

Introduction to Databases:

- **Databases and DBMSs:**
- Databases are everywhere, often behind the scenes.
- **DBMS (Database Management System):** A powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time, safely.

Examples of DBMS are:

- IBM DB 2
- Oracle DB
- MongoDB
- MySQL
- PostgreSQL

I.e. A DBMS is a software package designed to define, manipulate, retrieve and manage data in a database. A DBMS generally manipulates the data itself, the data format, field names, record structure and file structure. It also defines rules to validate and manipulate this data.

- **Database:** A collection of data managed by a DBMS.
- **DBMS vs Files:**
- While we can manage a large collection of data with files, and in fact, the first commercial databases evolved in this way, there are some weaknesses/problems with using files.
- The weaknesses/problems are:
 - Retrieving information from files is hard.
 - It can be hard to find the exact information you're looking for.
 - Information can be unorganized.
 - Potential security issues.
- **Data Models:**
- A **data model** is a notation for describing data, including:
 - The structure of the data
 - Constraints on the content of the data
 - Operations on the data
- Data models define how the logical structure of a database is modeled. They define how data is connected to each other and how they are processed and stored inside the system.
- Some specific data models are:
 - **Relational data model**
 - **Semistructured data model**
 - E.g. XML
 - No schema is required and no instance is made.
 - We can immediately write queries on the data.
 - It is a much looser approach.
 - **Unstructured data**
 - E.g. MongoDB
 - Uses (key, value) pairs.
 - Values could be anything, a full document, a video, etc.
 - Does not follow the traditional way of building relations.
 - **Graph data model**
 - Useful for applications such as social networking.

- Every DBMS is based on some data model.
- **Relational Data Model:**
- Is the traditional and one of the most powerful ways to represent data in databases.
- Based on the concept of relations in math.
- We can think of a **relation** as tables of rows and columns.
I.e. A relation can be represented using tables of rows and columns.
- A table has rows and columns, where rows represent records and columns represent the attributes/features.
- A column in a table is also known as an **attribute**.
- A row in a table is also known as a **tuple**.
- E.g. Consider the table below:

Teams	Name	Home Field	Coach
	Rangers	Runnymede CI	Tarvo Sinervo
	Ducks	Humber Public	Tracy Zheng
	Choppers	High Park	Ammar Jalali

Every team has a name, a home field and a coach. These are all features/attributes of the teams.

Each row contains records for each team. The Rangers' home field is Runnymede CI and their coach is Tarvo Sinervo.

- E.g. Here are some datasets that are used for Twitter:
 1. Tweets (ID, Creator ID, Text, Creation Time)

ID	Creator ID	Text	Creation Time

2. User(ID, First Name, Last Name)
 3. Follow(User ID1, User ID2)
 4. Likes(TweetID, Number of Likes)
- **DBMS Language Interfaces:**
 - Different kinds of languages allow different kinds of interaction with a DBMS.

