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)

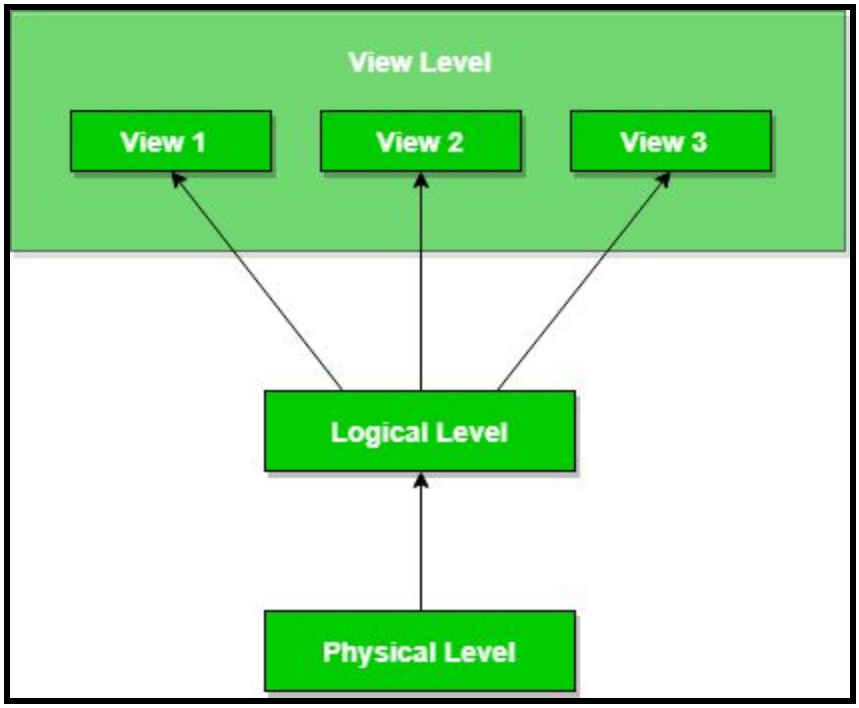
ID	Creator ID	Text	Creation Time

  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, ... D_n are domains.
  The **cartesian product** of D_1, D_2, …, D_n, denoted as $D_1 \times D_2 \times ... \times D_n$, is the set of all tuples such that $d_1 \in D_1$, $d_2 \in D_2$, … ,$d_n \in D_n$.
- A **mathematical relation** on D_1, D_2, … , 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 = {<q, 2, 100>, <s, 3, 200>, <p, 1, 200>} 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, …, a_n form a **superkey** for relation R, ∄ tuples t1 and t2 such that $(t1.a_1 = t2.a_1) \wedge (t1.a_2 = t2.a_2) \wedge \ldots \wedge (t1.a_n = t2.a_n)$.
- 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 above:
  Players[team] ⊆ 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.

Simplifications:

- While learning relational algebra, we will assume the following:
  - Relations are sets, so now two rows are the same.
  - Every cell has a value.
- In SQL, we will drop these assumptions.
- But for now, they simplify our queries.

Relational Algebra Basics:

- Relational algebra is an algebra whose operands are relations or variables that represent relations. Furthermore, operators are designed to do the most common things that we need to do with relations in a database. The result is an algebra that can be used as a query language for relations.
- In relation algebra, operands are tables and operators are what we can do with those tables.
  Some operators are:
  - Select
  - Project
  - Cartesian Product
  - Joins

Select:

- Used for selecting rows based on some condition.
- Notation: $\sigma_c(R)$
- R is a table.
- Condition c is a boolean expression. It can use comparison operators and boolean operators. The operands are either constants or attributes of R.
- The result is a relation with the same schema as the operand but with only the tuples that satisfy the condition.
- E.g. Consider the relation below:

Relation Sells:

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

If I do $\sigma_{bar="Joe's"}(Sells)$, the output or result is:

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75

- E.g. Consider the schema below:

Movies(mID, title, director, year, length)
Artists(aID, aName, nationality)
Roles(mID, aID, character)

Foreign key constraints:
- Roles[mID] ⊆ Movies[mID]
- Roles[aID] ⊆ Artists[aID]

To find all British actors, I will do the following query: $\sigma_{nationality="British"}(Artists)$.
To find all the movies from the 1970s, I will do the following query:
$\sigma_{year \geq 1970 \wedge year < 1980}(Movies)$.

Project:

- Used for selecting columns based on some condition.
- Notation: $\pi_L(R)$
- R is a table.
- L is a subset, not necessarily a proper subset, of the attributes of R.
  I.e. L is a list of attributes that are in R. It could be 1 column or all the columns.
- The result is a relation with all the tuples from R but with only the attributes in L, and in that order.
- It's called project because it gets the columns.
- **Note:** The outcome of the project operator is a set, and thus, there can be no duplicate values.
  E.g. Consider the relation below and the query $\pi_{director}(Movies)$:

Movies

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

The output of the above query is:

Director
Kubrick
Altman
Polanski
Lucas

- E.g. Consider the schema below:

Movies(<u>mID</u>, title, director, year, length)
Artists(<u>aID</u>, aName, nationality)
Roles(<u>mID, aID, character</u>)

Foreign key constraints:

- Roles[mID] ⊆ Movies[mID]
- Roles[aID] ⊆ Artists[aID]

To find the names of all directors of movies from the 1970s, I will do the following query $\pi_{director}(\sigma_{year\geq 1970 \wedge year<1980}(Movies))$. This is called a **nested query**.

- E.g. Consider the relation below and the query $\pi_{beer,price}(Sells)$:

Relation Sells:

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

The result relation is:

beer	price
Bud	2.50
Miller	2.75
Miller	3.00

Cartesian Product:

- Notation: **R1 x R2**
- The result is a relation with every combination of a tuple from R1 concatenated to a tuple from R2.
- Its schema is every attribute from R followed by every attribute of S, in order.
- Suppose there are m attributes in R1 and n attributes in R2. Then, **R1 x R2** has m*n tuples.
- **Note:** If an attribute occurs in both relations, it occurs twice in the result prefixed by relation name.

