COMP23111 - Fundamentals of Databases

November 26, 2017

1 Introduction to Data Management

A **database management system (DBMS)** is a collection of interrelated data and a set of programs to access those data. It provides a **convenient** and **efficient** way to store and retrieve database-information using advanced techniques as transaction and recovery management. It allows decoupling a logical view of data from storage technology and formats, thus lowering the cost of developing new target applications.

The previously used **file-processing system** stored permanent records in various files and thus needed different application programs and sharing was not easy. Major disadvantages:

- **Data redundancy and inconsistency:** leads to higher storage and access cost.
- **Difficulty in accessing data:** data cannot be retrieved in a convenient and efficient manner.
- **Data isolation:** application is peculiar and specific, developed for a target data from scratch.
- **Integrity problems.**
- **Atomicity problems:** difficult to ensure atomicity, either it happens entirely or not at all.
- **Concurrent-access anomalies.**
- **Security problems:** not every user of the database system should be able to access all data.

Data abstraction: the system hides certain details of how the data are stored and maintained. We have:

- **Physical level (lowest):** how the data are actually stored (complex low-level data structures). It defines formats, records, files, clustering, indexes, compression, replication, redundancy, etc...
- **Logical level (next-higher):** what data are stored and the relationships among those.
- **View level (highest):** description of only part of the database, simplify user-system interaction.

An **instance** of the database is the collection of information stored in it at a particular moment in time. A database **schema** is its overall design. The **physical schema** describes it at the physical level while the **logical schema** describes the design at the logical level. **Subschemas** describe different views of the database.

A **data model** is a collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints. It defines whether there is a distinct notion of a schema and what the schema is allowed to declare.

- **Relational Model:** it uses a collection of tables to represent both data and the relationships among those. It is an example of **record-based model** since the database is structured in fixed-format records of several types. It is the most widely used data model.
- **Entity-Relationship Model:** an **entity** (or attribute) is an object in real world distinguishable from others. It is a popular, high-level conceptual data modelling approach which focuses on data requirements.
- **Object-Based Data Model:** it extends the E-R model with notions of encapsulation, method and object identity.
- **Semistructured Data Model:** it permits the specification of data where individual data items of the same type may have different sets of attributes. Use of **Extensible Markup Language (XML).**

A **database system** provides a **data-definition language (DDL)** to specify the database schema and a **data-manipulation language** to express queries and updates. Different types of database language are needed as this approach enforces **separation of concerns**.
A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organised by the appropriate data model. It can be:

- **Procedural:** require a user to specify what data are needed and how to get them.
- **Declarative or Non-Procedural:** only require a user to specify what data are needed, it is usually easier to learn.

The types of **data-access** are **retrieval, insertion, deletion, modification**.
A **query** is a statement requesting the retrieval of information through a **query-language (QL)**, which returns values without changing the database instance.
A **data storage and definition language** specifies the storage structure and access methods used by the database system.

The data values must satisfy certain **consistency constraints**:

- **Domain constraints:** a domain of possible values must be associated with every attribute.
- **Referential integrity:** a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation.
- **Assertions:** any condition that the database must always satisfy.
- **Authorisation:**
  - **Read authorisation:** reading but no modification.
  - **Insert authorisation:** insertion but no modification.
  - **Update authorisation:** modification but no deletion.
  - **Delete authorisation:** deletion.

Application programs interact with the database via **host languages** since there are some computations not possible using SQL (input from users, output to display, ...). These programs:

- Provide an application program interface (set of procedures) that can be used to send DML and DDL statements to database and retrieve the results.
- Extend the host language syntax to embed DML calls within the host language program using a **DML precompiler**.

Database design process:

1. Characterise fully the data needs of the prospective database users.
2. **Conceptual design phase:** data model choice and conceptual schema. Includes the **specification of functional requirements**.
3. **Logical design phase:** conceptual schema mapped to implementation data model.

4. **Physical design phase:** physical features of the database are specified.

The **Unified Modelling Language (UML)** is commonly used to express graphically an entity-relationship model using a *rectangular box* for entity sets and a *diamond* for relationships.

Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.

Normalisation: design schemas in an appropriate **normal-form** using **functional dependencies** in order to avoid **anomalies**.

The **query processor** helps the database system to simplify and facilitate access to data. It is made of:

- **DDL interpreter:** it interprets DDL statements and records definition in **data dictionaries**.
- **DML compiler:** it translates DML statements in a query language into an evaluation plan consisting of low-level instructions. It also perform **query optimisation** picking the lowest cost evaluation plan.
- **Query evaluation engine:** it executes low-level instructions.

The **storage manager** provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. It is responsible for *storing, retrieving and updating data* in the database. It has:

- Components:
  - **Authorisation and integrity manager:** tests for the satisfaction of integrity constraints and check for the authority of users to access data.
  - **Transaction manager:** ensures the database is always consistent and non conflicting.
  - **File manager:** manages the space allocation on disk storage and data structures.
  - **Buffer manager:** responsible for fetching data and select what to cache.
- Data structures:
  - **Data files:** store the database itself.
  - **Data dictionary:** stores metadata (schemas).
  - **Indixes:** provide fast access to data items.

A **transaction** is a collection of operations that performs a single logical functions. They need to be:

- **Atomicity:** if two or more discrete pieces of information are involved, either all of the pieces are committed or none.
- **Consistency:** it either creates a new valid state of data or return to its previous state.
- **Isolation:** a process not yet committed must remain isolated from any other transaction.
- **Durability:** in the event of a failure, the data is available and in a correct state (*failure recovery).*

The **transaction manager** consists of the **concurrency-control manager** (controls interaction) and the **recovery** one (ensures atomicity and consistency).

In a **two-tier architecture** the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements.
In a **three-tier architecture** the client machine acts as a front end and does not contain any direct database calls, but instead uses an **application server**.

Data mining is the process of semi-automatically analysing large databases to find patterns.

There are different users:

- **Naïve users**: unsophisticated users who interact with the system by invoking one of the application programs.
- **Application programmers:** computer professionals who write application programs.
- **Sophisticated users:** interact with the system without writing programs.
- **Specialised users:** write specialised database applications that do not fit into the traditional data-processing framework.

Database administrator (DBA): schema definition, storage structure and access-method definition, schema and physical-organisation modification, granting for authorisation and data access and routine maintenance.

2 The Entity-Relationship approach

Database application: a particular database and the associated programs that implement the database queries and updates.
In **ER** diagrams operations on objects are specified to outline the **functional requirements**.

2.1 Database design process

1. **Requirements collection and analysis:**
   - **Data requirements** by interviewing prospective database users.
   - **Functional requirements:** user defined operations (or transactions) applied to databases.
2. **Conceptual design:** create **conceptual model** (also called a conceptual schema, it is a concise description of the data requirements of the users, including detailed descriptions of the entity types, relations and constraints.
3. **Logical Design/ Data model mapping:** database schema in the implementation data model.
4. **Physical Design:** the internal storage structures, file organisations, indexes, access paths and physical design parameters for the database files are specified. Application programs are designed and implemented.

2.2 Entity types, entity sets, attributes and keys

An **entity** is a thing in the world with an independent existence.
An entity has **attributes** which are particular properties that can describe it. An attribute can be:

▲ **Simple/Atomic:** are not divisible.