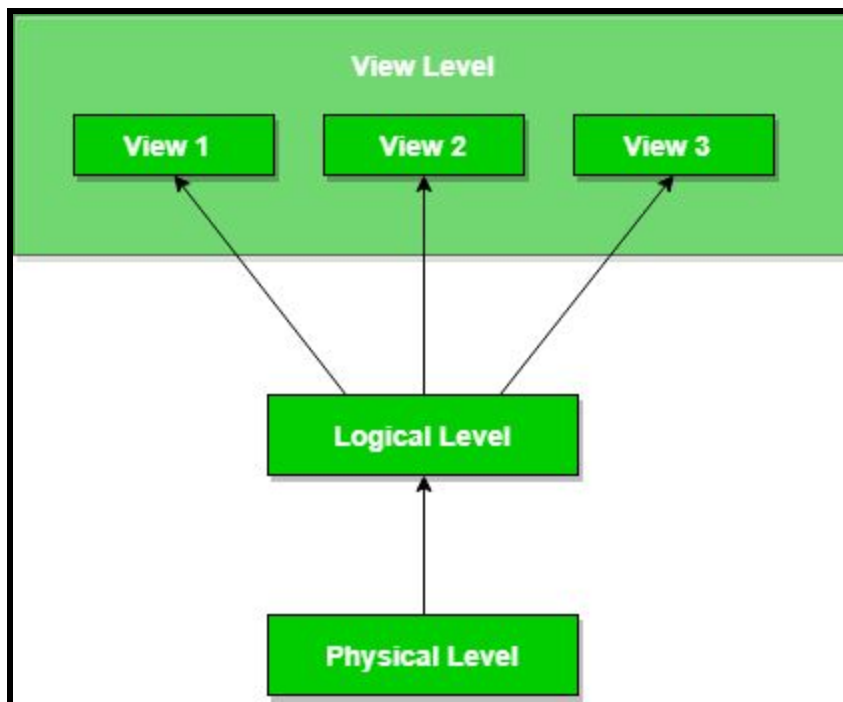
- E.g.
 1. A textual language, such as SQL, can be used in the command line.
 2. A textual language, such as SQL, is embedded in a host language, such as Java.
 3. A textual language is embedded in a 4GL (An ad-hoc language designed for a specific purpose, such as report generation).
 4. A form-oriented language meant to be user friendly, such as QBE.
- **What a DBMS Provides:**
- Ability to specify the logical structure of the data explicitly, and have it enforced.
- Ability to query (search) or modify the data.
 - Going back to the Twitter example. Suppose you wanted to see all the people who have liked at least one of your tweets. If you did this manually, you would have to go through all your tweets and write peoples' names down. This will take a long time. However, if you did a query, it would only take milliseconds.
 - Furthermore, suppose that you want to delete one of your tweets. By modifying the table(s) that contains the tweet, we can delete it.
 - Or, if you wanted to edit a tweet, by modifying the table(s) that contains the tweet, we can edit it.
- Good performance under heavy loads (huge data, many queries).
 - Think of Twitter. Every second, there are millions of accounts posting tweets, liking other tweets, retweeting, deleting tweets, adding people, deleting people, etc. The databases must be able to handle all of these activities.
- Durability of the data.
 - You don't want your data to go away randomly.
 - The data must be safe and must be able to be accessed at any time.
 - Often, there are backups of the data in databases.
- Concurrent access by multiple users/processes.
- **Overall Architecture of a DBMS:**
- The DBMS sits between the data and the users or between the data and an application program.
- Within the DBMS are layers of software for:
 - Parsing queries
 - Implementing the fundamental operations
 - Optimizing queries
 - Maintaining indices on the data. Indices help with accessing data much faster.
 - Accessing the files that store the data and indices
 - Management of buffers
 - Management of disk space
- **Transactions:**
- A **transaction** is a sequence of actions such that either they all execute or none are executed.

I.e. A transaction is a way of representing a state change.

E.g. Suppose you want to withdraw \$500 from your bank account and transfer it to your friend's bank account. To do that, you have first to withdraw the amount from the source account, and then deposit it to the destination account. The operation has to succeed in full. If you stop halfway, the money will be lost, and that is very bad.
- The full set of properties we want from a transaction are called the **ACID properties**.
- **ACID Properties:**
- **Atomicity:** Transactions happen completely or not at all.

- **Consistency:** Transactions must preserve all consistency constraints that have been defined.
I.e. The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Isolation:** Each transaction must appear as if they are executing in isolation, even though many others are executing concurrently.
I.e. In a database system where more than one transaction is being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.
- **Durability:** Once the transaction is complete, its effect persists even if there are failures, such as power failure or system crash, or intentional attacks. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Query Optimization:**
- A DBMS could implement a query many ways, but it wants to find the fastest and most efficient method to do so.
- The DBMS:
 - Tracks table stats like number of rows, number of distinct keys.
 - Maintains indices on the tables using balanced trees, hashing, etc.
 - Tracks index stats like tree height for tree indices.
- A query optimizer uses these to generate efficient execution plans for queries.
- **Concurrent Access:**
- Often, multiple users will simultaneously access 1 account.
E.g. Suppose Fahiem and Margot have two joint accounts
 1. Savings (with \$10,000)
 2. Chequing (with \$2,000)
 Simultaneously, Margot looks up their total while Fahiem does a transfer between accounts. Margot should see \$12,000 no matter what.
- Hence, we need to interleave processes to keep the CPU busy.
- However, in a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We need concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. The DBMS can use **locks** to ensure concurrency control.
- By using locks, it allows users to pretend they are the only user.
- Before reading or writing a piece of data, the transaction must request and wait for a lock on it. A transaction can't have the lock if another transaction has it.
- **Crash Recovery:**
- If a machine crashes in the middle of a transfer of funds, we do not want to lose money.
- DBMSs ensure that every transaction is atomic, meaning either all of it happens, or none of it happens, by using a **log** of all actions.
- Before any change to the DB, the DBMS records it in the log.
- After a crash, for every partially complete transaction, undo all changes.

