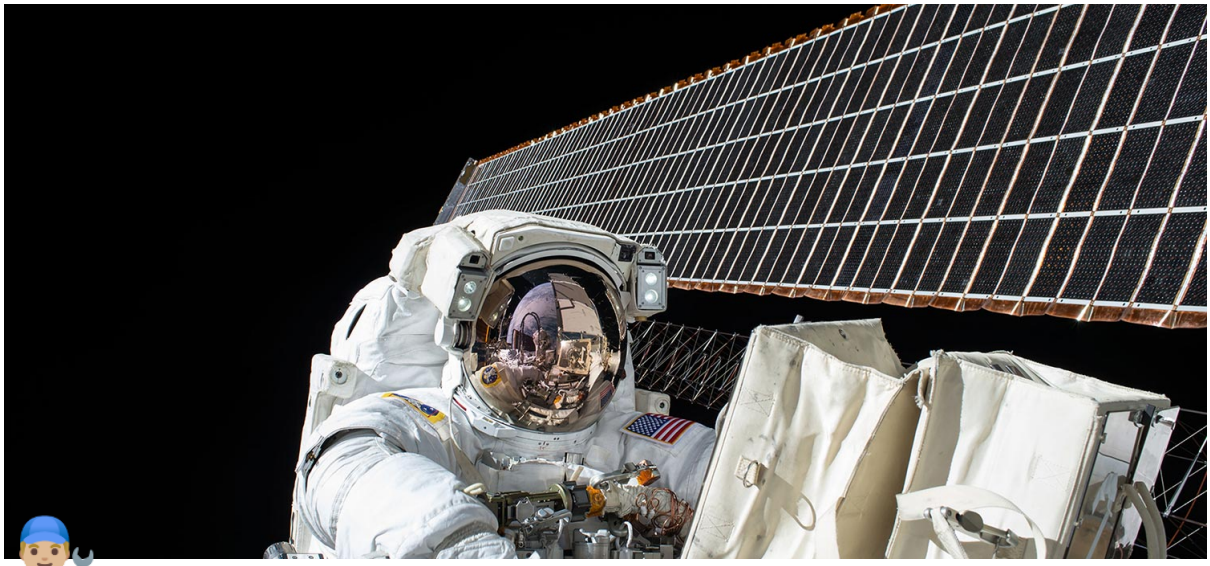
Characteristics of functional dependencies:

- 1:1 relationship in the direction from left to right.
- hold for all ***time***
- full functional dependencies between left hand side and right hand side.

▼ Define a key of a relation with given functional dependencies

Step	Description
1	Pick attributes that only appear in left into U_L
2	same as step 1, but in right into U_R
3	same as step 1, but in both left and right into U_{L+R}
4	closure of the subset of U_L. 💥 if the closure contains all the attributes of the relation ➡️ the closure is the candidate key.
5	closure of the subset of U_{L+R} 💥 if the closure contains all the attributes of the relation ➡️ the closure is the candidate key
Note	we dont need to traverse the closure in which the candidate key appears in

https://www.youtube.com/watch?v=L0LEtrIDYrE



Theory of Relational Algebra

UNARY OPERATIONS

Selection (or Restriction)

> The Selection operation works on a single relation R and defines a relation that contains only those tuples of R **that satisfy the specified condition (predicate).**

💎 **NOTATION:** $\sigma_{\text{predicate}}(R)$

Projection

> The Projection operation works on a single relation R and defines a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.

💎 **NOTATION:** $\prod_{\text{predicate}}(R)$

SET OPERATIONS

Union

> The union of two relations R and S defines a relation that contains all the tuples of R, or S, or both R and S,

> duplicate tuples being eliminated. R and S must be union-compatible.

💎 **NOTATION:** $R \cup S$

- If R and S have I and J tuples, respectively, their union is obtained by *concatenating them into one relation* **with a maximum of** $(I + J)$ **tuples**.

❓ **NOTE:** Union is possible **only if the schemas of the two relations match**, that is, if *they have the same number of attributes* with each pair of corresponding attributes having the same domain. In other words, the relations must be **union-compatible.**

▼ Set-difference

> The Set difference operation defines a relation consisting of the tuples that *are in relation R, but not in S*. **R and S must be union-compatible**.

💎 **NOTATION:** $R - S$

▼ Intersection

> The Intersection operation defines a relation consisting of the set of all tuples that are in *both R and S*. **R and S must be union-compatible**.

💎 **NOTATION:** $R \cap S$

Note that we can express the Intersection operation in terms of the Set difference operation:

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

▼ Cartesian product

> The Cartesian product operation defines a relation that is the concatenation of every tuple of relation R with every tuple of relation S.

💎 **NOTATION:** $R \times S$

The Cartesian product operation multiplies two relations to define another relation consisting of all possible pairs of tuples from the two relations.

❓ If one relation has I tuples and N attributes and the other has J tuples and M attributes, the Cartesian product relation will contain $(I * J)$ tuples with $(N + M)$ attributes. It is possible that the two relations may have attributes with the same name. In this case, the attribute names are prefixed with the relation name to maintain the uniqueness of attribute names within a relation.

▼ Decomposing complex operations

- We use the assignment operation, denoted by ←, to name the results of a relational algebra operation.

✅ $\text{TempViewing}(\text{clientNo}, \text{propertyNo}, \text{comment}) \leftarrow \text{PclientNo}, \text{propertyNo}, \text{comment}(\text{Viewing})$
$\text{TempClient}(\text{clientNo}, \text{fName}, \text{lName}) \leftarrow \text{PclientNo}, \text{fName}, \text{lName}(\text{Client})$
$\text{Comment}(\text{clientNo}, \text{fName}, \text{lName}, \text{vclientNo}, \text{propertyNo}, \text{comment}) \leftarrow \text{TempClient} \times \text{TempViewing}$
$\text{Result} \leftarrow \sigma_{\text{sclientNo}=\text{vclientNo}}(\text{Comment})$

- Alternatively, we can use the Rename operation ρ (rho), which gives a name to the result of a relational algebra operation. Rename allows an optional name for each of the attributes of the new relation to be specified.

> The Rename operation provides a new name S for the expression E, and optionally names the attributes as $a_1, a_2, \ldots, a_n$.

💎 **NOTATION:** $\rho_s(E)$ or $\rho_{s(a_1,a_2,\ldots,a_n)}(E)$

▼ JOIN OPERATIONS

Why Join instead of Cartesian product?