▲ **Composite:** can be divided into smaller subparts, which represents more basic attributes with independent meaning; they can form a hierarchy; the value of a composite attribute is the concatenation of the values of its component simple attributes. If the composite attribute is references only as a whole, there is no need to subdivide it into component attributes.

⧫ **Single-valued:** single value for a particular entity.

⧫ **Multi-valued:** may have lower and upper bounds to constrain the number of values allowed for each individual entry.

- **Stored.**

- **Derived:** computed from stored one.

• **NULL:** not applicable, missing or not known value.

• **Complex:** arbitrarily-nested composite/multivalued.

An **entity type** defines a collection or set of entities that have the same attributes.
An **entity set** is the collection of all entities of a particular entity type in the database.
The **extension** of the entity type is the schema for a set of entities that share the same structure.
The **key attribute** is a value used to identify each entity uniquely. In ER diagrams it is underlined inside the oval. Some entity type have more than one key attribute. A key attribute **associated with a value set**, specifies the set of values that may be assigned to that attribute typically using the basic data types.

A **relationship type** R among n entity types defines a **relationship set** (a set of relationship instances) among entities of the same type.

The **degree** of a relationship type is the number of participating entity types, like *binary* or *ternary*.
The **role name** signifies the role that a participating entity plays in each relationship instance.
Recursive relationship: the same entity type participates more than once in a relationship type in different role, the role names becomes essential.

The **cardinality ratio** for a binary relationship specifies the maximum number of relationship instances that an entity type can participate in: we have $1 : N$, $N : 1$ and $M : N$ where M, N indicate no maximum number.
The **participation constrain** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. We can have:

- **Minimum cardinality constraint:** minimum number of relationship instances that each entity can participate in.
- **Partial participation:** some/part of the set must be related but not necessarily all.
- **Total participation:** every entity in the total set of entities must be related.

The **structural constraint** is composed by the cardinality ratio and the participation constraint.
For a $1 : N$ relationship type, a relationship attribute can be migrated only to the entity type on the N-side of the relationship. For $M : N$ relationship types, some attributes may be determined by the combination of participating entities in a relationship instance, not by any single entity. Such attributes must be specified as relationship attributes.

Weak entity types do not have key attributes of their own while **strong entity types** do.
The **identifying/owner entity type** is related to a weak entity type through a **identifying relationship**.
A weak entity type (sometimes is a complex attribute) normally has a **partial key**: attributes that uniquely identifies weak entities related to the same owner entity. A weak key is not global, it depends on the context to which it applies.

We choose to use singular names for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type.
The nouns tend to give rise to entity type names while the verbs tend to indicate the names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types.
To make relationships explicit we change attributes that represent relationships into relationship types and we determine the cardinality ratio and participation constraints of each relationship types.

3 The Enhanced Entity-Relationship Approach

Subgroupings are called **subclasses/subtypes**, each of which have **superclasses/supertypes** similar to *class/subclass relationships*.
The *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. **Type inheritance:** an entity that is a member of a subclass inherits all the attributes of the entity as a member of the superclass and the relationships.
Specialisation: process of defining a set of subclasses of the superclass of the specialisation (top-down conceptual synthesis). **Specific attributes/Local attributes:** attributes that apply only to entities of a particular subclass. Specialisation allows us to:

- Define a set of subclasses of an entity type.
- Establish additional specific attributes with each subclass.
- Establish additional specific relationship types between each subclass and other entity types.

Generalisation: process of defining a generalised entity type from the given entity types (bottom-up conceptual synthesis).

Predicate-defined/Condition-defined subclasses: there are conditions on the value of some attributes of the superclass. The subclass has thus a *defining predicate*.
Attribute-defined specialisation: all subclasses in a specialisation have their membership condition on the same attribute of the superclass, called **defining attribute**.
User-defined subclass: there is no condition to determine membership in a subclass. Membership is specified individually for each entity by the user.

Disjointness constraint: the subclasses of the specialisation must be disjoint. An entity can be a member of at most one of the subclasses of the specialisation. We use ⓓ to indicate it. If the subclasses are not constraint to be disjoint, their sets of entities may be **overlapping**, we indicate it with ⓞ.

Completeness/Totalness constraint on a specialisation:

- **Total:** every entity in the superclass must be a member of at least one subclass in the specialisation.
- **Partial:** allows entities *not* belong to any of the subclasses.

The disjointness and completeness constraints are independent: four different combinations. In general, a superclass that was identified through generalisation is usualy total, as the superclass is derived from the subclass and hence contains only the entities that are in the subclasses.

A **specialisation hierarchy** has the constraint that every subclass participates as a subclass in *only one* class/subclass relationship, like a **tree structure/strict hierarchy**.11 In a **specialisation lattice**, a subclass can be a subclass in more than one class/subclass relationship.
A **leaf node** is a class that has no subclasses of its own.
A **shared subclass** is a class with more than one superclass.

We can have **multiple inheritance**, where shared subclass directly inherits attributes and relationship from multiple classes, and **single inheritance**.
Some models and languages are limited to single inheritance and do not allow multiple inheritance. Some models do not allow an entity to have multiple types: an entity can be a member of only one leaf class.
Top-down and bottom-up conceptual refinement process the only difference between hierarchies and lattices relates to the manner or order in which the schema, superclasses and subclasses were created during the design process.
Union type/category: a single superclass/subclass relationship with more than one superclass, where the superclasses represent different entity types. It can be **total**, when it holds the union of all entities in its superclasses, or **partial**, when it holds a subset of the union.

3.1 Guidelines for the designing process

Represent only those subclasses that are deemed necessary to avoid extreme cluttering of the conceptual schema.
If a subclass has few specific local attributes and no specific relationships, it can be merged into the superclass.
If all the subclasses of a specialisation/generalisation have few specific attributes and no specific relationships they can be merged into the superclass and replaced with one or more type attributes that specify the subclass or subclasses each entity belongs to.

Union types and categories should be avoided unless the situation definitely warrants this type of constrict.

The choice of disjoint/overlapping and total/partial constraints on specialisation/generalisation is driven by the needs in the mini-world being modelled. As a default: overlapping and partial.

3.2 Formal definitions

A **class** is a set of collection of entities.
A **subclass** S is a class whose entities must always be a subset of the entities in another class, called the **superclass** C of the **superclass/subclass relationship** $S \subseteq C$.

A **specialisation** $Z = \{S_1, S_2, \ldots, S_n\}$ is a set of subclasses that have the same superclass G, which is called the **generalised entity type**. Z is **total** if $\bigcup_{i=1}^{n} S_i = G$ while otherwise is said to be partial. Z is **disjoint** if $S_i \cap S_j = \emptyset$ for $i \neq j$, otherwise is **overlapping**.
A subclass S of C is said to be **predicate-defined** if a predicate p on the attributes of C is used to specify which entities in C are members of $S \leftarrow S = C[p]$, otherwise is **user-defined**.
A specialisation Z is said to be **attribute-defined** if a predicate $(A = c_i)$, where A is the domain and c_i is the constant value, is used to specify membership in each subclass S_i in Z. A **category** T is a class that is a subset of the union of n defining superclasses, $T \subseteq (D_1 \cup \cdots \cup D_n)$.
A **base class** is the root superclass.