- **WAL Protocol:**
- **Write-ahead logging (WAL Protocol)** is a family of techniques for providing atomicity and durability, two of the ACID properties, in database systems. The changes are first recorded in the log, which must be written to stable storage, before the changes are written to the database.
- In a system using WAL, all modifications are written to a log before they are applied. Usually both redo and undo information is stored in the log.
- The purpose of this can be illustrated by an example. Imagine a program that is in the middle of performing some operation when the machine it is running on loses power. Upon restart, that program might need to know whether the operation it was performing succeeded, succeeded partially, or failed. If a write-ahead log is used, the program can check this log and compare what it was supposed to be doing when it unexpectedly lost power to what was actually done. On the basis of this comparison, the program could decide to undo what it had started, complete what it had started, or keep things as they are.
- **Summary of Needs and Means:**
- **Data Independence:** Data Independence is defined as a property of DBMS that helps you to change the database schema at one level of a database system without requiring to change the schema at the next higher level. Data independence helps you to keep data separated from all programs that make use of it.
The database has 3 levels:
 1. Physical
 2. Logical
 3. View



Physical Level:

- The lowest level.
- Describes how the data are actually stored.
- You can get the complex data structure details at this level.
- These details are often hidden from the programmers.

Logical Level:

- The middle level.
- Describes what data are stored in the database and what relationships exist between the data.
- Implementing the structure of the logical level may require complex physical low level structures. However, users of the logical level don't need to know about this.

We refer to this as the **physical data independence**.

Physical data independence is one of two types of data independence.

The other is **logical independence**.

Physical data independence helps you to separate logical levels from the physical levels. It allows you to provide a logical description of the database without the need to specify physical structures.

Compared to logical independence, it is easy to achieve physical data independence.

View Level:

- The highest level of abstraction.
- Describes only a small portion of the database.
- Allows users to simplify their interaction with the database system.
- The user just interacts with the system with the help of GUI and enters the details at the screen, they are not aware of how the data is stored and what data is stored.
- **Data Integrity:** Data integrity is the overall completeness, accuracy and consistency of data. This can be indicated by the absence of alteration between two instances or between two updates of a data record, meaning data is intact and unchanged. Constraints on the data can be defined and the DBMS will enforce them.
- **Data Security**
- **Concurrent Access:** Locking
- **Crash Recovery:** WAL protocol to ensure atomicity of transactions.
- **Speed Despite Voluminous Data:** DBMS uses indices and/or query optimization to maximize efficiency.
- **Why not always use a DB:**
 - They are expensive and complicated to set up and maintain.
 - They are general-purpose. Software specifically written for a given task may be better for that task.
- **Roles:**
 - **Database implementers:** Build DBMS software.
 - **Database administrator (DBA):** Sets up and maintains the database.
 - **Application programmers:** Write software that accesses the database.
 - **Sophisticated users:** Write their own queries.
 - **End users:** Use a simple interface, usually with forms.

- **The DBA's role:**
 - Designing the logical schema.
 - Designing the physical schema.
 - Granting access to relations and views.
 - Do backups, log maintenance, failure recovery.
 - Performance tuning.
- **History:**
 - Mid 1960's
 - The first databases were developed.
 - Used the hierarchical data model.
 - The most popular database was IBM's IMS.
 - Early 1970's
 - Started using the network data model.
 - Database programmers followed pointers around the database.
 - Mid-Late 1970's
 - Codd proposes the relational model.
 - Initially, it couldn't compete on performance.
 - 1980's
 - The relational model becomes dominant.
 - Codd wins the Turing Award
 - 1990's
 - SQL
 - The web explodes. Databases must handle huge volumes of transactions 24/7 with high reliability.
 - Early 2000's
 - XML and XQuery


Relational Model:

- **Introduction and Terminology:**
 - The relational model is based on the concept of a relation or table.
 - A **relation** is a table.
 - A column is an **attribute**.
 - A row is a **tuple**.
 - The **arity of a relation** is the number of attributes.
 - The **cardinality of a relation** is the number of tuples.
 - **Domain** is synonymous with data type.
 - An **attribute domain** refers to the data type associated with a column (E.g. Text, Int, etc).
 - **Relations in Math:**
 - A **domain** is a set of values.
 - Suppose D_1, D_2, \dots, D_n are domains.
 - The **cartesian product** of D_1, D_2, \dots, D_n , denoted as $D_1 \times D_2 \times \dots \times D_n$, is the set of all tuples such that $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$.
 - A **mathematical relation** on D_1, D_2, \dots, D_n is a subset of its cartesian product.
 - E.g.
- Let $A = \{p, q, r, s\}$, $B = \{1, 2, 3\}$ and $C = \{100, 200\}$.
- $R = \{ \langle q, 2, 100 \rangle, \langle s, 3, 200 \rangle, \langle p, 1, 200 \rangle \}$ is a relation on A, B, C.

- Our database tables are relations too.

E.g.

Consider the database table below:

Games	Home team	Away team	Home goals	Away goals
	Rangers	Ducks	3	0
	Ducks	Choppers	1	1
	Rangers	Choppers	4	2
	Choppers	Ducks	0	5

{<Rangers, Ducks, 3, 0>, <Ducks, Choppers, 1, 1>, <Rangers, Choppers, 4, 2>, <Choppers, Ducks, 0, 5>} is a relation.

- Relations in math are positional.
E.g. <A, B> is not the same as <B, A>.
- In relational DBs, we name the attributes so position doesn't matter. However, positional notation is still an option in the relational model, and in fact is supported by DBMSs. For example, in SQL, you can refer to a field by position number rather than attribute name.
- **Relation Schemas vs Instance:**
- A **relation schema** is the definition of the structure of the relation. It describes the table name, attributes, the names of the attributes, the domain of the attributes, and the constraints on the attributes. In the schema, by convention we often underline a key.
- The notation for expressing a relation's schema is:

Table Name(Column 1's name, Column 2's name, ..., Column n's name).

E.g. The relation schema for the table shown below

Teams	Name	Home Field	Coach
	Rangers	Runnymede CI	Tarvo Sinervo
	Ducks	Humber Public	Tracy Zheng
	Choppers	High Park	Ammar Jalali
	Crullers	WTCS	Anna Liu

is: **Teams(Name, HomeField, Coach)**

- A **relation instance** is a particular data in the relation. It is a set of tuples that each conform to the schema of the relation. Relation instances do not have duplicate tuples.
Note: Two tuples are identical if all values in both tuples are the same.

E.g.

Suppose we have the below two rows:

Ducks	Humber Park	Tracy Z
Ducks	Humber Park	Anna M

These two tuples are not the same because their last values differ.

- Instances change constantly while schemas rarely change.
- Conventional databases store the current version of the data. Databases that record the history are called **temporal databases**.
- **Database Schemas and Instances:**
- A **database schema** is a collection or set of relation schemas.
- A **database instance** is a collection or set of relation instances.
- **Relations are Sets:**
- A relation is a set of tuples, which means:
 - There can be no duplicate tuples and
 - Order of the tuples doesn't matter.
- In another model, relations are bags, a generalization of sets that allows duplicates. Commercial DBMSs use this model, but for now, we will stick with relations as sets.
- **Superkey and Keys/Candidate Keys:**
- Informally: A **superkey** is a set of one or more attributes whose combined values are unique.

I.e. No two tuples can have the same values on all of these attributes.
- Formally: If attributes a_1, a_2, \dots, a_n form a **superkey** for relation R , \nexists tuples t_1 and t_2 such that $(t_1.a_1 = t_2.a_1) \wedge (t_1.a_2 = t_2.a_2) \wedge \dots \wedge (t_1.a_n = t_2.a_n)$.
- It may have additional attributes that are not needed for unique identification.

If an attribute is already a super key and other attributes get added to the set, then the set is still a super key.

E.g. Suppose the attribute ID is a super key. If we add other attributes to the set, such as name and phone number, the new set is still a super key.
- E.g. of a super key.

Consider the relation schema **Course(dept, number, name, breadth)**.

Suppose our knowledge of the domain tells us that no two tuples can have the same value for dept and number.

This means that {dept, number} is a superkey.

This is a constraint on what can go in the relation.
- **Note:** Every relation has a superkey. At the very worst case, all the attributes make up that superkey. This is because, by definition, no two rows can be identical.
- A **key** or **candidate key** is a minimal superkey. Minimal means that no attributes can be removed from the superkey without making it no longer a superkey.