> Typically, we want only combinations of the Cartesian product that satisfy certain conditions and so we would normally use a Join operation instead of the Cartesian product operation.

❓ **Join** is a derivative of Cartesian product, equivalent to performing a Selection operation, using the join predicate as the selection formula, over the Cartesian product of the two operand relations

▼ Theta join (θ-join)

> **The Theta join operation** defines a relation that *contains tuples satisfying the predicate* F **from the Cartesian product of R and S**. The predicate F is of the form $R.a_i \ \theta \ S.b_i$, where θ may be one of the comparison operators $(<, \leq, >, \geq, =, \neq)$.

💎 **NOTATION:** $R \bowtie_F S$

We can rewrite the Theta join in terms of the basic Selection and Cartesian product
operations:

$$R \bowtie_F S = \sigma_F(R \times S)$$

❓ As with a Cartesian product, *the degree of a Theta join is the sum of the degrees of the operand relations R and S*. In the case where **the predicate F contains only equality**, the term **Equijoin** is used instead.

▼ Natural join

> **The Natural join** is an **Equijoin** of the two relations R and S **over all common attributes x**. One occurrence of each common attribute is eliminated from the result.

💎 **NOTATION:** $R \bowtie S$

▼ Outer join

> **The (left) Outer join** is a join in which **tuples from R that do not have matching values in the common attributes of S** are also included in the result relation. *Missing values in the second relation are set to null.*

💎 **NOTATION:** $R ⟕ S$

- The Outer join is becoming more widely available in relational systems and is a specified operator in the SQL standard.

❓ **The advantage of an Outer join is that information is preserved**; that is, the Outer join preserves tuples that would have been lost by other types of join.

- Left (natural) **Outer join** *keeps every tuple in the left-hand relation in the result*.
- Similarly, there is a **Right Outer join** that *keeps every tuple in the right-hand relation in the result*.
- There is also a **Full Outer join** that *keeps all tuples in both relations, padding tuples with nulls when no matching tuples are found*.

▼ Semijoin

> **The Semijoin operation** defines a relation that *contains the tuples of R that participate in the join of R with S satisfying the predicate F*.

💎 **NOTATION:** $R \triangleright_F S$

▼ DIVISION OPERATIONS

@Manh Khang

The Division operation is useful for a particular type of query that occurs quite frequently in database applications.

> **The Division operation** defines a relation over the attributes C that *consists of the set of tuples from R that match the combination of* ***every tuple in S***.

💎 **NOTATION:** $R \div S$ **= tuples of R associated with all tuples of S.**

We can express the Division operation in terms of the basic operations:

$$T_1 \neg \prod_{C}(R)$$
$$T_2 \neg \prod_{C}((T_1 \times S) - R)$$
$$T \neg T_1 - T_2$$

Identify all clients who have viewed all properties with three rooms.

We can use the Selection operation to find all properties with three rooms followed by the Projection operation to produce a relation containing only these property numbers. We can then use the following Division operation to obtain the new relation shown in Figure 5.12.

$$(\Pi_{\text{clientNo, propertyNo}}(\text{Viewing})) \div (\Pi_{\text{propertyNo}}(\sigma_{\text{rooms} = 3}(\text{PropertyForRent})))$$

$\Pi_{\text{clientNo,propertyNo}}(\text{Viewing})$

clientNo	propertyNo
CR56	PA14
CR76	PG4
CR56	PG4
CR62	PA14
CR56	PG36

$\Pi_{\text{propertyNo}}(\sigma_{\text{rooms}=3}(\text{PropertyForRent}))$

propertyNo
PG4
PG36

RESULT

clientNo
CR56

Figure 5.12 Result of the Division operation on the Viewing and PropertyForRent relations.

▼ AGGREGATE OPERATIONS

> Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list. AL contains one or more (<aggregate_function>, <attribute>) pairs.

💎 **NOTATION:** $_{\text{AL}}(R)$

- The main aggregate functions are:
  - **COUNT** – returns the number of values in the associated attribute.
  - **SUM** – returns the sum of the values in the associated attribute.
  - **AVG** – returns the average of the values in the associated attribute.
  - **MIN** – returns the smallest value in the associated attribute.
  - **MAX** – returns the largest value in the associated attribute.

⚠️ These operations **can work with NULL values** can they just consider NULL to be 0.

▼ GROUPING OPERATION

> **Groups the tuples of relation R** by the grouping attributes, GA, and then applies the aggregate function list AL to define a new relation. **AL contains one or more (<aggregate_function>,**

> **<attribute>) pairs**. The resulting relation *contains the grouping attributes, GA, along with the results of each of the aggregate functions*.

💎 **NOTATION:** $_{\text{GA}}\,_{\text{AL}}(R)$;

The general form of the grouping operation is as follows

$$a_1, a_2, \ldots, a_n \quad \langle A_p a_p \rangle, \langle A_q a_q \rangle, \ldots, \langle A_z a_z \rangle \; (R)$$

where:

- R is any relation, $a_1, a_2, ..., a_n$ are attributes of R on which to group
- $a_p, a_q, ..., a_z$ are other attributes of R
- $A_p, A_q, ..., A_2$ are aggregate functions.

✅ The tuples of R are partitioned into groups such that:
- all tuples in a group have the same value for $a_1, a_2, ..., a_n$;
- tuples in different groups have different values for $a_1, a_2, ..., a_n$

Theory of Relational Calculus

> **The relational calculus is not related to differential and integral calculus in mathematics**, but takes its name from a branch of symbolic logic called **predicate calculus**. When applied to databases, it is found in **two forms**: **tuple relational calculus**, as originally proposed by Codd (1972a), and **domain relational calculus**, as proposed by Lacroix and Pirotte (1977).

- In first-order logic or predicate calculus, **a predicate is a truth-valued function with arguments**. When we substitute values for the arguments, the function yields an expression, called a **proposition**, which can be either true or false.
- If a predicate contains a variable, there must be an **associated range** for x.
- A language that can be used to produce any relation that can be derived using the relational calculus is said to be **relationally complete**.

▼ Tuple Relational Calculus

The calculus is based on the use of **tuple variables**. **A tuple variable** is a variable that “ranges over” a named relation: that is, a variable whose only permitted values are tuples of the relation

For example, to specify the range of a tuple variable S as the Staff relation, we write:

```
Staff(S)
```

To express the query *“Find the set of all tuples S such that F(S) is true”*, we can write:

> 🥈 {S | F(S)}

F is called a **formula** (**well-formed formula**, or **wff** in mathematical logic).