The goal of **knowledge representation** techniques is to develop concepts for accurately modelling some **domain knowledge** by creating an **ontology** that describes the concepts of the domain and how these concepts are interrelated.
KR schemas include **reasoning mechanisms** that deduce additional facts from those stored in the database, it can answer queries that involve inferences over the stored data.

The process of **classification** involves systematically assigning similar objects/entities to classes/types. **Instantiation** is the inverse of classification and refers to the generation and specific examination of distinct objects of a class.
Exception objects: objects with properties that differ in some respects from the other of the class.
Meta-class: one class is an instance of another class.
Identification: abstraction process whereby classes and objects are made uniquely identifiable by means of some **identifier**. Identification happens for:

- To distinguish among database objects and classes.
- To identify database objects and to relate them to their real-world counterparts.

Aggregation is an abstraction concept for building composite objects from their component objects.
Association is used to associate objects from several *independent* classes.
Semantic Web: based on *ontologies*, specification of a *conceptualisation* which is a set of concepts used to represent the knowledge of interest to users.

4 The Relational Model

Data model: notation for describing the structure of the data in a database along with the constraints of that data. It also describes operations on data (queries and modifications).
Relational model: tables representing information, it has only one collection type. It is a regular, rectangular tables with typed, atomic columns. Table definitions capture relation schemas/types while table states capture relation instances. The **columns** are attributes with associated domain and data types. The **rows** are tuples of one component for each attribute of the relation, each component of each tuple have to be *atomic*. It leads to simple algebra, calculi and efficient implementations.

By limiting **queries** and **modifications**, it is possible for programmers to describe database **operations** at a very high level, yet have the database management system implement the operation efficiently.

Semi-structured data model: data is organised in tree or graph structures using `XML` which is a way to represent data by hierarchically nested tagged elements.
Tags: define the role played by different pieces of data.
Relation: principal data-structuring concept. *Relation names* begin with a capital letter while *attributes* begin with a lower-case one. Each attribute name is associated with a **domain** (a particular element type).
Schema: name and set of attributes for a relation. A **relational schema** is the relation name, together with attributes and types. A **database schema** is a collection of relational schemas. A particular data for a relation or collection of relations is called an **instance** of the schema.
Key: a set of attributes forms a *key* for a relation if we do not allow two types in a relation instance to have the same values in all the attributes of the key.

`SQL`: principal query language for relational database systems. It has different data types:

- **Char(n)/Varchar(n):** character strings of fixed or varying length.
- **Bit(n):** bit strings of fixed or varying length.
- **Boolean:** attribute whose value is logical (`true, false, unknown`).
- **Int/Integer/Shortint**.
- **Float/Real:** floating point numbers.
- **Date and Time**.

Data declaration: the `CREATE TABLE` statement allows to declare the schema for stored relations (*tables*), specifying attributes, types default values and keys.
Altering schema: the `ALTER` statement add/remove attributes and change the default value associated with it, the `DROP` statement completely eliminate relations or other schema elements.
Declaring keys: we can declare it as a `PRIMARY KEY` or as `UNIQUE`.

Relational algebra: operators and atomic operands. It has:

- $\cup$ **Union:** elements present in both, R or S
- $\cap$ **Intersection:** elements present in both, R and S.
- $-$ **Difference:** set of elements in one but not the other.
- Π **Projection:** removes undesired columns from the argument relation to produce the results.
- σ **Selection:** produces a result consisting of all tuples of the argument relation that satisfy the selection condition.
- $\times$ **Cartesian Product:** set of pairs formed by crossing the first element of the pair to be any element of R and the secondary element of S.
- $\bowtie$ **Natural Join:** splice together those pairs agreeing on all attributes common to the two relations.

- θ **Theta Join:** pairs concatenated if they meet a selection condition associated with it.

Contraints in relational algebra:

- If R is an expression of relational algebra, then $R = \emptyset$ is a constraint that says that *the value of R must be empty.* Hence, there no tuples in the result of R.
- If R and S are expressions of relational algebra, then $R \subset S$ is a constraint that says that *every tuple in the result of R must also be in the result of S.*

Referential integrity constraint: a value appearing in one context also appears in another related context.

5 The Relational Language

SEQUEL: Structured **E**nglish **QUE**ry **L**anguage, is `SQL`. It is based on a *declarative relational calculus*. The most important `SQL` queries can be translated into an equivalent procedural algebraic expression for evaluation.

- `SELECT` statement: retrieves data from one or more (columns included in the final result) or other database objects.
- `FROM` clause: lists the source objects from which data are selected (tables, views, partitions, etc).
- `WHERE` clause: limits conditionally the rows emitted to the query's final result set. Each condition is entered as a comparison of two values or expressions. It is intended to restrict, or reduce, the result.

  After the `WHERE` clause is applied, the detailed result set is obtained.
- `GROUP BY` clause: aggregates the filtered result set available after processing `FROM` and `WHERE`. Any column of any object listed in the `FROM` clause. A **rollup** produces subtotal values, a **cube** produces cross-tabulation values.
- `HAVING` clause: restricts the grouped summary rows to those who satisfy the condition. The `GROUP BY` and
- `HAVING` clauses are interchangeable positionally.
- `ORDER` clause: orders the final set of rows returned by the statement. The size is important (size = rows $\times$ number of bytes per row = total bytes of data in the result set) because small sort should be accomplished entirely in memory whereas large sorts use temporary disk space.
- `INSERT` statement: adds rows to a table, partition or view. It can be a **single-table insert**, where it inserts values into one row of one table (by query or specifying values), or a **multi-table insert**, which inserts rows into one or more tables using a subquery. When `ALL` is specified, the statement performs unconditional multitable insert.
- `UPDATE` statement: changes the column values of existing rows in a table.

  `UPDATE` specifies the table to update
  `SET` specifies which columns are changed and the modified values
  `WHERE` filter conditionally which rows are updated.
- `DELETE` statement: remove rows from a table.

  `DELETE` *(stands alone)* `FROM` identifies the table from which rows are to be deleted
  `WHERE` filters conditions to help determine which rows are deleted.
- `MERGE` statement: combines the ability to update or insert rows into a table by deriving conditionally the rows to be updated or inserted from one or more sources.

The `SQL` language offers many alternatives that can produce the same result set.
The basic `SQL` query blocks map to Relational Algebra as follows:

1. Form the cascade binary **product** of the relations in the `FROM` clause.
2. To this result, apply a **selection** operator whose predicate is the one in the `WHERE` clause.
3. To this result, apply a **projection** whose attribute list is that in the `SELECT` clause.
4. If `DISTINCT` is requested, to this result, apply a **duplicate removal** operator.

6 ER to relational mapping

1. **Mapping of regular entity types**
   For each strong entity type E in the ER schema, create a relation R that includes all the simple attributes of E, and choose one key as the primary one. An **entity relation** is a relation created from the mapping of entity types.

2. **Mapping of weak entity types**
   For each weak entity type, create a relation R and include all simple attributes. include as foreign key attributes of R the primary key attribute(s) of the relation (S) that corresponds to the owner entity type (S). A **primary key** is a combination of primary key (S) of owner and partial key of weak entity type.

3. **Mapping of** $1:1$ **relationship types** For each $1:1$ relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. Three possible approaches:

   (a) **Foreign key approach**
   Choose a relation S 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 of the $1:1$ relationship type R as attributes of S.