  E.g. Consider the relations below and the query **R1 x R2**:

  R1

A	**B**
1	2
3	4

R2

B	C
3	5
4	7

The result is:

A	R1.B	R2.B	C
1	2	3	5
1	2	4	7
3	4	3	5
3	4	4	7

- **Note:** Projecting onto fewer attributes can remove what it was that made two tuples distinct. This is because wherever a project operation might "introduce" duplicates, only one copy of each is kept.
  E.g. Consider the relation below and the query π_{age}**(People)**:

People

name	age
Karim	20
Ruth	18
Minh	20
Sofia	19
Jennifer	19
Sasha	20

The result of the query is

age
20
18
19

- Cartesian product can be inconvenient as it can introduce nonsense tuples.

Natural Join:

- Notation: **R ⋈ S**
- The result is defined by:
  - Taking the Cartesian product.
  - Selecting to ensure equality on attributes that are in both relations (as determined by name).
  - Projecting to remove duplicate attributes.

- Some properties of natural joins are:
    1. **Commutative: R ⋈ S = S ⋈ R**
       Although attribute order may vary.
       This will matter later when we use set operations.
    2. **Associative: R ⋈ (S ⋈ T) = (R ⋈ S) ⋈ T**
       So when writing n-ary joins, brackets are irrelevant.
       We can just write: **R1 ⋈ R2 ⋈ . . . ⋈ Rn**
- **Note:** If R and S don't have share any attribute(s) with the same name, then **R ⋈ S** will be the same as **R x S**.
- E.g. Consider the relations below and the query **R ⋈ S**:

R

A	**B**
1	2
2	3

S

C	**D**
5	6
7	9

The result is:

A	**B**	**C**	**D**
1	2	5	6
1	2	7	9
2	3	5	6
2	3	7	9

Here, since R and S don't share an attribute with the same name, **R ⋈ S** is the same as **R x S**.

- E.g. Consider the relations below and the query **R** ⋈ **S**:

R

Number	**Square**
1	1
2	4

S

Number	**Cube**
1	1
2	8

The result is:

Number	**Square**	**Cube**
1	1	1
2	4	8

- E.g. Consider the relations below and the query **R** ⋈ **S**:

R

Number	**Square**
1	1
3	9

S

Number	**Cube**
1	1
2	8

The result is:

Number	**Square**	**Cube**
1	1	1

Here, because R.Number doesn't have 2 and S.Number doesn't have 3, both rows get omitted from the result.

- E.g. Consider the relations below and the query **Sells ⋈ Bars**:

Sells(

bar,	beer,	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

)

Bars(

bar,	addr
Joe's	Maple St.
Sue's	River Rd.

)

The result is:

bar,	beer,	price,	addr
Joe's	Bud	2.50	Maple St.
Joe's	Milller	2.75	Maple St.
Sue's	Bud	2.50	River Rd.
Sue's	Coors	3.00	River Rd.

- E.g. Consider the relations below and the question "How many tuples are in **Artists × Roles**?":

Artists:

aID	aName	nationality
1	Nicholson	American
2	Ford	American
3	Stone	British
4	Fisher	American

Roles:

mID	aID	character
1	1	Jack Torrance
3	1	Jake 'J.J.' Gittes
1	3	Delbert Grady
5	2	Han Solo
6	2	Bob Falfa
5	4	Princess Leia Organa

The answer is 24. There are 4 tuples in Artists and 6 in roles. 4*6 = 24.

Now, with the same 2 relations from above, the answer to the question "How many tuples are in **Artists ⋈ Roles**?" is 6. There will be 2 rows with an aID of 1, 2 rows with an aID of 2, 1 row with an aID of 3 and 1 rot with an aID of 4.

- E.g. Consider the relations below. What is the result of:
  $\pi_{aName}(\sigma_{director="Kubrick"}(Artists \bowtie Roles \bowtie Movies))$?

Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polaski	1974	131
4	Repulsion	Polaski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

Artists:

aID	aName	nationality
1	Nicholson	American
2	Ford	American
3	Stone	British
4	Fisher	American

Roles:

mID	aID	character
1	1	Jack Torrance
3	1	Jake 'J.J.' Gittes
1	3	Delbert Grady
5	2	Han Solo
6	2	Bob Falfa
5	4	Princess Leia Organa

The answer is:

Nicholson
Stone

- Here are 3 special cases for natural join:
  1. **No tuples match:**
     - In this case, the result is a relation with no tuples.
     - E.g.

Employee	Dept
Vista	Sales
Kagani	Production
Tzerpos	Production

Dept	Head
HR	Boutilier

In this example, both relations have the Dept attribute, but no tuples match. Hence, the result of **Table_1 ⋈ Table_2** would be a table with no tuples.

2. **Exactly the same attributes:**
- In this case, we're looking for tuples that are exactly the same in both tables.
- E.g.

Artist	Name
9132	William Shatner
8762	Harrison Ford
5555	Patrick Stewart
1868	Angelina Jolie

Artist	Name
1234	Brad Pitt
1868	Angelina Jolie
5555	Patrick Stewart

Here, the result would be:

Artist	**Name**
1868	Angelina Jolie
5555	Patrick Stewart

This is because only the above 2 rows are in both tables.

3. **No attributes in common:**
- As mentioned previously, the result of this would be the cartesian product of the two tables.

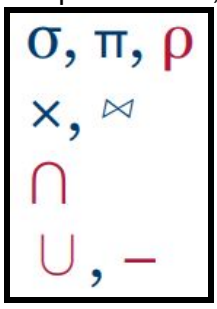
- Natural joins can over-match.
  Natural join bases the matching on attribute names, but what if two attributes have the same name, and we don't want them to have to match?
- Natural joins can also under-match.
  What if two attributes don't have the same name and we do want them to match?

Theta Join:

- It's common to use σ to check conditions after a Cartesian product.
  Theta Join makes this easier.
- Notation: $R \bowtie_{condition} S$
- The result is the same as the Cartesian product followed by select.
  In other words, $R \bowtie_{condition} S = \sigma_{condition}(R \times S)$.
- The word "theta" has no special connotation. It is an artifact of a definition in an early paper. You save just one symbol.
- You still have to write out the conditions, since they are not inferred.