▼ THE EXISTENTIAL AND UNIVERSAL QUANTIFIERS

There are two quantifiers we can use with formulae to tell how many instances the predicate applies to:

> 🥈 **The existential quantifier** $\exists$ (“*there exists*”) is used in formulae that must be true for at least one instance.

> 🥈 **The universal quantifier** $\forall$ (“*for all*”) is used in statements about every instance.

- Tuple variables that are qualified by $\forall$ or $\exists$ are called **bound variables**; the other tuple variables are called **free variables**. The only free variables in a relational calculus expression should be those **on the left side of the bar** (|).

▼ EXPRESSION AND FORMULAE

- As with the English alphabet, in which some sequences of characters do not form a correctly structured sentence, in calculus not every sequence of formulae is acceptable. *The formulae should be those sequences that are unambiguous and make sense*.

  $$S_1.a_1, S_2.a_2, ..., S_n.a_n | F(S_1, S_2, ..., S_m)\ m \geq n$$

  where $S_1, S_2, ..., S_n ..., S_m$ are tuple variables; each a_i is an attribute of the relation over which S_i ranges; and F is a formula. **A (well-formed) formula is made out of one or more atoms**, where an atom has one of the following forms:

  - $R(S_i)$, where S_i is a tuple variable and R is a relation.
  - $S_1.a_1\ \theta\ S_j.a_2$, where S_i and S_j are tuple variables, a_1, is an attribute of the relation over which S_i ranges, a_2 is an attribute of the relation over which S_j ranges, and θ is one of the comparison operators $(<, \leq, > , \geq, = . \neq)$; the attributes a_1 and a_2 must have domains whose members can be compared by θ
  - $S_i.a_1\ \theta\ c$, where S_i is a tuple variable, a_1 is an attribute of the relation over which S_i ranges, c is a constant from the domain of attribute a_1, and θ is one of the comparison operators.

- We recursively build up formulae from atoms using the following rules:
  - **An atom is a formula**
  - If F_1 and F_2 are formulae, **so are their conjunction** $F_1\ F_2$, **their disjunction** $F_1\ F_2$, and **the negation** $\sim F_1$.

- If F is a formula with free variable X, then $(\exists X)(F)$ and $(\forall X)(F)$ are also formulae.

▼ SAFETY OF EXPRESSIONS

It is possible for a calculus expression to generate an infinite set, for example:

```
🥈 {S | ∼ Staff(S)}
```

Such an expression is said to be **unsafe**.

- To avoid this, we have to add a restriction that all values that appear in the result must be values in the domain of the expression E, denoted $dom(E)$.

▼ Domain Relational Calculus

- An expression in the domain relational calculus has the following general form:

$$\{d_1, d_2, ..., d_n | F(d_1, d_2, \ldots, d_m)\} \ \ m \geq n$$

where $d_1, d_2, ..., d_n, ..., d_m$ represent domain variables and $F(d_1, d_2, ..., d_m)$ represents a formula composed of atoms, where each atom has one of the following forms:

  - $R(d_1, d_2, ..., d_n)$, where R is a relation of **degree** n and each d_i is a **domain variable**.
  - $d_i \, \theta \, d_j$, where d_i and d_j are domain variables and θ is one of the comparison operators $(<, \leq, >, \geq, =, \neq)$; the domains d_i and d_j must have members that can be compared by θ.
  - $d_i \, \theta \, c$, where d_i is a domain variable, c is a constant from the domain of d_i, and θ is one of the comparison operators.
- We recursively build up formulae from atoms using the following rules:
  - **An atom is a formula**
  - If F_1 and F_2 are formulae, **so are their conjunction** $F_1 \ F_2$, **their disjunction** $F_1 \ F_2$, and **the negation** $\sim F_1$.
  - If F is a formula with free variable X, then $(\exists X)(F)$ and $(\forall X)(F)$ are also formulae.

Use basic security techniques such as authorization and user management on an existing database management system

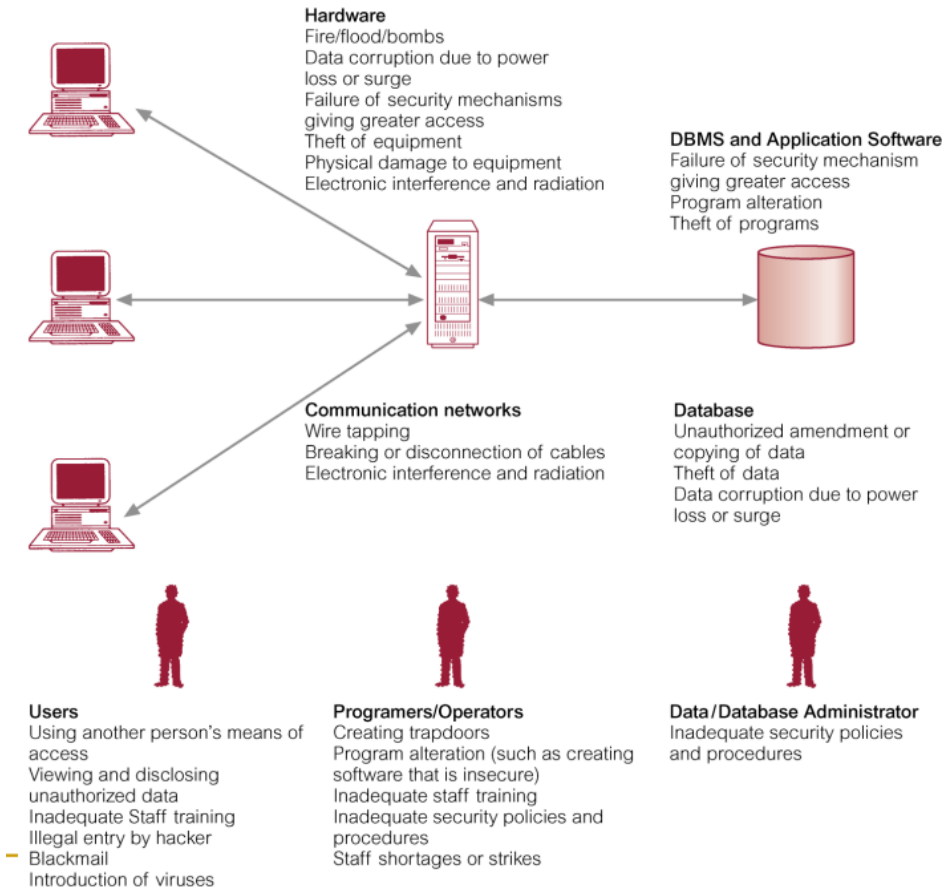
L.O.	3.4
Session	
Tag	Final

Types of Security

- **Various legal and ethical issues** regarding the right to access certain information.
- **Policy issues** at the governmental, institutional, or corporate level regarding what kinds of information should not be made publicly available.
- **System-related issues** such as the system levels at which various security functions should be enforced.
- **The need in some organizations to identify multiple security levels** and to categorize the data and users based on these classifications.