   (b) **Merged relation approach**
   Merge two entity types and the relationship into a single relation. Both participations must be total: tables have same number of tuples at all time.

   (c) **Cross-reference or relationship relation approach**
   Relationship relation: 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 adn 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 *drawbacks* are the extra relations and extra joins.

4. **Mapping of** $1:N$ **relationship types** For each regular binary N relationship type R, identify the relation S that represents the participating entity type at the N-side of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R. Each entity instance on the N-side is related to at most one entity instance on the 1-side. Include any simple attributes of the $1:N$ relationship type as attributes of S. An alternative approach is to use relationship relation (cross-reference).

5. **Mapping of binary** $M:N$ **relationship types**
   For each binary $M:N$ relationship type R, create a new relation S to represent R. The primary key is a combination of primary eys of relations as foreign key. Include any simple attribute.
   We cannot represent an $M:N$ relationship type by a single foreign key attribute in one of the participating relations because of the $M:N$ cardinality ratio. We must create a separate relationship relation S.

6. **Mapping of multivalued attributes**
   For each multivalued attribute A, create a new relation R. R is the attribute corresponding to A plus the primary key attribute of the relation.

7. **Mapping of** N**-ary relationship types**
   For each n-ary relationship type R, where $n > 2$, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. The primary key is a combination of all foreign keys that reference the relations representing the participating entity types. If the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E corresponding to E.

8. **Options for mapping specialisation or generalisation**
   For **multiple-relations** we have:

   (a) **Superclass and subclass**
   Create a relation L for C with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\}$ and $PK(L) = K$. Create a relation L_i for each subclass S_i, $i \leq i \leq m$, with the attributes $Attrs(L_i) = \{k\} \cup \{attributes\ of\ S_i\}$ and $PK(L_i) = k$.

   (b) **Subclass relations only**
   Create a relation L_i for each subclass S_i, $i \leq i \leq m$, with the attributes $Attrs(L_i) = \{attributes\ of\ S_i\} \cup \{k, a_1, \ldots, a_n\}$ and $PK(L_i) = k$. This only works for total subclasses and it is only recommended if the specialisation has the disjointedness constraints.

   For **single-relations** we have:

   (a) **One type attribute**
   Create a single relation schema L with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\} \cup \{attributes\ of\ S_1\} \cup \cdots \cup \{attributes\ of\ S_m\} \cup t$ and $PK(L) = k$. t is the **type** or **discriminating** which indicates the subclass to which each tuple belongs. This only works for a specialisation whose subclasses are disjoint. It generates many `NULL` values if many specific attributes exits in the subclasses.

   (b) **Multiple type attribute**
   Create a single relation schema L with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\} \cup \{attributes\ of\ S_1\} \cup \cdots \cup \{attributes\ of\ S_m\} \cup \{t_1, t_2, \ldots, t_n\}$ and $PK(L) = k$. Each t_i, $1 \leq i \leq m$, is a Boolean type attribute indicating whether a tuple belongs to subclass S_i. Used for specialisation whose subclasses are overlapping.

9. **Mapping of union types (categories)**
   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 category. The keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the category.

7 Database Security - Concepts, Approaches and Challenges

Security breaches are typically categorized as *unauthorised data observation*, *incorrect data modification*, and *data unavailability*.
Thus, a complete solution to data security must meet the following three requirements:

1. *Secrecy or confidentiality* - protection of data against unauthorised disclosure.
2. *Integrity* - prevention of unauthorised an improper data modification.
3. *Availability* - prevention and recovery from hardware and software errors and from malicious data access denials making the database system unavailable.

Data protection is ensured by different components of a database management system (DBMS). In particular, an *access control mechanism* ensures data confidentiality.
Semantic correctness is verified by a set of conditions, or predicates, that must be verified against the database state.
Several new areas of database security are emerging as active research topics, such as the development of query techniques for encrypted data and privacy-preserving techniques for the database.

Access control mechanisms of current DBMSs are based on discretionary policies governing the accesses of a subject to data based on the subjects identity and authorisation rules. Discretionary in the sense that they allow subjects to grant authorisations on the data to other subjects: it is thus related to the *authorisation administration policy* (function of granting and revoking authorisations).
authorisation administration in the System R model is based on the ownership approach coupled with administration delegation. Any database user authorised to do so can create a new table. When a user creates a table, he becomes the owner of the table and is solely and fully authorised to exercise all access modes on the table. The owner, however, can delegate privileges on the table to other subjects by granting these subjects authorisations with the so-called *grant option*.
Under the *closed world* policy, whenever a subject tries to access a table and no authorisation is found in the system catalogs, the subject is denied access. Therefore, the lack of authorisation is interpreted as no authorisation. Drawback: the lack of an authorisation for a subject on a table does not prevent this subject from receiving this authorisation some time in the future. Because of this, an explicit *negative authorisation* expresses a denial for a subject to access a table under a specified mode. Conflicts between positive and negative authorisations are resolved by applying the *denials-take-precedence* policy under which negative authorisations override positive authorisations.
Such a model has been further extended with a more flexible conflict resolution policy, based on the concept of *more specific* authorisation. Such a concept introduces a partial order relation among authorisations which is taken into account when dealing with conflicting authorisations. For example, the authorisations granted directly to a user are more specific than the authorisations granted to the groups of which the user is a member. Therefore, a negative authorisation can be overridden by a positive authorisation, if the latter is more specific than the former. If, however, two conflicting authorisations cannot be compared under the order relation, the negative authorisation prevails.
In *Sea View*, authorisations can specify which users or groups are authorised to access particular tables and which users and groups are specifically denied for particular tables. Unlike positive authorisations, negative authorisations cannot specify an access mode. A special access mode, called *null*, is used to denote a negative authorisation. If a subject receives a null access mode on a table, the subject cannot exercise any access mode on the table. Conflicts between positive and negative authorisations are solved on the basis of the following policy:

1. authorisations directly granted to a user take precedence over authorisations specified for groups to which the user belongs.
2. a null mode authorisation given to a subject overrides any other authorisation granted to the same subject.

Content-based access control requires that access control decisions be based on data contents.*Protection views* specifically tailored to support content-based access control and *shorthand views* specifically tailored to simplify query writing.

RBAC models are based on the notion of role. A *role* represents a specific function within an organisation and can be seen as a set of actions or responsibilities associated with this function. All authorisations needed to perform a given activity are granted to the role associated with that activity, rather than being granted directly to users. Users are then made members of roles, thereby acquiring the *role's authorisations*. *Role hierarchies* allow one to represent role-subrole relationships, thus enabling authorisation inheritance and *separation of duty (SoD) constraints* (prevent a subject from receiving too many authorisations).

Mandatory Access Control (MAC) policies regulate accesses to data by subjects on the basis of predefined classifications of subjects and objects in the system. *Objects* are the passive entities storing information, such as relations,tuples in a relation, or elements of a tuple, while *subjects* are active entities performing data accesses. Access control in MAC models is based on:

- *No read-up* - a subject can read only those objects whose access classes are dominated by the access class of the subject.
- *No write-down* - a subject can write only those objects whose access classes dominate the access class of the subject.

Relevant features of further extended systems include:

- Fine-grained flexible authorisation models for complex, multimedia objects.
- Flexible user specification mechanisms based on user credentials and profiles.
- Access control mechanisms tailored to information dissemination strategies and third party publishing architectures.
- Support for distributed cooperative data modifications and complex workflow-based activities.