Precedence:

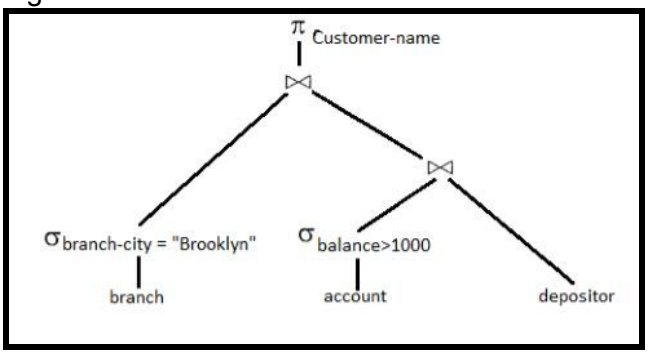
- Expressions can be composed recursively.
- It helps to annotate each subexpression, showing the attributes of its resulting relation.
- Parentheses and precedence rules define the order of evaluation.
- The precedence, from highest to lowest, is:



- Unless very sure, use brackets.

Breaking down expressions:

- Complex nested expressions can be hard to read.
- There are two alternative notations allow us to break them down:
  1. **Expression trees:**
     - Leaves are relations.
     - Interior notes are operators.
     - E.g.



  2. **Sequences of assignment statements:**
     - We can use assignment operators.
     - With assignment operators, we assign an expression to a relation.
     - Notation: **R := Expression**
     - Alternate notation: **R(A1, ..., An) := Expression**
       With this notation, we can rename the column names from the expression.
       I.e. This notation lets you name all the attributes of the new relation.
       Note: The number of columns from the expression must match the number of columns we put in the LHS.
     - **Note:** R must be a temporary variable, not one of the relations in the schema.
       I.e. You are not updating the content of a relation.
     - E.g.

$$\text{CSCoffering} := \sigma_{\text{dept='csc'}}\,\text{Offering}$$
$$\text{TookCSC(sid, grade)} := \pi_{\text{sid, grade}}(\text{CSCoffering} \bowtie \text{Took})$$
$$\text{PassedCSC(sid)} := \pi_{\text{sid}}\,\sigma_{\text{grade}>50}(\text{TookCSC})$$

     - Whether/how small to break things down is up to you. It's all for readability.
     - Assignment helps us break a problem down.
     - It also allows us to change the names of relations and attributes.

Rename:

- Notation: $\rho_{R1}(R2)$ or $\rho_{R1(A1, ..., An)}(R2)$
  The second way lets you rename all the attributes as well as the relation.
- Note that these are equivalent:
  R1(A1, ..., An) := R2
  R1 := $\rho_{R1(A1, ..., An)}(R2)$
- ρ is useful if you want to rename within an expression.

Union:

- Notation: **R U S**
- It includes all tuples that are in tables R or S. It also eliminates duplicate tuples.
- For a union operation to be valid, the following conditions must hold:
  - R and S must be the same number of attributes.

  - The attribute names of R has to match with the attribute names in S.
  - The attribute domains need to be compatible.
  - Duplicate tuples should be automatically removed.
- E.g. Consider the following relations and the query **A U B**:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

The result is:

Table A ∪ B	
column 1	column 2
1	1
1	2
1	3

Intersection:

- Notation: **R ∩ S**
- It includes all tuples that are in both tables R and S.
- For an intersection operation to be valid, the following conditions must hold:
  - The attribute names of R has to match with the attribute names in S.
  - R and S should be union compatible.
  - The result is a relation of all the tuples in both R and S.
- E.g. Consider the following relations and the query **A ∩ B**:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

The result is:

Table A ∩ B	
column 1	column 2
1	1

Difference:

- Notation: **R - S**
- It includes all tuples that are in table R but not in S.
- For a difference operation to be valid, the following conditions must hold:

- The attribute names of R has to match with the attribute names in S.
- R and S should be union compatible.
- The result is a relation of all the tuples in R but not in S.

- E.g. Consider the following relations and the query **A - B**:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

The result is:

Table A - B	
column 1	column 2
1	2

Left Outer Join:

- Notation: $R \bowtie_L S$
- The left outer join operation allows keeping all tuples in the left relation. However, if no matching tuple is found in the right relation, then the attributes of the right relation in the join result are filled with null values.
- E.g. Consider the relations below and the query $R \bowtie_L S$:

R

Number	Square
1	1
3	9

S

Number	Cube
1	1
2	8

The result is:

Number	Square	Cube
1	1	1

3	9	-

Right Outer Join:

- Notation: $R \bowtie_R S$
- The right outer join operation allows keeping all tuples in the right relation. However, if no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.
- E.g. Consider the relations below and the query $R \bowtie_R S$:

R

Number	Square
1	1
3	9

S

Number	Cube
1	1
2	8

The result is:

Number	Square	Cube
1	1	1
2	-	8

Full Outer Join:

- Notation: $R \bowtie_O S$
- In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.
- E.g. Consider the relations below and the query $R \bowtie_O S$:

R

Number	Square
1	1
3	9

S

Number	Cube
1	1
2	8

The result is:

Number	Square	Cube
1	1	1
2	-	8
3	9	-

Division:

- Notation: **R/S**
- It is used when we wish to express queries with “all” or “every”.
- For integers, A/B is the largest int Q st Q x B ≤ A .
- For relations, A/B is the largest relation Q st Q × B ⊆ A.
- It is another “convenience” operator. But if you need it, it is a huge convenience. Defining a query without it is complicated.

Summary of operators:

Operation	Name	Symbol
choose rows	select	σ
choose columns	project	π
combine tables	Cartesian product	×
	natural join	⋈
	theta join	$\bowtie_{condition}$
rename relation [and attributes]	rename	ρ
assignment	assignment	:=

Note: Some operations are not necessary. You can get the same effect using a combination of other operations. An example of this is theta join. We call this **syntactic sugar**.