THREATS TO DATABASES:

- **Loss of integrity:** Database integrity refers to the requirement that information be protected from improper modification. Modification of data includes creating, inserting, and updating data; changing the status of data; and deleting data. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts. If the loss of system or data integrity is not corrected, continued use of the contaminated system or corrupted data could result in inaccuracy, fraud, or erroneous decisions.
- **Loss of availability:** Database availability refers to making objects available to a human user or a program who/which has a legitimate right to those data objects. Loss of availability occurs when the user or program cannot access these objects.
- **Loss of confidentiality:** Database confidentiality refers to the protection of data from unauthorized disclosure. The impact of unauthorized disclosure of confidential information can range from violation of the Data Privacy Act to the jeopardization of national security. Unauthorized, unanticipated, or unintentional disclosure could result in loss of public confidence, embarrassment, or legal action against the organization.



A DBMS typically includes a database security and authorization subsystem that is responsible for ensuring the security of portions of a database against unauthorized access. It is now customary to refer to two types of database security mechanisms:

- **Discretionary security mechanisms:** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
- **Mandatory security mechanisms:** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization.

▼ DATABASE SECURITY: NOT AN ISOLATED CONCERN

■ **Discretionary security mechanisms:** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
■ **Mandatory security mechanisms:** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization.

▼ Control measures

Four main control measures are used to provide security of data in databases:

■ **Access control:** The security mechanism of a DBMS must include provisions for restricting access to the database as a whole. The security problem associated with databases is that of controlling the access to a **statistical database**, which is used to provide statistical information or summaries of values based on various criteria.
■ **Inference control:** *The countermeasures to statistical database security problem*
■ **Flow control:** Prevents information from flowing in such a way that it reaches unauthorized users/
■ **Data encryption:** Used to protect sensitive data (such as credit card numbers) that is being transmitted via some type communication network. The data is **encoded** using some **coding algorithm**. An unauthorized user who accesses encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher the data

▼ Database Security and the DBA

The **database administrator (DBA)** is the central authority for managing a database system.

The DBA's responsibilities include:

- granting privileges to users who need to use the system.
- classifying users and data in accordance with the policy of the organization.

⇒ The DBA is responsible for the overall security of the database system

The DBA has a DBA account in the DBMS:

- Sometimes these are called a system or **superuser account**.
- These accounts provide powerful capabilities such as:
  1. **Account creation**
  2. **Privilege granting**
  3. **Privilege revocation**
  4. **Security level assignment**
- Action *1 is access control*, whereas *2 and 3 are discretionary* and *4 is used to control mandatory authorization*.

▼ Access Protection, User Accounts, and Database Audits

- Whenever a person or group of persons need to access a database system, the individual or group must first apply for a user account.

  → The DBA will then create a new **account id** and **password** for the user if he/she deems there is a legitimate need to access the database.
- The user must log in to the DBMS by entering account id and password whenever database access is needed
- The database system must also keep track of all operations on the database that are applied by a certain user throughout each login session.

→ To keep a record of all updates applied to the database and of the particular user who applied each update, we can modify system log, which includes an entry for each operation applied to the database that may be required for recovery from a transaction failure or system crash.

- If any tampering with the database is suspected, a database audit is performed.
  → A database audit consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period.
- A database log that is used mainly for security purposes is sometimes called an audit trail.

▼ Discretionary Access Control (DAC)

❓ User can **protect what they own**.

Owner may **grant access to other**.

Owner can **define the type of access** (read/write/execute/…) given to others.

The typical method of enforcing discretionary access control in a database system is based on the **granting and revoking privileges**.

▼ TYPES OF DISCRETIONARY PRIVILEGES

- **The account level:** At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- **The relation level (or table level):** At this level, the DBA can control the privilege to access each individual relation or view in the database.
- The privileges at the **account level** apply to the capabilities provided to the account itself and can include:
  - the **CREATE SCHEMA** or **CREATE TABLE** privilege, to create a schema or base relation;
  - the **CREATE VIEW** privilege;
  - the **ALTER** privilege, to apply schema changes such adding or removing attributes from relations;
  - the **DROP** privilege, to delete relations or views;
  - the **MODIFY** privilege, to insert, delete, or update tuples;
  - and the **SELECT** privilege, to retrieve information from the database by using a **SELECT** query.
- The second level of privileges applies to the **relation level**
  → This includes **base relations** and virtual (**view**) relations.
- In SQL the following types of privileges can be granted on each individual relation R:
  - **SELECT** (retrieval or read) privilege on R:

        - This gives the account retrieval privilege.
        - The **SELECT** statement is used to retrieve tuples from R.
    - **REFERENCES** privilege on R:
        - This gives the account the capability to **reference** relation R when specifying integrity constraints.
        - The privilege can also be **restricted** to specific attributes of R.
    - **MODIFY** privileges on R:
        - This gives the account the capability to modify tuples of R.
        - In SQL this privilege is further divided into **UPDATE**, **DELETE**, and **INSERT** privileges to apply the corresponding SQL command to R.
        - In addition, both the **INSERT** and **UPDATE** privileges can specify that only certain attributes can be updated by the account.
- Notice that to create a **view**, the account must have **SELECT** privilege on all relations involved in the view definition.
- The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the access matrix model where:
    - The **rows** of a matrix M represents **subjects** (users, accounts, programs).
    - The **columns** represent **objects** (relations, records, columns, views, operations).
    - Each position **M(i,j)** in the matrix represents the types of privileges (read, write, update) that **subject i** holds on **object j.**

	O_1	…	O_i	…	O_m
S_1	$A[s_1, o_1]$		$A[s_1, o_i]$		$A[s_1, o_m]$
…					
S_i	$A[s_i, o_1]$		$A[s_i, o_i]$		$A[s_i, o_m]$
…					
S_n	$A[s_n, o_1]$		$A[s_n, o_i]$		$A[s_n, o_m]$

- To control the granting and revoking of relation privileges, for each relation R in a database:
    - The owner of a relation is given all privileges on that relation.
    - The owner account holder can pass privileges on any of the owned relation to other users by granting privileges to their accounts.
    - The owner account holder can also take back the privileges by revoking privileges from their accounts.