Furthermore, a candidate key is a super key with no repeated attribute.

In the schema, by convention, we often underline a key.

Note: Since a key is a set of attributes, it can be made up of multiple attributes.

Furthermore, when we underline the key in the schema, we must underline all the attributes that make up the key.

Lastly, the combination of attributes make up the key. The individual attributes themselves do not make up a key.

E.g.

Suppose we have the schema **Person(FirstName, LastName, Address, Age)**.

This means that {FirstName, LastName} is the key.

However, FirstName is not a key and LastName is not a key, but their combination is a key.

- **Note:** A relation can have more than 1 candidate key.
- Properties of candidate keys:
 1. It must contain unique values.
 2. It may have multiple attributes.
 3. It must not contain null values.
 4. It must contain minimum fields to ensure uniqueness.
 5. It must uniquely identify each record in a table.

- E.g.

Consider the relation schema **Course(dept, number, name, breadth)**.

Suppose our knowledge of the domain tells us that no two tuples can have the same value for dept and number.

This means that {dept, number} is a superkey.

However, it also means that {dept, number, name}, {dept, number, breadth}, and {dept, number, name, breadth} are all superkeys.

However, only {dept, number} is a candidate key because it is a minimal superkey.

- **Note:** If a set of attributes is a key for a relation,
 - It does not mean merely that there are no duplicates in a particular instance of the relation.
 - It means that in principle there cannot be any.
 - Only a domain expert can determine that.

Often we invent an attribute to ensure all tuples will be unique, such as SIN, ISBN number, etc.

- A key defines a kind of **integrity constraint**.
- **Foreign Keys:**
- Relations often refer to each other.
- A **foreign key** is a column or group of columns that creates a relationship between two tables. The purpose of foreign keys is to maintain data integrity and allow navigation between two different instances of an entity.
- The referring attribute is called a **foreign key** because it refers to an attribute that is a key in another table.
- This gives us a way to refer to a single tuple in that relation.
- A foreign key may need to have several attributes.
- **Note:** A foreign key can refer to the same relation.

- E.g. In relation Players, team is a foreign key on tID in relation Teams.

Teams	tID	country	olympics	coach
	71428	Canada	2010	Davidson
	10001	Canada	1994	Babcock
	22324	USA	2010	Wilson
Players	pID	name	team	position
	8812	Szabados	71428	goal
	1324	MacLeod	71428	right defense
	9876	Agosta	71428	left wing
	1553	Hunter	10001	centre

- **Declaring Foreign Keys:**
- Notation: $R[A]$ is the set of all tuples from R, but with only the attributes in list A where R is a relation and A is a list of attributes in R.
- We declare **foreign key constraints** this way: $R_1[X] \subseteq R_2[Y]$ where
 1. X and Y may be lists of attributes, of the same arity.
I.e. The number of columns in X must be the same as the number of columns in Y.
 2. Y must be a key in R_2 .
- E.g. Going back to the Teams and Players tables from the previous page, $Players[team] \subseteq Teams[tID]$.
- **Referential Integrity Constraints:**
- A **referential integrity** constraint is specified between two tables.
- In a referential integrity constraint, if a foreign key in Table 1 refers to a key of Table 2, then every value of the foreign Key in Table 1 must be null or be available in Table 2.
I.e. Referential integrity requires that, whenever a foreign key value is used it must reference a valid, existing key in the parent table.
- These $R_1[X] \subseteq R_2[Y]$ relationships are a kind of **referential integrity constraint** or **inclusion dependency**.
- **Note:** Not all referential integrity constraints are foreign key constraints.
 $R_1[X] \subseteq R_2[Y]$ is a foreign key constraint iff Y is a key for relation R_2 .
I.e. What makes a referential integrity constraint a foreign key constraint is that $R_1[X] \subseteq R_2[Y]$ must satisfy the below 2 conditions, especially the second one:
 1. X and Y may be lists of attributes, of the same arity.
 2. Y must be a key in R_2 .
- If Y is not a key, then it's not a foreign key constraint.
It's still a referential integrity constraint, but it's not a foreign key constraint.
Hence, the foreign key constraint is a subset of the referential integrity constraint.
- **Advantages of Relational Model:**
- Simple and elegant.
- Even non-technical users can understand the notion of tables.
- The expression of queries is easy: in terms of rows and columns.
- It supports data independence: can think only in terms of the conceptual schema or view-level schema.