Expressing Integrity Constraints:

- We’ve used this notation to express inclusion dependencies between relations R1 and R2: **R1[X] ⊆ R2[Y]**.
  Recall that the attributes in X must be a subset of the attributes in Y.
- We can use relational algebra to express other kinds of integrity constraints.
- Suppose R and S are expressions in relational algebra. We can write an integrity constraint in either of these ways:
  1. R = ∅
  2. R ⊆ S (equivalent to saying R - S = ∅)

Summary of techniques for writing queries in relational algebra:

- **Approaching the problem:**
  - Ask yourself which relations need to be involved. Ignore the rest.
  - Every time you combine relations, confirm that:
    1. Attributes that should match will be made to match and
    2. Attributes that will be made to match should match.
  - Annotate each subexpression, to show the attributes of its resulting relation.
- **Breaking down the problem:**
  - Remember that you must look one tuple at a time.
    If you need info from two different tuples, you must make a new relation where it's in one tuple.
  - Use the assignment operator to define intermediate relations.
    - Use good names for the new relations.
    - Name the attributes on the LHS each time, so you don't forget what you have in hand.
    - Add a comment explaining exactly what's in the relation.

Specific types of query:

- **Max (min is analogous):**
  - Pair tuples and find those that are not the max.
  - Then subtract from all to find the maxes.
- **"k or more":**
  - Make all combinations of k different tuples that satisfy the condition.
- **"exactly k":**
  - "k or more" - "(k+1) or more".
- **"every":**
  - Make all combos that should have occurred.
  - Subtract those that did occur to find those that didn't always. These are the failures.
  - Subtract the failures from all to get the answer.

Relational Algebra is procedural:

- A relational algebra query itself suggests a procedure for constructing the result.
  I.e. It describes how one could implement the query.
- We say that it is **procedural**.

Evaluating queries:

- Any problem has multiple RA solutions.
  - Each solution suggests a "query execution plan".
  - Some may seem more efficient than others.
- In RA, we won't care about efficiency; it's an algebra.
- However, in a DBMS, where queries actually are executed, efficiency matters.
  - Which query execution plan is most efficient depends on the data in the database and what indices you have.
  - Fortunately, the DBMS optimizes our queries.
  - We can focus on what we want, not how to get it.
    I.e. Even if we write the queries in a very non-optimized way, the DBMS will optimize it for us.

Relational Calculus:

- Another abstract query language for the relational model.
- Based on first-order logic.
- RC is "declarative". The query describes what you want, but not how to get it.
- Queries look like this: **{t|t ε Movies ∧ t[director] = "Scott"}**

Examples:

Here is the schema:

Schema

Note: "breadth" is a boolean indicating whether or not a course satisfies the breadth requirement for degrees in the Faculty of Arts and Science.

Student(sID, surName, firstName, campus, email, cgpa)

Course(dept, cNum, name, breadth)

Offering(oID, dept, cNum, term, instructor)

Took(sID, oID, grade)

Offering[dept, cNum] ⊆ Course[dept, cNum]

Took[sID] ⊆ Student[sID]

Took[oID] ⊆ Offering[oID]

- **Queries:**

  Write a query for each of the following:

  1. Student number of all students who have taken csc343.

     Soln:

     $\pi_{SID}(\sigma_{dept = \text{“CSC”} \text{ and } cNum = 343}(Took \bowtie Offering))$

  2. Student number of all students who have taken csc343 and earned an A+ in it.

     Soln:

     $Took\_CSC343(SID) := \pi_{SID}(\sigma_{dept = \text{“CSC”} \text{ and } cNum = 343}(Took \bowtie Offering))$
     $Got\_A+(SID) := \pi_{SID}(\sigma_{grade >= 90}(Took))$
     $Took\_CSC343\_And\_GotA+(SID) := TookCSC343(SID) \cap GotA+(SID)$

  3. The names of all such students.

     Soln:

     $Took\_CSC343(SID) := \pi_{SID}(\sigma_{dept = \text{“CSC”} \text{ and } cNum = 343}(Took \bowtie Offering))$
     $Got\_A+(SID) := \pi_{SID}(\sigma_{grade >= 90}(Took))$
     $Took\_CSC343\_And\_GotA+(SID) := TookCSC343(SID) \cap GotA+(SID)$
     $\pi_{surName,\ firstName}(Took\_CSC343\_And\_GotA+ \bowtie Students)$

  4. The names of all students who have passed a breadth course with Professor Picky.

     Soln:

     $SID(sid) := \pi_{sid}(\sigma_{grade \geq 50 \text{ and } instructor = \text{‘Picky’} \text{ and } breadth = True}(Took \bowtie Offering \bowtie Course))$
     $Answer(surName, firstName) := \pi_{surName,\ firstName}(SID \bowtie Student)$

  5. sID of all students who have earned some grade over 80 and some grade below 50.

     Soln:

     $SID\_Over\_80(SID) := \pi_{SID}(\sigma_{grade > 80}(Took))$
     $SID\_Under\_80(SID) := \pi_{SID}(\sigma_{grade < 50}(Took))$
     $SID\_Over\_80 \cap SID\_Under\_80$

6. Terms when Cook and Pitassi were both teaching something.

   Soln:
   Cook_Term(term) := $\pi_{term}(\sigma_{instructor = \text{“Cook”}}(Offering))$
   Pitassi_Term(term) := $\pi_{term}(\sigma_{instructor = \text{“Pitassi”}}(Offering))$
   Cook_Term ∩ Pitassi_Term
7. Terms when either of them was teaching csc463.

   Soln:
   Cook_Term(term) := $\pi_{term}(\sigma_{dept = \text{“csc”} \text{ and } cNum = 463 \text{ and } instructor = \text{“Cook”}}(Offering))$
   Pitassi_Term(term) := $\pi_{term}(\sigma_{dept = \text{“csc”} \text{ and } cNum = 463 \text{ and } instructor = \text{“Pitassi”}}(Offering))$
   Cook_Term U Pitassi_Term
8. sID of students who have earned a grade of 85 or more, or who have passed a course taught by Atwood.

   Soln:
   85_or_more(SID) := $\pi_{SID}(\sigma_{grade >= 85}(Took))$
   Passed_Atwood(SID) := $\pi_{SID}(\sigma_{grade >= 50 \text{ and } instructor = \text{“Atwood”}}(Took \bowtie Offering)$
   85_or_more U Passed_Atwood
9. Terms when csc369 was not offered.