▼ SPECIFYING PRIVILEGES USING VIEWS

- The mechanism of **views** is an important discretionary authorization mechanism in its own right.
    - If the owner A of a relation R wants another account B to be able to **retrieve only some fields** of R, then A can **create a view** V of R that includes **only those attributes** and then grant SELECT on V to B.

- The same applies to limiting B to retrieving only certain tuples of R; a view V' can be created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access.

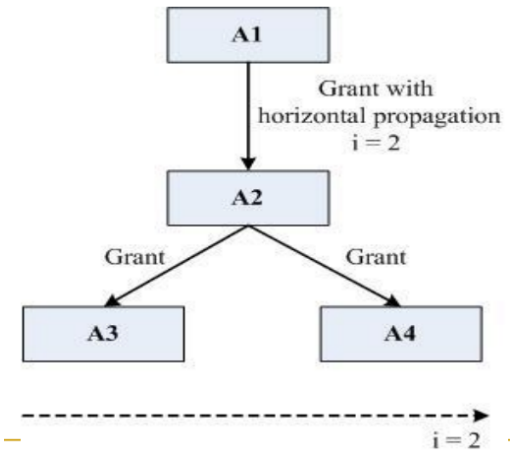
▼ REVOKING PRIVILEGES

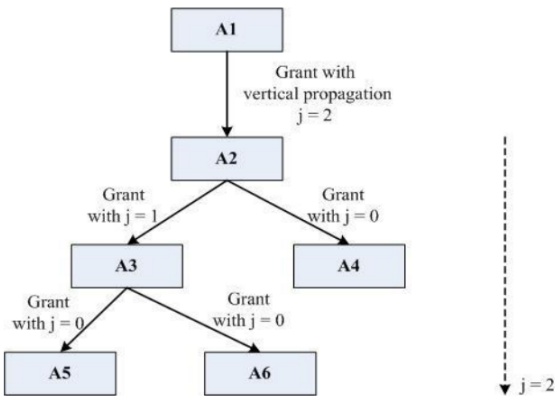
- In some cases it is desirable to grant a privilege to a user temporarily.
  - The owner of a relation may want to grant the **SELECT** privilege to a user for a specific task and then revoke that privilege once the task is completed.
  - Hence, a mechanism for **revoking** privileges is needed. In SQL, a **REVOKE** command is included for the purpose of **canceling privileges**.

▼ PROPAGATION OF PRIVILEGES USING THE GRANT OPTION

- Whenever the owner A of a relation R grants a privilege on R to another account B, privilege can be given to B with or without the GRANT OPTION.
- If the **GRANT OPTION** is given, this means that B can also grant that privilege on R to other accounts.
  - Suppose that B is given the **GRANT OPTION** by A and that B then grants the privilege on R to a third account C, also with GRANT OPTION. In this way, privileges on R can **propagate** to other accounts without the knowledge of the owner of R.
  - If the owner account **A now revokes** the privilege granted to B, **all the privileges that B propagated** based on that privilege should automatically **be revoked** by the system.