The most common approach to address the privacy of released data is *anonymisation*: to modify the data by removing all information that can directly link data items with individuals. *Data mining* approaches instead are based on modifying or perturbing the data in some way, and a problem common to most of these techniques is the quality of the resulting database.

Research efforts now need to be devoted on a large number of topics including *data quality and completeness*, *intellectual property rights*, *access control and privacy for mobile users* and *database survivability*.

8 Safeguarding Privacy in a Connected World

Individuals have the right to enjoy effective control over their personal information in a coherent way in the EU, so that in this way individuals are less hesitant to use new services online. This stimulates economic growth, creates new jobs and fosters innovation in the EU.

Right now data protection rules among different Member States is not coherent.

The commission is proposing new rules which will:

- **Improve individuals' ability to control their data**:
  Improve clarity of consent for using data, give a right to be forgotten online, give a right to obtain a copy of the stored data and the freedom to move it across providers, improve information about the use of their personal data.
- **Improve the means for individuals to exercise their rights**:
  Strengthening national data protection authorities' independence and powers in order to act in a faster and more effective way, enhancing administrative and judicial remedies when data protection rights are violated.
- **Reinforce data security**:
  Encouraging the use of privacy-enhancing technologies, privacy-friendly default settings, privacy certification schemes and oblige providers to notify users about data breaches without delays.
- **Enhance the accountability of those processing data**:
  Requiring data controllers to designate a Data Protection Officer in companies with more than 250 employees which deal with risky (privacy-related) processing, introducing the "Privacy by Design" principle, introducing the obligation to carry out Data Protection Impact Assessments.

A strong, clear and uniform legislative framework at EU level will help to unleash the potential of the Digital Single Market and foster economic growth, innovation and job creation. All of this will produce savings for billions of Euro.

The EU's new reformed data protection framework therefore aims to ensure a consistent, high level of data protection to enhance mutual trust between police and judicial authorities of different Member States, thus contributing further to a free flow of data, and effective cooperation between police and judicial authorities.

In cases when EU citizens' data are stored outside the EU, EU laws shall apply putting an effort on simplifying and enhancing the laws on international transfers.

9 Normalization

Redundancy: the information is stored in more than one place within a database.

- **Redundant storage:** some information is stored repeatedly.
- **Update anomalies:** inconsistency is created unless all copies are similarly updated.
- **Insertion anomalies:** some information might not be stored unless others are too.
- **Deletion anomalies:** some information might not be deleted unless others are too.

`NULL` values cannot help eliminate redundant storage or update anomalies, while they can address insertion and deletion anomalies. Redundancy arises when a relation schema forces an association between attributes that is not natural.

A **decomposition of a relation schema** R consists of replacing the relation schema by two (or more) relation schema that each contain a subset of the attributes of R and together include all attributes in R. Two properties:

- **Lossless-join:** enables us to recover any instance of the decomposed relation from corresponding instances of smaller relations.
- **Dependency-preservation:** enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations.

Functional dependency: integrity constraint that generalises the concept of a key. It is a statement about *all* allowable instances.
Integrity constraints, in particular functional dependencies, can be used to identify schemas with redundancy and to suggest refinements (the main refinement technique is decomposition). A **legal** instance must satisfy all specified IC, including all specified FDs. A primary key constraint is a special case of an FD. An FD f is implied by a given set F of FDs if f holds on every relation instance that satisfies all dependencies in F. **Armstrong's Axioms:**

- **Reflexivity:** if $X \supseteq Y$ then $X \to Y$.
- **Augmentation:** if $X \to Y$ then $XZ \to YZ$ for any Z.
- **Transitivity:** if $X \to Y$ and $Y \to Z$ then $X \to Z$.

Complete: repeated application will generate all FDs in the closure F^+. In other words, using the Armstrong's axioms repeatedly to infer dependencies until no more dependencies can be inferred from F, results in a complete set of all possible dependencies that can be inferred from F.
Sound: generate only FDs in F^+ when applied to a set F of FDs.In other words, given a set of FDs, F, specified on relation R, any dependency that we can infer from F holds in every instance of R that satisfies the dependencies in F.
Additional rules that follow from Armstrong's Axioms:

- **Union:** if $X \to Y$ and $X \to Z$ then $X \to YZ$.
- **Decomposition:** if $X \to YZ$ then $X \to Y$ and $X \to Z$.

From previous sections we know that:

- A **superkey** of a relational schema $R = A_1, A_2, ..., A_n$ is a set of attributes with the property that no tuples t_1 and t_2 will have $t_1[S] = t_2[S]$.
- A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.
- If a relational schema has more than one key, each is called **candidate key**. One of the candidate key is arbitrarily designated to be the primary key, while the others secondary keys.
- A **prime attribute** must be a member of some candidate key, otherwise it is a **nonprime attribute**.

- A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X if removal of any attribute A from X means that the dependency does not hold any more.

Attribute closure: set of attributes A such that $X \rightarrow A$ can be inferred using the Armstrong's Axioms.
Normalization: process of decomposing unsatisfactory (i.e. badly designed) relations by breaking up their attributes into smaller relations.
De-normalization: process of storing the join of higher normal form relations as a base relation, which is lower normal form. If a relation is in a certain *normal form* , it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help. As well as detecting redundancies, the presence of FDs in a relation help us identify in which normal form the relation is and decide whether decomposing the relation is worthwhile.

A relation is in **first normal form** if every field contains only atomic values (no lists, no sets).

Second normal form: partial dependencies are not allowed (if it is in 1NF and if every non-prime attribute A in R is fully functional dependent on the primary key, where the primary key contains multiple attributes).

Third normal form: BCNF or A is part of some key for R (if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key.

The relation R is in **Boyce-Codd Normal Form** if, for every functional dependency $X \rightarrow A$, where X is a subset and A an attribute, $A \in X$ OR X is a superkey. Typically, R is NOT in BCNF if there are multiple candidate keys and the keys are composed of multiple attributes (and there are common attributes between the keys).
If a relation is in BCNF, every field of every tuple records a piece of information that cannot be inferred from the values in all other fields in the relation instance.

The 2NF and 3NF still allow presence of redundancy which can be detected via FDs. However, BCNF ensures that no redundancy can be detected via FD information alone, and thus it is the most desirable normal form form the point of view of redundancy.

Partial dependency: X is a proper subset of some key k.
Transitive dependency: X is not a proper subset of any key.
All decompositions used to eliminate redundancy must be lossless.

Decomposition into BCNF:

1. Suppose that R is not in BCNF. let $X \subset R$, A be a single attribute in R, and $X \rightarrow A$ be a functional dependency that causes a violation of BCNF. Decompose R into $R - A$ and XA.
2. If either $R - A$ or XA is not BCNF, decompose them further recursively.

$R - A$: set of attributes other than A in R.
XA: union of attributes in X and A.
A **minimal cover** for a set F of FDs is a set G of FDs such that:

- Every dependency in G is in form $X \rightarrow A$, where A is a single attribute.
- The closure F^+ is equal to the closure G^+.
- If we obtain a set H of dependencies from G by deleting one or more dependencies or by deleting attributes from a dependency in G, $F^+ \neq H^+$.