   Soln:
   All_Term(Term) := $\pi_{Term}(Offering)$
   CSC369_Offered_Terms(Term) := $\pi_{Term}(\sigma_{dept=\text{”CSC”} \text{ and } cNum=369}(Offering))$
   All_Term - CSC369_Offered_Terms
10. Department and course number of courses that have never been offered.

    Soln:
    All_Courses(Dept, cNum) := $\pi_{dept,\ cNum}(Course)$
    Offered_Courses(Dept, cNum) := $\pi_{dept,\ cNum}(Offering)$
    All_Courses - Offered_Courses
11. SIDs and surnames of all pairs of students who've taken a course together.

    Soln:
    -- T1 and T2 are the same as Took.
    T1 := $\rho_{T1}(Took)$
    T2 := $\rho_{T2}(Took)$

    -- Gets pairs of sids s.t. they are taking the same class.
    -- **Note:** We use T1.sid < T2.sid and not T1.sid != T2.sid because we don't want
    -- duplicates. I.e. If student A and B are taking the class together, and we have
    -- A.sid and B.sid, we don't also want B.sid and A.sid.
    Pairs(sid1, sid2) := $\pi_{T1.sid,\ T2.sid}(\sigma_{T1.sid < T2.sid \text{ and } T1.oid = T2.oid}(T1 \times T2))$

    -- Gets the surname of the first student.
    FirstName(sid1, sid2, name1) := $\pi_{sid1,\ sid2,\ surname}(\sigma_{sid1 = sid}(Pairs \times Student))$

    -- Gets the surname of the second student.
    $\pi_{sid1,\ sid2,\ name1,\ surname}(\sigma_{sid2 = sid}(FirstName \times Student))$

12. sID of student(s) with the highest grade in csc343, in term 20099.

    Soln:
    Takers(sid, grade) := $\pi_{sid, grade}$(Took $\bowtie_{dept="CSC" \text{ and } cNum=343 \text{ and } term = 20099}$ Offering)
    NotTop(sid) := $\pi_{T1.sid}$($\sigma_{T1.grade < T2.grade}$($\rho_{T1}$Takers x ρ_{T2}Takers))
    Answer(sid) := π_{sid}(Takers) - NotTop
13. sID of students who have a grade of 100 at least twice.

    Soln:
    Answer(sid) := $\pi_{T1.sid}$($\sigma_{T1.sid == T2.sid \text{ and } T1.oid != T2.oid \text{ and } T1.grade=100 \text{ and } T2.grade=100}$($\rho_{T1}$Took x ρ_{T2}Took))
14. sID of students who have a grade of 100 exactly twice.

    Soln:
    At_Least_Twice(sid) := $\pi_{T1.sid}$($\sigma_{T1.sid = T2.sid \text{ and } T1.oid != T2.oid \text{ and } T1.grade=100 \text{ and } T2.grade=100}$($\rho_{T1}$Took x ρ_{T2}Took))
    At_Least_Thrice(sid) := $\pi_{T1.sid}$($\sigma_{T1.sid = T2.sid = T3.sid \text{ and } T1.oid != T2.oid \text{ and } T1.oid != T3.oid \text{ and } T2.oid != T3.oid \text{ and } T1.grade=100 \text{ and } T2.grade=100 \text{ and } T3.grade-100}$($\rho_{T1}$Took x ρ_{T2}Took x ρ_{T3}Took))
    Answer(sid) := At_Least_Twice - At_Least_Thrice

    Note: Since != isn't transitive, we can't do T1.oid != T2.oid != T3.oid.
15. sID of students who have a grade of 100 at most twice.

    Soln:
    Answer(sid) := π_{sid}(Student) - At_Least_Thrice
16. Department and cNum of all courses that have been taught in every term when csc448 was taught.

    Soln:
    448Terms(Term) := π_{term}($\sigma_{dept="CSC" \text{ and } cNum=448}$Offering)
    DidHappen(dept, cNum, term) := $\pi_{dept, cNum, term}$(Offering)
    ShouldHappen(dept, cNum, term) := $\pi_{dept, cNum}$(DidHappen) x 448Terms
    Didn'tHappen(dept, cNum, term) := ShouldHappen - DidHappen
    Answer(dept, cNum) := $\pi_{dept, cNum}$(DidHappen) - $\pi_{dept, cNum}$(Didn'tHappen)
17. Name of all students who have taken, at some point, every course Gries has taught (but not necessarily taken them from Gries)

    Soln:
    CoursesByGries(dept, cNum) := $\pi_{dept, cNum}$($\sigma_{instructor="Gries"}$Offering)
    ShouldHappen(sid, dept, cNum) := π_{sid}Student x $\pi_{dept, cNum}$CoursesByGries
    Didn'tHappen(sid) := π_{sid}Student - π_{sid}ShouldHappen
    Answer(surName, firstName) := $\pi_{surName, firstName}$((π_{sid}(Student)-Didn'tHappen) $\bowtie$ Student)

- **Integrity Constraints:**
  Express the following constraints using the notation R = ∅ or R - S = ∅:
  1. Courses at the 400-level cannot count for breadth.

     Soln:
     $\sigma_{\text{cNum >= 400 and cNum < 500 and Breath = True}}(\text{Course}) = \emptyset$
  2. CSC490 can only be offered at the same time as CSC454.

     Soln:
     $\text{490Terms(Term)} := \pi_{\text{Term}}(\sigma_{\text{dept = "CSC" and cNum = 490}}(\text{Offering}))$
     $\text{454Terms(Term)} := \pi_{\text{Term}}(\sigma_{\text{dept = "CSC" and cNum = 454}}(\text{Offering}))$
     $\text{490Terms - 454Terms} = \emptyset$

Relational Algebra:

- We can perform queries on a set of relations to get information from them. A **query** is a request for data or information from a relation. The input is a relation and the output is a new relation.
- An algebra is a mathematical system consisting of the following:
    1. **Operands:** Variables or values from which new values can be constructed.
    2. **Operators:** Symbols denoting procedures that construct new values from given values.
- **Relational algebra** is a widely used procedural query language. It collects instances of relations as input and gives occurrences of relations as output. It uses various operations to perform this action. It is an algebra whose operands are relations or variables that represent relations. Operators are designed to do the most common things that we need to do with relations in a database.
- Relational algebra operations are performed recursively on a relation. The output of these operations is a new relation, which might be formed from one or more input relations. Relational algebra operations do not modify the input relation in any way.