▼ LIMITING THE HORIZONTAL PROPAGATION



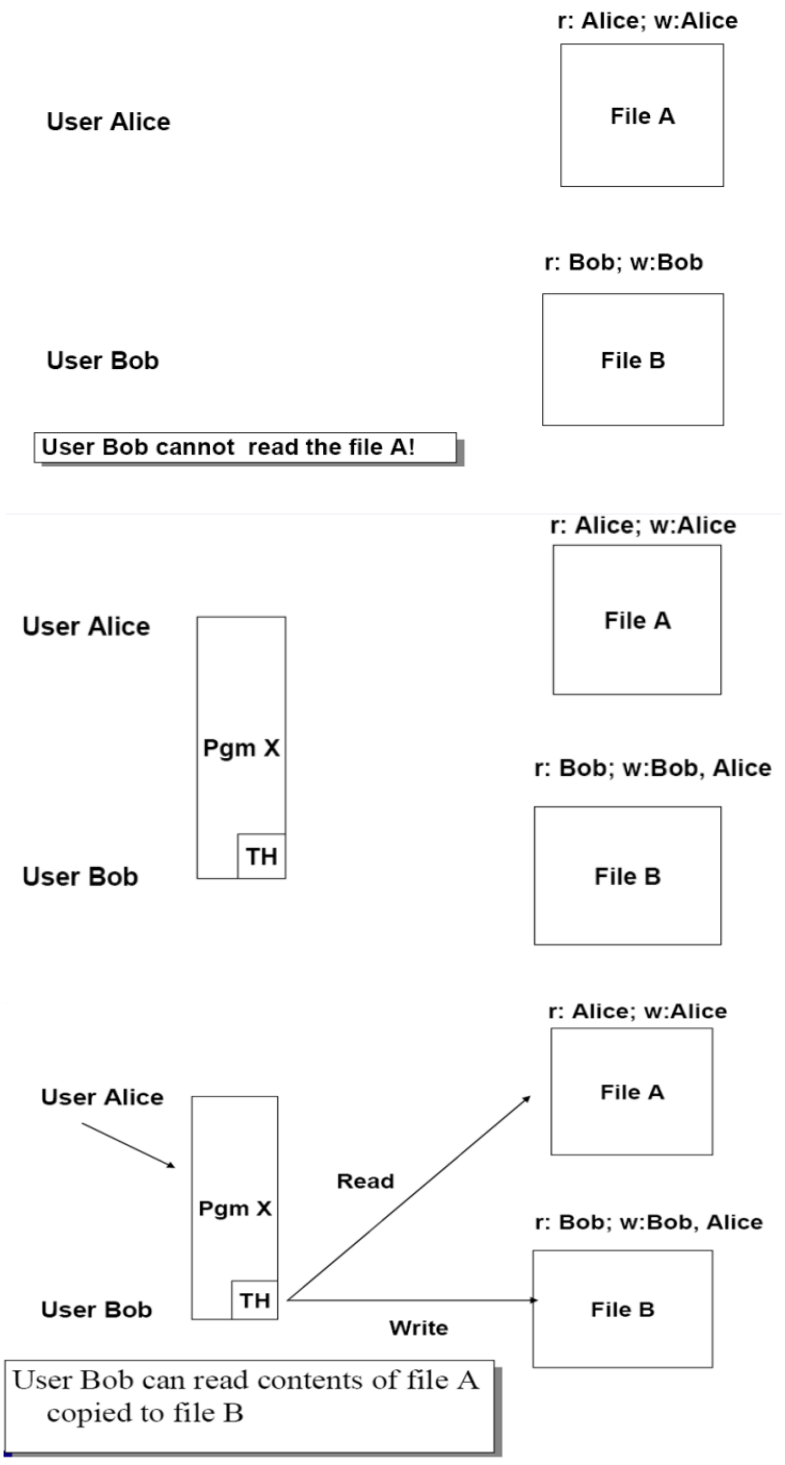


▼ INHERENT WEAKNESS OF DAC

- Unrestricted DAC allows information from an object which can be read by a subject to be written to any other object

- Bob is denied access to file Y, so he asks Alice to copy Y to X that he can access.

- Suppose our users are trusted not to do this deliberately. It is still possible for **Trojan Horses** to copy information from one object to another.



▼ Mandatory Access Control (MAC)

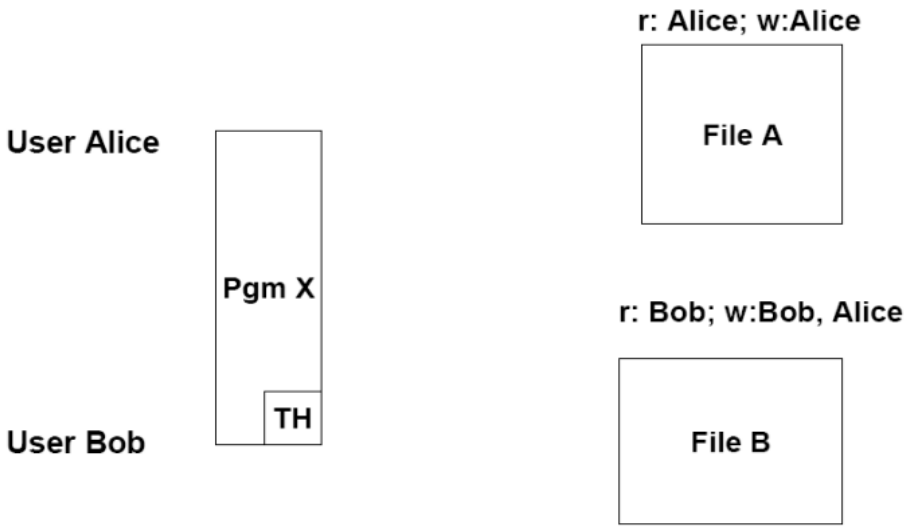
💎 **Granting access** to the data on the basis of users' clearance level and the sensitivity level of the data.

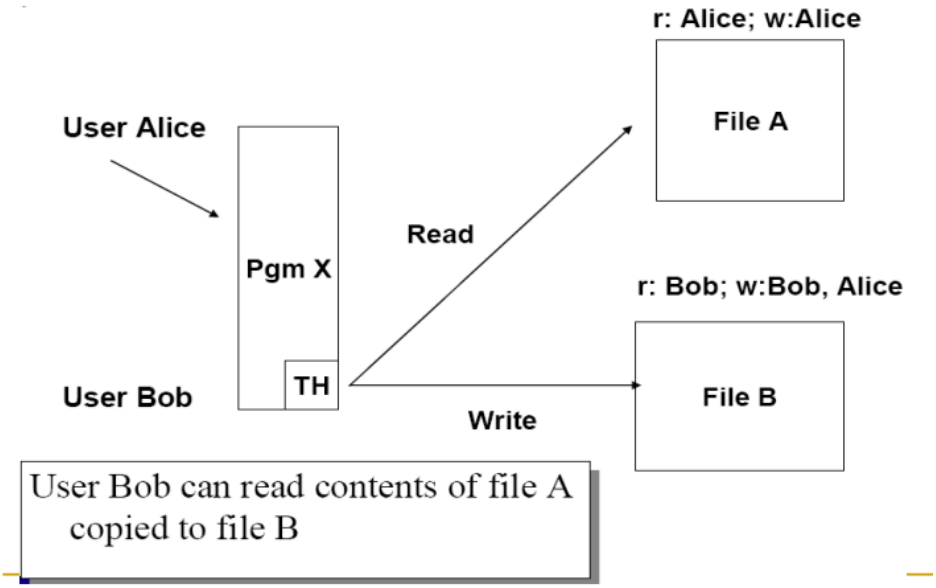
- **Bell-LaPadula's two principles**: *no read-up & no write-down secrecy*. **This model only focues on confidentiality of data.**

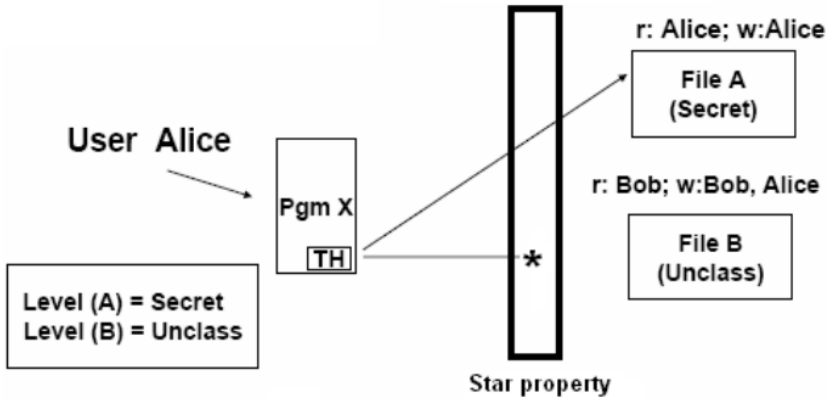
▼ BELL-LAPADULA MODEL

✅ Typical **security classes** are *top secret (TS), secret (S), confidential (C)*, and *unclassified (U)*, where TS is the highest level and U is the lowest one: $TS \geq S \geq C \geq U$

- Two restrictions are enforced on data access based on the subject/object classifications:
  - **A subject S is not allowed read access to an object O** unless **class(S) ≥ class(O)**. This is known as the simple security property.
  - **A subject S is not allowed to write an object O unless class(S) ≤ class(O)**. This known as the **star property** (or *** property**).