In order to obtain a minimal cover we can follow this algorithm:

- Put the FDs in a Standard Form using the decomposition axioms.
- Minimise the left side of each FD.
- Delete redundant FDs.

Synthesis: take all the attributes over the original relation R and minimal cover F for the FDS that hold over it and adda a relation schema XA to the decomposition of R for each FD $X \to A$ in F.
The **multivalued dependency** $X \to\to Y$ is said to hold over R if, in every legal instance r of R, each X value is associated with a set of Y values and this set is independent of the values in other attributes.

Fourth normal form: for every multivalued dependency $X \to\to Y$ that holds over R, $Y \supseteq X$ or $XY = R$ or X is a superkey.

A **join dependency** is said to hold over R if $R_1, \ldots, R_n$ is a lossless-join decomposition.

Fifth normal form: for every join dependency $\bowtie \{R_1, \ldots, R_n\}$ that holds over R, $R_i = R$ for some i or the join dependency is implied by the set of those FDs over R in which the left side is a key for R.

Inclusion dependency: some columns of a relation are contained in others.

10 Advanced SQL

Initial versions of SQL were **not computationally complete** (they had no programming constructs), while most recents versions allow **SQL** to be embedded in high level programming languages even though the main problem wiht this approach is **impedance mismatch**.

SQL is a **declarative language** that handles rows of data, whereas a high level language is a procedural language that can handle only one row at a time.

PL/SQL is a block-structured language. **Anonymous blocks** are divided in three parts:

- **Optional declaration part:** variables, constants, cursors and exceptions are defined and possibly initialized.
- **Mandatory executable part:** variables are manipulated.
- **Optional exception part:** exception raised during the executions are handled.

Variables and constants must be declared before they can be referenced in other statements (including other declarative statements), possible to NOT NULL.It possible to use %TYPE to declare a variable to be of the same type of an attribute, and %ROWTYPE to declare one to be of the same type as an entire row. Variable assignment is done with an assignment statement (:=) or with the SQL SELECT or FETCH statement.

SQL supports if statements conditional case, iterative conditions like while and repeat and sequential flow of control mechanisms.

Exception: identifier raised during the execution of a block that terminates its main body of action. The **exception handlers** handle raised exceptions. We declare it as:

```
DECLARE {CONTINUE|EXIT|UNDO} HANDLER
FOR SQLSTATE {sqlstateValue|conditionName|SQLEXCEPTION|
               SQLWARNING|NOTFOUND} handlerAction;

DECLARE conditionName CONDITION
[FOR SQLSTATE sqlstateValue]
```

Cursor allows the rows to be accessed one at a time. It must be *declared* and *opened* before it can be used, and *closed* when not used. Cursors can be parametrised so that the same definition can be reused. Row can be updated or deleted once a cursor has fetched them.
By default, if the Oracle server cannot acquire the locks on the rows in the active set in a SELECT FOR UPDATE cursor, it waits indefinitely. To prevent this, the optional NOWAIT keyword can be specified and a test can be made to see if the locking has been successful.

Subprograms: blocks that can take parameters and be invoked.
Procedure: takes a set of parameters, can modify and return data, might return values to the caller.
Function: takes a set of parameters, can modify and return data, returns a *single value* to the caller.
Package: collection of procedures, functions, variables and SQL statements that are grouped together and stored as a single program unit. In the **specification** we declare all public constructs of a package, in the **body** one defines all constructs of the package (basically it implements the specification).

Trigger: defines action that the database should take when some event occurs in the application. It is defined as:

```
CREATE TRIGGER TriggerName
    BEFORE | AFTER | INSTEAD OF
    INSERT | DELETE | UPDATE [OF TriggerColumnList]
    ON TableName
    [REFERENCING {OLD | NEW} AS {OldName | NewName}
    [FOR EACH {ROW | STATEMENT}]
    [WHEN Condition]
```

```
<trigger action>
```

A trigger is based on the **Event-Condition-Action (ECA)** model where the event(s) trigger the rule, condition check whether the action can be executed and the action is the action to be taken.

Row-level: executes for each row of the table that is affected by the triggering event.
Statement-level: execute only once if multiple rows are affected by triggering event. triggers can also activate themselves one after the other.
Triggers present many advantages:

- Elimination of redundant code.
- Simplifying modifications.
- Increased security.
- Improved integrity.
- Improved processing power.
- Good fit with the client-server architecture.

And they present some disadvantages:

- Performing overhead.
- Cascading effects.
- Cannot be scheduled.
- Less portable.

Atomicity of data: repeating groups are not allowed in the relational model.
Condition: boolean expression referring to literals and to event properties through correlation variables (i.e. new).
Action: can refer to correlation variables, can test the type of event being reacted to but cannot use transaction control commands directly.

11 Transactions, Concurrency Control and Recovery

Transaction: action, or a series of actions, carried out by user or application, which reads or updates contents of a database. It transforms a database from one consistent state to another. A committed transaction cannot be aborted , and an aborted transaction that is rolled back can be restarted later.
Four basic properties, (**ACID**):

- **Atomicity:** a transaction could either complete or have no effect at all.
- **Consistency:** a transaction should correctly transform the database state to reflect the effect of a real world event.
- **Isolation:** the effect of concurrently executing a set of transactions is the same as if they had executed serially (serializable).
- **Durability:** the effect of a transaction on the database state should not be lost once the transaction has committed.

Executing a set of consistent transactions serially is correct, but the performance obtained from this execution might be inadequate. On the other hand, executing a set of consistent transactions concurrently offers performance benefits, but might not be correct.

The **Lost Update Problem** can be avoided by preventing a transaction from reading until after the other transaction updated.

Serialisability: to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution. **Serialisable:** the operations of distinct transactions are on different data items, or the operations of distinct transactions are read operations on the same data.
Non-serialisable: the operations of two distinct transactions are read and write operations (or two write ones) on the same data item.

12 An Overview of Transaction Processing

Isolation is achieved through **serial schedule**: transactions run one after the other.
A **serialisable schedule** is a schedule that is equivalent (produces the same effects, the values returned by the corresponding read operations in the two schedules are the same, the write operations to each data item occur in the same order in both schedules) to a serial schedule.
In general, requests (from different transactions) commute if either of these holds:

- They refer to different data types.
- They are both read requests (a read and write do not commute).

In a serial schedule, a transaction can affect the execution of a subsequent transaction by causing a transition to a new database state.

The **concurrency control** enforces isolation by controlling the schedule of database operations.
Two-phase locking protocol: associates a lock with each item in the database and requires that a transaction hold the lock before it can access the item. The two phases are *lock* and *unlock* and it is *strict*: second phase collapsed to a single point in time when the transaction completes. If a transaction requests to read an item and no other transaction holds a write lock on that item to the transaction and allows the operation to proceed. A **read lock** is a *shared lock* because read operations commute.

Conflict: a transaction requests a read lock when another holds a write one. **Write lock** is an *exclusive lock* as it excludes *all* other locks.
Once a lock has been granted to a transaction, the transaction retains the lock. When the transaction completes, it releases all locks it has been granted. If a request does not conflict with the previously granted ones, it is granted.

The concurrency control uses locks to remember the database operations previously performed by currently active transactions.
Non-strict: unlocking phase starts after the transaction has obtained all of the locks it will ever need and continues until the transaction completes. It guarantees serialisability but problems arise when transaction aborts.