SELECT (σ):

- The SELECT operation is used for selecting a subset of the tuples according to a given selection condition. It is denoted by **$\sigma_p(x)$**. It is used as an expression to choose tuples which meet the selection condition. The select operation selects tuples that satisfy a given predicate.
- Notation: **$\sigma_p(x)$**
    - σ is the selection predicate.
    - x is the name of the relation.
    - p is the propositional logic. It is a boolean formula of terms and connectives.
      These connectives are: **^(and), V(or), ~(not)**.
      The operators are: **<, >, ≤, ≥, =, ≠**
      These terms are: **attribute operator attribute** and **attribute operator constant**.
- E.g. Consider the below relation.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Instructor Relation

If we do $\sigma_{SALARY >= 85000}$(instructor), we get all the tuples with attribute salary at least 85000

from the instructor relation.
I.e. We would get this as the output:

ID	name	dept_name	salary
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

PROJECTION (π):

- The projection operation gets the specified attributes from a relation.
- Notation: $\pi_{A1, A2, An}(r)$
  - π denotes the project operation.
  - A1, A2, An are the attributes in the relation, r.
  - r is the name of the relation.
- E.g. Consider the relation below:

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Instructor Relation

If we do: $\pi_{ID,salary}$(instructor), it would get the attributes ID and salary from the instructor relation.
I.e. This would be the output:

ID	salary
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

NATURAL JOIN (⋈):

- Combines two relations into a single relation.
- Can only be performed if there is a common attribute between the relations. The name and domain of the attribute must be the same. Note that if there is an entry in only one relation, it will be omitted from the result relation.
- Also called inner join.
- Notation: **r ⋈ s**
- E.g. Consider the two relations below:

Instructor Relation

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

and Department Relation

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Since they both have the attribute dept_name and since the domain of both dept_name is string, if we do instructor ⋈ department, we get the following output:

ID	*name*	*salary*	*dept_name*	*building*	*budget*
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Theta-Join($\bowtie_C$):

- Theta join combines tuples from different relations provided they satisfy the theta condition.
- Notation: **r $\bowtie_C$ s**
- R3 = R1 $\bowtie_C$ R2
  - Take the product R1 X R2.
  - Then apply σ_C to the result. As for σ, C can be any boolean-valued condition.
- E.g.

Left Outer Join ($\bowtie_L$):

- The left outer join operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.
- Notation: **R$\bowtie_L$S**

Right Outer Join ($\bowtie_R$):

- The right outer join operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.
- Notation: **A$\bowtie_R$B**

Full Outer Join ($\bowtie_O$):

- In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.
- Notation: **A$\bowtie_O$B**

CARTESIAN PRODUCT (x):

- The cross product of 2 relations. The cross product produces all possible pairs of rows of the two relations.
- This is used to merge columns from two relations. Generally, a cartesian product is never a meaningful operation when it performs alone. However, it becomes meaningful when it is followed by other operations.
- Notation: **r x s**
- E.g. The cross product of {a, b} and {c, d} is {a,c}, {a,d}, {b,c} and {b,d}.

- E.g. Consider the 2 relations below:

A	B
α	1
β	2

r

and

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

If we do r x s, we get the following output:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

- A problem arises when the 2 relations share a same attribute name. How would we differentiate between the 2 attributes? We can rename the attributes of the relations.

RENAME (ρ):

- Notation: **$\rho_x(E)$**
  - E is the relation name.
  - x is what will be prepended to all the attribute names in relation E.
  - The rename operation renames all attributes in relation E by prepending them with x.
- E.g. Suppose the below table is the relation r.

A	B
a	1
b	2

If I do $P_r(r) \times P_s(r)$, we get the following relation:

r.A	r.B	s.A	s.B
a	1	a	1
a	1	b	2
b	2	a	1
b	2	b	2

UNION (U):

- Union operator when applied on two relations R1 and R2 will give a relation with tuples which are either in R1 or in R2. Furthermore, it eliminates all duplicate tuples.
  I.e. The tuples that are in both R1 and R2 will appear only once in the result relation.
- For a union operation to be valid, the following conditions must hold:
  1. The 2 relations must have the same **arity** (same number of attributes).
  2. The attribute domains must be compatible.
     I.e. The ith column of relation 1 must be of the same domain as the ith column of relation 2.
  3. Duplicate tuples are automatically eliminated.
- Notation: **r U s**
- E.g. Consider the 2 relations below:

A	B
α	1
α	2
β	1

r

and

A	B
α	2
β	3

s

If we do r U s, we will get the output:

A	B
α	1
α	2
β	1
β	3

- E.g. Consider the 2 relations below:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

If we do A U B, we get the following output:

Table A ∪ B	
column 1	column 2
1	1
1	2
1	3

DIFFERENCE (-):

- Returns a relation consisting of all the tuples which are present in the first relation but are not in the second relation.
- Notation: **r - s**
- E.g. Consider the 2 relations below:

A	B
α	1
α	2
β	1

r

and

A	B
α	2
β	3

s

If we do r - s, we will get the following output:

A	B
α	1
β	1

- E.g. Consider the 2 relations below:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

If we do r - s, we will get the following output:

Table A - B	
column 1	column 2
1	2

INTERSECTION (∩):

- Defines a relation consisting of a set of all tuple that are in both A and B, where A and B are 2 relations.
- Notation: **A ∩ B**
- E.g. Consider the 2 relations below:

A	B
α	1
α	2
β	1

r

and

A	B
α	2
β	3

s

If we do r ∩ s, we will get the output:

A	B
α	2

- E.g. Consider the 2 relations below:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

If we do A ∩ B, we will get the output:

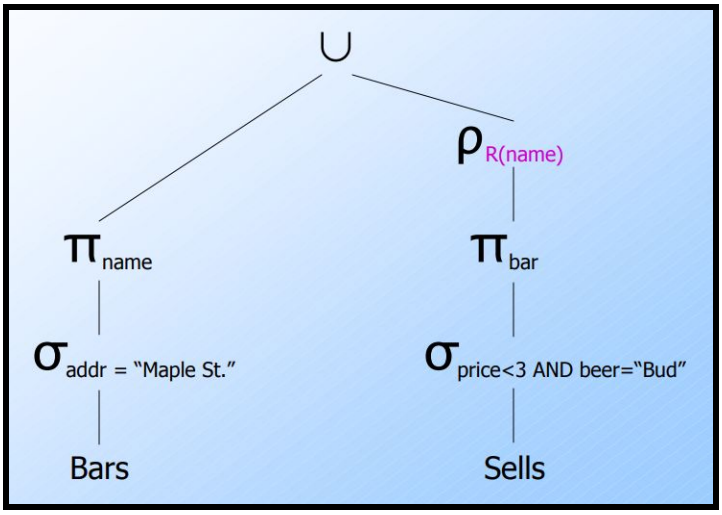
Table A ∩ B	
column 1	column 2
1	1

- **Note:** r ∩ s = r–(r–s)

Building Complex Expressions:

- Combine operators with parentheses and precedence rules.
- Three notations:
  1. **Sequences of assignment statements:**
     a. Create temporary relation names.
     b. Renaming can be implied by giving relations a list of attributes.
     c. E.g. R3 = R1 $\bowtie_C$ R2 can be written as:
        R4 = R1 X R2
        R3 = σ_C (R4)
  2. **Expressions with several operators:**
     a. E.g. R3 = R1 $\bowtie_C$ R2 can be written as R3 = σ_C (R1 X R2).
     b. Precedence of relational operators:
        1. [σ, π, ρ] (Highest)
        2. [X, $\bowtie$]
        3. ∩
        4. [∪ , —] (Lowest)
  3. **Expression trees:**
     - Leaves are operands. Variables stand for relations.
     - Interior nodes are operators, applied to their child or children.
     - E.g. Using the relations Bars(name, addr) and Sells(bar, beer, price), find the names of all the bars that are either on Maple St. or sell Bud for less than $3.

       Solution:



Summary of Relational Algebra Operations:

Operation	**Purpose**
Select(σ)	The select operation is used for selecting a subset of the tuples according to a given selection condition
Projection(π)	The projection eliminates all attributes of the input relation but those mentioned in the projection list.
Union Operation(∪)	It includes all tuples that are in tables A or in B.
Difference(-)	The result of A - B, is a relation which includes all tuples that are in A but not in B.
Intersection(∩)	Intersection defines a relation consisting of a set of all tuple that are in both A and B.
Cartesian Product(X)	Cartesian product merges columns from two relations.
Theta Join($\bowtie_C$)	The general case of JOIN operation is called a Theta join.
Natural Join($\bowtie$)	Natural join can only be performed if there is a common attribute (column) between the relations. Same as inner join.
Left Outer Join($\bowtie_L$)	In left outer join, the operation allows keeping all tuple in the left relation.
Right Outer join($\bowtie_R$)	In right outer join, the operation allows keeping all tuple in the right relation.
Full Outer Join ($\bowtie_O$)	In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.
Rename(ρ)	Renames the attributes of the relation.

Examples of Joins in Relational Algebra

Suppose we have the following 2 relations below:

A

Num	Square
1	1
2	4
3	9

B

Num	Cube
2	8
3	27
4	64

Natural Join:

$A \bowtie B$ will get us the relation:

Num	Square	Cube
2	4	8
3	9	27

Left Join:

$A \bowtie_L B$ will get us the relation:

Num	Square	Cube
1	1	Null
2	4	8
3	9	27

Right Join:

$A \bowtie_R B$ will get us the relation:

Num	Square	Cube
2	4	8
3	9	27
4	Null	64

Examples of Joins in Relational Algebra

Full Outer Join:

A $\bowtie_O$ B will get us the relation:

Num	**Square**	**Cube**
1	1	Null
2	4	8
3	9	27
4	Null	64

Cartesian Join:

A X B will get us the relation:

A.num	A.square	B.num	B.cube
1	1	2	8
1	1	3	27
1	1	4	64
2	4	2	8
2	4	3	27
2	4	4	64
3	9	2	8
3	9	3	27
3	9	4	64

Relational Algebra Examples

Consider the 2 relations below:

A

Num	Square
1	1
2	4

B

Num	Cube
1	1
3	27

1. Select:
   - Denoted by $\sigma_c(R)$ where
     - σ is the select symbol.
     - c is the condition/propositional logic.
     - R is the name of the relation.
   - Select gets all tuples from the given relation that satisfies the condition.
   - E.g. 1 $\sigma_{Num>1}(A)$ will give us

Num	Square
2	4

   - E.g. 2 $\sigma_{Num=3}(B)$ will give us

Num	Cube
3	27

   - E.g. 3 $\sigma_{Num<2}(B)$ will give us

Num	Cube
1	1

2. Project:
- Denoted by $\Pi_{col1, col2, ..., coln}(R)$ where
  - Π is the project symbol.
  - col1, ..., coln are the names of attributes of the given relation.
  - R is the name of the relation.
- Project gets all the attributes listed.
- E.g. 4 $\Pi_{Num}(A)$ will give us

Num
1
2

3. Natural Join:
- Denoted by $R \bowtie S$ where $\bowtie$ is the natural join symbol and R and S are relations.
- Also called inner join.
- It joins 2 relations based on same column name(s), and same values in those columns.

Note: If the 2 relations have no common attribute name, the result is the cartesian product of the 2 relations.
- E.g. 5 $A \bowtie B$ gives us

Num	Square	Cube
1	1	1

Here, both A and B have an attribute called Num. We join the tables based on that col. Next, we try to find all values that are in both relation's Num col. The only value is 1. Hence, we have a row with 1 under the Num col in the result relation.

Furthermore, since 2 is only in A and 3 is only in B, we omit those.

- E.g. 6 Consider these 2 new relations below

C

A	B
1	2
2	3

D

x	y
4	6
3	5

C ⋈ D gives us

A	B	x	y
1	2	4	6
1	2	3	5
2	3	4	6
2	3	3	5

Here, since C and D do not share any cols, the result is C × D or the cartesian product of C and D.