▼ MULTILEVEL RELATION

- **Multilevel relation:** MAC + relational database model.
- **Data objects:** attributes and tuples.
- Each attribute A is associated with a **classification attribute C**.
- A **tuple classification** attribute **TC** is to provide a classification for each tuple as a whole, the highest of all attribute classification values.

$$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

- The **apparent key** of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation.

  → *A multilevel relation will appear to contain different data to subjects (users) with different security levels.*

SELECT * FROM EMPLOYEE

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

- A user with security **level S**:

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

- A user with security **level C**:

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	NULL C	C
Brown C	NULL C	Good C	C

- A user with security **level U**:

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	NULL U	NULL U	U

▼ PROPERTIES OF MULTILEVEL RELATION

- **Read and write operations:** satisfy the **No Read-Up** and **No Write-Down** principles
- **Entity integrity:** all attributes that are members of the apparent key **must not be null** and must have the **same security classification** within each individual tuple.
- In addition, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key.

  → This **constraint** ensures that a user can see the key if the user is permitted to see any part of the tuple at all.
- **Polyinstantiation:** where several tuples can have the same apparent key value but have different attribute values for users at different classification levels.

- A user with security **level C**:

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	NULL C	C
Brown C	NULL C	Good C	C

```
UPDATE EMPLOYEE
SET Job_performance = ‘Excellent’
WHERE Name = ‘Smith’;
```

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Smith U	40000 C	Excellent C	C
Brown C	80000 S	Good C	S

Polyinstantiation of the Smith tuple.

▼ COMPARING DAC AND MAC

- By contrast, mandatory policies ensure a high degree of protection in a way, they prevent any illegal flow of information.
- Mandatory policies have the drawback of being too rigid and they are only applicable in limited environments.
- In many practical situations, discretionary policies are preferred because they offer a better trade-off between security and applicability.

▼ Role-based access control (RBAC)

- **Role-based access control (RBAC)** emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprisewide systems.
- Its basic notion is that permissions are associated with roles, and users are assigned to appropriate roles.
- Roles can be created using the **CREATE ROLE** and **DESTROY ROLE** commands.
  - The **GRANT** and **REVOKE** commands discussed under DAC can then be used to assign and revoke privileges from roles.

❓ **RBAC** appears to be a viable alternative to traditional discretionary and mandatory access controls; **it ensures that only authorized users are given access to certain data or resources**.

- Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.
- Role hierarchy in **RBAC** is a natural way of organizing roles to reflect the organization's lines of authority and responsibility.
- Another important consideration in **RBAC** systems is the possible temporal constraints that may exist on roles, such as time and duration of role activations, and timed triggering of a role by an activation of another role.
- Using an **RBAC** model is highly desirable goal for addressing the key security requirements of Web-based applications.
- In contrast, discretionary access control (**DAC**) and mandatory access control (**MAC**) models **lack capabilities** needed to support the security requirements emerging enterprises and Web-based applications.

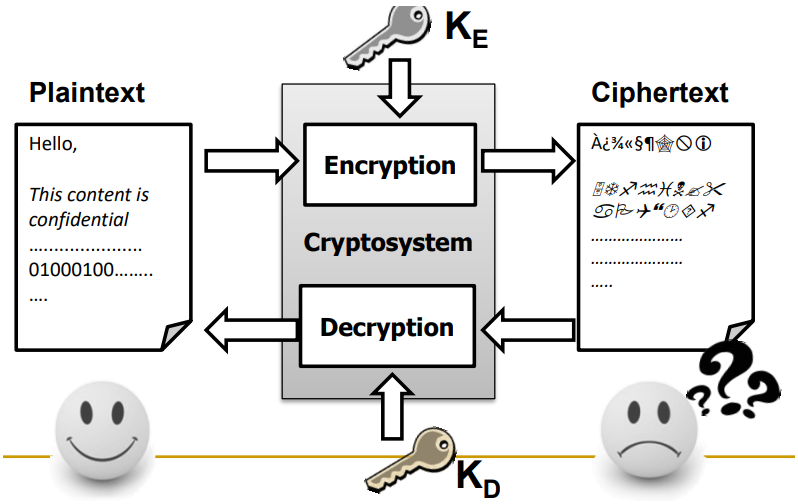
▼ Encryption & PKI (Public Key Infrastructure)

▼ ENCRYPTION

- The **encoding of the data by a special algorithm** that renders the data unreadable by any program without the decryption key.
- **Symmetric cryptography:** sender and receiver use the same key.
- **Asymmetric cryptography:** encryption & decryption keys.
- **Plaintext** is the original content which is readable as textual material. Plaintext needs protecting.
- **Ciphertext** is the result of encryption performed on plaintext using an algorithm. Ciphertext is not readable.

> ❓ **NOTE:**
>
> Cryptosystems = encryption + decryption algorithms.

- Encryption, decryption process needs **keys**.
- **Symmetric (shared-/secret-key) cryptosystem:** the same key for (en/de)cryption algorithms ($K_E = K_D$).
- **Asymmetric (public-key) cryptosystem:** public & private keys ($K_E \neq K_D$).



- (Most popular) **Symmetric techniques**: DES (**Data Encryption Standard**), AES (**Advanced Encryption Standard**).
  - The same key is used for both encryption and decryption.
  - Faster than encryption and decryption in publickey (PK) cryptosystems.
  - Less security comparing to encryption and decryption in PK cryptosystems.
- **Asymmetric techniques:** RSA (**cryptosystem algorithm**), DSA (**digital signature algorithm**)

▼ SYMMETRIC TECHNIQUES

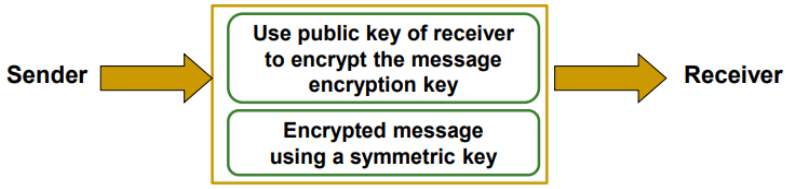
- **DES: Data Encryption Standard**
  - A message is divided into 64-bit blocks
  - Key: 56 bits
  - Brute-force or exhaustive key search attacks: Some hours.
- **Triple DES:** run the DES algorithm a multiple number of times using different keys
  - m : plaintext
  - ε_{k_1} : encryption by key k_1
  - c: ciphertext
  - D_{k_1} : decryption by key k_1
  - **Encryption**: $c \leftarrow \varepsilon_{k_1}(D_{k_2}(\varepsilon_{k_1}(m)))$
  - **Decryption**: $c \leftarrow D_{k_1}(\varepsilon_{k_2}(D_{k_1}(m)))$