Anomalies:

- **Dirty read:** a value not written by a committed transaction might never appear in the database.
- **Nonrepeatable read:** two reads are not the same if some other writes in between.
- **Lost update:** a transaction releases read lock before having acquired a write one.
- **Uncommitted dependency:** a transaction can see intermediate results of another transaction before it has committed.
- **Inconsistent analysis:** a first transaction reads several values but a second transaction updates some of them during the execution of the first.

Deadlock: a wait loop - a sequence of transactions in which each transaction is waiting to access an item locked by another transaction. In case of deadlock, a transaction in the cycle must be aborted by DBMS.
Timeout: whenever a transaction has been waiting for long, the control assumes that a deadlock exists and aborts the transaction.
Lock: a table might contain thousands of tuples, and locking an entire table because a small number of its tuples are being accessed might result in a serious loss of concurrency. We can lock *only* tuples returned by a select statement.
A **phantom** can lead to nonserialisable schedules and hence invalid results.

Isolation levels from weaker to stronger:

- **Read uncommitted:** no lock needed to read, dirty reads are possible.
- **Read committed:** short read and long write, dirty reads not permitted.

- **Repeatable read:** long read on tuple, non repeatable and dirty reads not permitted.
- **Serialisable:** non repeatable reads, ditry reads and phantoms not permitted.

Latches: internal locks that provide isolation.
Long-lock duration: it can be held until commit time.
Short-lock duration: it can be held only as long as the statement is executed.
Different transactions in the same application can execute at different isolation levels, and each such transactions sees or does not see the phenomena corresponding to its level.

Snapshot: an isolation level that is not part of the ANSI standard, but is provided by at least one commonly used DBMS, it uses a **multiversion** database (old value committed is preserved). A protocol called **first-committer-wins** controls the updates of each transaction. The *first* eliminates dirty reads, T commits if no other transactions are present. Updated a data item that T updated. Committed between T's read request and commit.
Write skew: in snapshot, transactions writes to different items in database. It is not possible to maintain all the different versions of the database.

Granularity: size of the unit locked. It can be **fine** when the unit locked is small (like a tuple lock, which increases the number of locks needed, the time spent requesting it and the space required for the concurrency control) and allows more concurrency (only the data used are locked). It can be **coarse** when the table lock covers all the pages of a table.

Intention lock: weaker than read or write locks:

- **Intention shared:** on a table, to obtain a shared lock on a tuple.
- **Intention exclusive:** to obtain exclusive lock on tuples in table.
- **Shared intention exclusive:** to update some tuples but read all of them.

Lock escalation: a **threshold** is set in the concurrency control that limits the number of fine-grained locks a transaction can obtain on a particular table. When T reaches the threshold, the concurrency control attempts to escalate the fine-grained locks for a single coarse-grained one.
With granular locking:

- If a transaction wants to read one or more tuples in a table, it gets an S lock on entire table.
- If a transaction wants to write one or more tuples in a table, it gets a SIX lock on the table and write locks on the tuples it wants to write.

Atomicity: a transaction either successfully completes or aborts. The transaction aborted must be rolled back using only the information stored or mass storage and all transactions active at the same time of the crash must be aborted.
Durability: after a transaction commits, the changes it makes to the database are not lost.

Log: append-only sequence of records that describes database updates made by transactions and used to restore a database to a consistent state following a failure. It contains begin, update, commit, abort records from a number of transactions running at a particular time. It is stored on a non-volatile storage device, distinct from the mass storage device that contains database (it survives process crash and media failure). Consulted by the system to achieve both atomicity and durability - **update records**.
Undo changes: an update record contains the **before image** which is a physical copy of the item before the change. **Undo record**: update record.
When T is initiated, a *begin record* containing its **transaction ID** is appended to the log. When T commits, a **commit record** is written to the log and an *abort record* if T aborts and the log is scanned backwards. The system periodically appends a **checkpoint record** to the log that lists the current transactions. The update record must always be appended to the log before database updated - **Write-ahead loss**.

Recovery: process of restoring a database to a correct state in the event of a failure. **Cache:** set of page buffers in main memory used to store copies of database pages in active use.
Log records are not immediately written to the log but temporarily stored in log buffer.

The **after image (redo record)** contains the new value of the item written by T.
Mirrored disk: mass storage system which writes the same info on two different disks. Increase **availability**, if one fails, the system continues with the other.
The entire database might be periodically copied (**dumped**) to mass storage. A **fuzzy dump** is performed while the system is operating and transactions are executed.

A **global transaction** is a transaction executed as part of a multidatabase systems (like a distributed one).
A **subtransaction** is a transaction executed as part of a distributed transaction.
Transaction manager: part of transaction processing system that makes distributed transactions atomic. It is the **coordinator** while the database managers are the **cohorts**.
Two-phase commit protocol (without no guarantee global transactions atomic/serialisable):

1. Coordinator prepares message to each cohort (check willingness to commit). Cohort willing to commit, appends a **prepared record** to log buffer. The cohort **force-write** guaranteeing that prepared records are durable.
   Cohorts is in **prepared state** and can reply to the prepare message with a vote message. The vote is **ready** is cohort willing to commit, **aborting** if not. Cohorts cannot change their minds.

2. Coordinator receives each cohort's vote. All votes ready so T can be committed globally and forced a **commit record** to its log. Coordinator sends each cohort a commit message. Cohort forces **commit record** to database manager's log, releases locks. Coordinator receives a done message from each cohort, it appends the **complete record** to log.

Uncertain period: the interval between sending a ready vote message to the coordinator and receiving the commit or abort message back. It involves **blocked cohorts**.
Because long delays generally imply an unacceptable performance penalty, many systems abandon the protocol when the uncertainty period exceeds a given time. To guarantee the serialisability of distributed transactions, we must ensure not only that the subtransaction are serialisable at each site, but that there is an equivalent serial order on which all sites can agree.

Mutual consistency: all the replicas of an item should have the same value. It can be **strong**, when every committed replica *always* have the same value as every other replica, or **weak**, when all committed replica *eventually* will have the same value.
Replica control: part of system that knows which and where replicas are.
Read one/write all- *read value* from the nearest replica and *write* to all replicas:

- **Synchronous-update system:** when a transaction updates an item, all replicas are locked and updated before the transaction commits.

- **Asynchronous-update system:** only one replica is updated before the transaction commits. It can happen as:

  - **Group replication:** a transaction can lock and update any replica. Weak mutual consistency is not guaranteed.

  - **Primary copy replication:** a unique replica of each data is designated as the **primary copy** while the others are **secondary** ones. Update and locks occur on primary copy and then propagated.

Since asynchronous update produces greater transaction throughput than does synchronous update, it is the most widely used form of replication.

13 File Organisation and Indexes

File organisation: physical arrangement of data in a file into records/pages of secondary storage. The **access method** is the series of steps involved in storing and retrieving records from a file.
Heap (unordered) file: stores records in the same order they are inserted. It is *good* because large number of records into a file, *bad* because only selected records are to be retrieved.
Sequential (ordered) file: store records sorted on the values of one or more fields (**ordering fields)**. Rarely used for database storage unless a primary index is added to the file. Inserting and deleting records is problematic because order must be maintained. *Binary search* is used to search, and it represents an improvement over linear search.