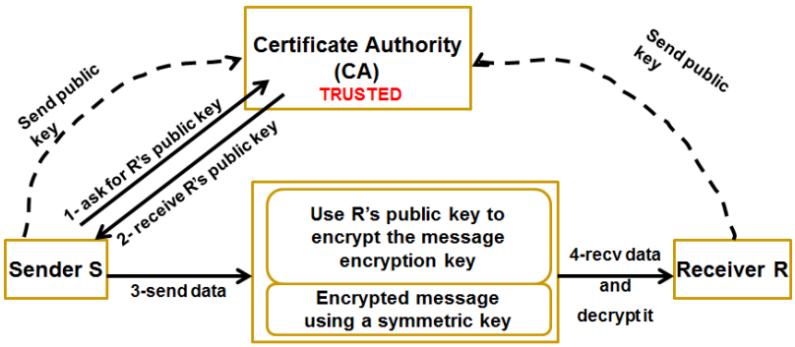
    $\rightarrow$ Be compatible with DES when $k_1 = k_2$
  - The triple DES can also use three different keys.
- **AES: Advanced Encryption Standard (Rijndael)**
  - **Jan 2, 1997:** NIST announced the initiation of a new symmetric-key block cipher algorithm, AES, as the new encryption standard to replace the DES.

    - **Oct 2, 2000:** Rijndael was selected. Rijndael is designed by two Belgium cryptographers: Daemen and Rijmen.
- The key size and the block size can be independently specified to 128, 192 or 256 bits.
- Rijndael is a block cipher with a variable block size and variable key size.

▼ ASYMMETRIC TECHNIQUES

- **RSA:** named after 3 inventors Rivest, Shamir, Adleman
    - **Two keys:** public key and private key
    - **Public key** is used for encrytion.
    - **Private key** is used for decrytion
- **Encryption key:** public key
- **Decryption key:** private key
- **Asymmetric techniques:** more secure but expensive in terms of computational costs





❓ **On which hard mathematical problem does RSA base its security? (src: DBS test - BKEL)**
ANSWER: Factorization of big numbers
Link:

> RSA numbers - Wikipedia
> In mathematics, the RSA numbers are a set of large semiprimes (numbers with exactly two prime factors) that were part of the RSA Factoring Challenge. The challenge was to find the prime factors of each number. It was created by RSA Laboratories in March 1991 to
> 𝕎 https://en.wikipedia.org/wiki/RSA_numbers

▼ DIGITAL SIGNATURES

- A **digital signature** is an example of using encryption techniques to provide authentication services in ecommerce applications.
- A digital signature is a means of associating a mark unique to an individual with a body of text.
  - The mark should be unforgettable, meaning that others should be able to check that the signature does come from the originator.
- A digital signature consists of a string of symbols:
  - Signature must be different for each use.
    - This can be achieved by making each digital signature a function of the message that it is signing, together with a timestamp.
  - Public key techniques are the means creating digitalb signatures

▼ DIGITAL CERTIFICATES

- One concern with the public key approach: must ensure that you are encrypting to the correct person’s public key.
- A solution: digital certificates.
- A form of credentials (like a physical passport).
- Included with a person’s public key to verify that a key is valid.

Components of a digital certificate

- A digital certificate:
  - A public key
  - Certificate info (identifying information such as name, ID)
  - One (or more) digital signatures
- Certificates are used when it is necessary to exchange public keys with someone (when you cannot manually exchange via a diskette or USB drive)

▼ **SOME QUIZ FROM BKEL:**

> **Q1:** The accounting branch of a large organization requires an application to process expense vouchers. Each voucher must be input by one of many accounting clerks, verified by the clerk’s applicable supervisor, then reconciled by an auditor before the reimbursement check is produced. Which access control technique should be built into the application to best serve these requirements?
>
> **Answer:** RBAC

> **Q2:** Company X is planning to implement rule based access control mechanism for controlling access to its information

> assets, what type of access control is this usually related to?
>
> **Answer:** Discretionary Access Control and RBAC

> **Q3:** Which of the following mechanisms can be used for user authentication?
>
> 1. Something the user do not knows
> 2. Something the user possesses
> 3. Something the user is
> 4. Where the user is
> 5. Access control methods
>
> **Answer:** 2, 3 and 4

Write data definition and manipulation (query, insertion, deletion, and update) statements as well as procedures/functions/triggers for data processing in SQL

L.O.	3.1
Session	
Tag	Final

Procedure	Function
void	return a value

Stored procedure is useful in these circumstances:

- many apps that need a certain database program (stored procedure) can invoke it so that duplication of effort is avoided
- Exec a program at the server reduce efficiency
- Allowing more complex types of derived data to be present to the db user ⇒ enhance modeling power provided by views
- Check for constraints byond specification by triggers and assertions

SQL Stored Procedures

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again. So if you have an SQL query that you write over and over again, save it as a stored procedure,

https://www.w3schools.com/sql/sql_stored_procedures.asp

Function vs. Stored Procedure in SQL Server

I realize this is a very old question, but I don't see one crucial aspect mentioned in any of the answers: inlining into query plan. Functions can be... Scalar: CREATE FUNCTION ... RETURNS scalar_type

https://stackoverflow.com/questions/1179758/function-vs-stored-procedure-in-sql-server

```
create trigger total_salary
after insert
```

Aggregate functions ignore NULL values?

> ⚠️ only `COUNT(*)` include NULL values.

All aggregate functions **except COUNT(*) and GROUPING ignore nulls**. You can use the NVL function in the argument to an aggregate function to substitute a value for a null. **COUNT never returns null, but returns either a number or zero**. For all the remaining aggregate functions, **if the data set contains no rows**, or contains **only rows with nulls as arguments to the aggregate function**, then the function returns null.

Nulls: Nulls and Aggregate Functions

This tutorial demonstrates how aggregate functions deal with null values. Techniques for generating results that ignore nulls and results that include nulls are highlighted. Ignoring Nulls According to the SQL Reference Manual section on Aggregate Functions: All aggregate functions except COUNT(*) and GROUPING

https://www.sqlsnippets.com/en/topic-12656.html