Hash file: a **hash function** calculates the address of the page in which the records is to be stored based on one or more fields in the record. The base field is the **hash field**. The main disadvantage is static hashing, but can be solved through **dynamic hashing** (there is a directory which is a binary tree) or **extendible hashing** (there is a directory which is an array of size 2^d, where d is called the **global depth**).

Collisions occur when a new record hashes to a bucket that is already full, and can be resolved as follows:

- **Open addressing:** proceeding from the occupied position specified by the hash address, checks for subsequent empty positions.
- **Chaining:** overflow locations are kept by extending the bucket 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.

Folding: applying an arithmetic function to different parts of the hash field. It is *good* because the retrieval is based on key matching, *bad* because the retrieval is based on pattern matching, range of values, etc.

Index: data structure that allows the DBMS to locate particular records in a file more quickly and thereby speed response to user queries. It can be:

- **Primary index:** the data file is sequentially ordered by an ordering field, and the indexing field is built on the ordering key field, which is guaranteed to have a unique value in each record. It is a *sparse* index as it includes an entry for each disk block of the data file and the key of its anchor record rather than for every search value.
- **Clustering index:** the data file is sequentially ordered on a non-key field, and the indexing field is built on this non-key field (**clustering attribute**), so that there can be more than one record corresponding to a value of the indexing field (*sparse* index).
- **Secondary index**: index defined on a non-ordering fiels of the data file (*dense index*).
- **Sparse:** index record for only some of the search key values in the file.
- **Dense:** index record for every search key value in the file.

Secondary indexes provide a mechanism for specifying an additional key for a base relation that can be used to retrieve data more efficiently. However, there is an overhead involved in the maintenance and use of secondary indexes that has to be balanced against the performance improvement gained when retrieving data.

A **multi-level index** can be created for any type of first-level index as long as the first-level index consists of more than one disk block.

ISAM: two-level index structure by `IBM`. It is more versatile than hashing, supporting retrievals based on exact key match, pattern matching, range of values and part key specification. The

disadvantages are that it is **static** so performance deteriorates as the table is updated. Updated also cause the ISAM file to lose the access key sequence and, retrievals in order of access are slower.
B^+-Tree: file organisation which has dynamic index.
If the relation is not frequently updated or not very large or likely to be, ISAM may be more efficient as it has one less level of index than the B^+-Tree, whose leaf nodes contain record pointers.

Bitmap index: stores a bit vector for each attribute indicating which tuples contains this particular domain value. Each bit that is set to 1 corresponds to a row identifier. This is **space efficient** if number of different domain values is small.

Join index: index on attributes from two or more relations from the same domain. Precomputes the join, no need to perform the join each time the query is run. If the query has high frequency, the performance can be improved.

Cluster: group of one or more tables physically stored together because they share common columns and are often used together. The disk access time is improved. The **cluster key** is stored only once, and so clusters store a set of tables more efficiently than if the tables where stored individually (not clustered).
Oracle supports indexed clusters and hash clusters.

14 DBMS Architectures

Centralised DBS: run on a single computer system and do not interact with others. It can have a **coarse-granularity parallelism**, when 2-4 processors and main memory is shared, or a **fine-granularity parallelism**, when large number of processors and parallelism of tasks.
Centralised systems today act as a server system that satisfies requests generated by client systems.

Client-server DBS: large multi-user systems that aim to satisfy requests generated at many client systems, each (logically) a single-user system.
Back-end: manages access structures, query evaluation and optimisation, recovery and concurrency control.
Front-end: `SQL` User Interface, forms interfaces, report generation tools, data mining, etc.
The back-end and the front-end are interfaced through `SQL`.
Some transaction-processing systems provide a transactional remote procedure call interface to connect clients with a server. All calls enclosed in single transaction.

Storage manager: program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. Responsible for interaction with file manager and efficient storing, retrieving and updating data.

The main phases of **query processing** are parsing and translation, optimization and evaluation. The cost difference between a good and a bad way of evaluating a query can be enormous.

Server system architecture:

- **Transaction-server systems** or **Query-server system**: the client send requests to perform action through an interface, she then get the results back.
  Multiple processes accessing data in shared memory:
  - **Server processes:** receive user queries (transactions), execute them and send results.
  - **Lock manager processes:** implement lock manager functionality (ie. deadlock detection).
  - **Database writer processes:** output modified buffer blocks back to disk continuously.
  - **Log writer processes:** output log records from the log record buffer to stable storage.
  - **Check point processes:** perform periodic checkpoints.
  - **Process monitor processes:** monitoring and recovery action and restarting process.
- **Data-server systems**: allows clients to interact with the server by making requests to read or update data, in units such as files or pages (for database they can be smaller than files). They also provide indexing facilities and transaction facilities. It has several issues:
  - **Page shipping** vs. **Item shipping**.
  - **Adaptive lock granularity**.
  - **Data caching**.
  - **Lock caching**.
- **Cloud-based servers**: **cloud computing**, basically virtual machines, where the service provider runs its own software but runs it on computers provided by another company.

Parallel systems consists of multiple processors and multiple disks connected by a fast interconnection network. They improve processing and I/O speeds by using multiple processors and disks in parallel (many operations are performed simultaneously). They have two performance issues:

- **Throughput:** number of tasks that can be completed in a given time interval.

- **Response time:** the amount of time it takes to complete a single task from the time submitted.

The goal is to improve performance as we increase the degree of parallelism. **Speedup:** running a given task in less time by increasing the degree of parallelism. **Scaleup:** handling larger tasks by increasing the degree of parallelism (usually the most important metric for measuring efficiency). You can do:

- **Batch scaleup:** the size of the database increases, and the tasks are large jobs whose runtime depends on the size of the database.
- **Transaction scaleup:** the rate at which transactions are submitted to the database increases and the size of the database increases proportionally to the transaction rate.

A number of factors work against efficient parallel operation and can diminish both speedup and scaleup:

- **Start-up costs** associated with initiating a single process.
- **Interference** of processes may cause a slowdown.
- **Skew**.

Interconnection networks:

- **Bus:** used by system components to send data and receive data. They do not scale well with increasing parallelism, handle one component at a time.
- **Mesh:** components are nodes in grid connected to the adjacents. Scales better with increasing parallelism.
- **Hypercube:** components numbered in binary and connected one to another if their binary representations differ in exactly one bit. Communications delays significantly lower than in mesh.

Architectural models for parallel machines:

- **Shared memory:** processors and disks have access to a common memory, typically via a bus or through an interconnection network.*Good* - extremely efficient communication between processors. *Bad* - architecture is not scalable beyond 32 or 64 processors since the bus or the interconnection network becomes a bottleneck .
- **Shared disk** (clusters): All processors can directly access all disks via an interconnection network, but the processors have private memories. *Good* - provides a degree of fault-tolerance. *Bad* - bottleneck occurs at interconnection to disk subsystem.
- **Shared nothing:** do not share neither memory nor disks. *Good* - can be scaled up to thousands of processors without interference. *Bad* - cost of communication and non-local disk access and sending data involves software interaction at both ends.
- **Hierarchical:** hybrid of three architectures above.

Pros of distributed database systems:

- **Sharing data.**
- **Autonomy.**
- **Availability.**

Networks: they can be **local-area**, when the processors are distributed over a small geographical area, or **wide-area**, when a number of *autonomous* processors are distributed over a large geographical area.