- E.g. 7 Consider these 2 new relations below

E

A	B
1	2
2	3

F

A	B
1	3
2	3

E ⋈ F gives us

A	B
2	3

Here, since A and B have the same 2 cols, we need to use row(s) s.t. both values of those row(s) are the same for E and F. While both E and F have a 1 in their A col of their first row, that row's B value is different for E and F, so we can't use it. E and F's second row have the same values so we use those values.

4. Cartesian Product:
- Denoted by $R \times S$ where R and S are relations.
- Also called cross join.
- Produces all possible pairs of rows of the 2 relations.
- If R has m rows and S has n rows, then $R \times S$ has $m \cdot n$ rows.
- E.g. & $A \times B$ gives us

A.num	Square	B.num	Cube
1	1	1	1
1	1	3	27
2	4	1	1
2	4	3	27

5. Left Join:
- Denoted by $R \bowtie_L S$ where R and S are relations.
- keeps all the tuples from the first relation and tries to find matching tuples from the second relation. If there is no matching tuple, a null value is used.
- E.g. 9 $A \bowtie_L B$ gives us

Num	Square	Cube
1	1	1
2	4	Null

Since B doesn't have a 2 in its Num col, its value for col Cube when Num = 2 is Null.

6. Right Join:
- Denoted as $R \bowtie_R S$ where R and S are relations.
- Like left join, but this time, we keep all the tuples from the right relation and try to match tuples from the left relation.
- E.g. 10 $A \bowtie_R B$ gives us

Num	Cube	Square
1	1	1
3	27	Null

7. Full Join:
- Denoted by $R \bowtie_o S$ where R and S are relations.
- All tuples from both relations are included in the result.
- E.g. 11 $A \bowtie_o B$ gives us

Num	Square	Cube
1	1	1
2	4	Null
3	Null	27

8. Theta Join:
- Denoted by $R \bowtie_c S$ where R and S are relations and c is a condition.
- With $R \bowtie_c S$, you first do $R \times S$ and then σ_c on that.
- E.g. 12 $A \bowtie^{A.}_{Num>1} B$ gives us

A.num	Square	B.Num	Cube
2	4	1	1
2	4	3	27

9. Union:
- Denoted by R ∪ S where R and S are relations.
- Gives a relation with the tuples that are in R or S.
- Will eliminate duplicate tuples.
- E.g. 13 Consider R and S below:

R

A	B
1	2

S

A	B
3	4

R ∪ S

A	B
1	2
3	4

Note: For union, diff and intersection, the relations must have the same num of cols, and cols with the same names and order.

10. Difference:
- Denoted by R-S where R and S are relations.
- Gives a relation with all the tuples only in R and not in S or both R and S.
- E.g. 14 Consider R and S below

R

A	B
1	2
3	4

S

A	B
3	4
4	5

R-S

A	B
1	2

11. Intersection:
- Denoted by R ∩ S where R and S are relations.
- Gives a relation with all tuples in both R and S.
- R ∩ S is equivalent to R-(R-S).
- E.g. 15 Consider R and S below

R

A	B
1	1
2	3

S

A	B
1	1
3	4

R ∩ S

A	B
1	1

12. Finding "every" using relational algebra:

Consider the 2 relational schemas below:

Student(id, name)
Marks (id, class, mark)

Find the id of the students taking every class.

Soln:

1. $R_1 = \Pi_{id}(S) \times \Pi_{class}(M)$
2. $R_2 = R_1 - \Pi_{id, class}(M)$
3. $R_3 = \Pi_{id}(R_1) - \Pi_{id}(R_2)$

R_1 is a relation with every possible permutation of id and classes. Therefore, it is a table of every student taking every class.

When you do $R_1 - \Pi_{id, class}(M)$, you are removing all instances of a student actually taking a class from R_1. Therefore, R_2 is a table of the students not taking class(es). If a student is taking every class, their id wouldn't be in R_2.

Since $\Pi_{id}^{(R_1)}$ contains the id of every student and $\Pi_{id}^{(R_2)}$ contains the id of the students who are not taking some classes, $\Pi_{id}^{(R_1)} - \Pi_{id}^{(R_2)}$ will give us a table of the ids of the students taking every class.

13. Finding the biggest or highest:

Using the schemas in 12, find the id of the students who has the highest mark.

Soln:

1. $R_1 = \Pi_{id, mark}^{(M)}$
2. $R_2 = \rho_{R_2}^{(R_1)}$
3. $R_3 = \rho_{R_3}^{(R_1)}$
4. $R_4 = R_2 \bowtie_{(R_2.mark < R_3.mark)} R_3$
5. $R_5 = \Pi_{id}^{(R_1)} - \Pi_{R_2.id}^{(R_4)}$

R_1 is just a relation with only the id and mark columns from Marks. R_2 and R_3 are just renamed instances of R_1.

R_4 is a relation of all possible permutations between R_2 and R_3 where $R_2.mark < R_3.mark$. That means $R_2.id$ in R_4 does not contain the id of the students with the highest mark.

However, R2.id in R4 contains the id of all other students. Therefore, $\Pi_{id}(R_1) - \Pi_{R2.id}(R_4)$ gets back a table with the id of the students with the highest mark.

14 Finding the second biggest or second highest:

Using the schemas in 12, find the id of the students who have the second highest mark.

Soln:

$R_1 = \Pi_{id,\ mark}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = \rho_{R3}(R_1)$

$R_4 = R_2 \bowtie (R2.mark < R3.mark)\ R_3$

$R_5 = \Pi_{R2.id,\ R2.mark}(R_4)$

$R_6 = \rho_{R6}(R_5)$

$R_7 = R_5 \bowtie (R5.mark < R6.mark)\ R_6$

$R_8 = \Pi_{id}(R_5) - \Pi_{R5.id}(R_7)$

The steps R_1 to R_4 are used to remove the id of the students with the highest mark. In R4, R2.id is a relation that does not contain the id of the students with the highest mark. R5 is a table consisting of the R2.id and R2.mark cols from R4.

As such, it doesn't have the id of the students with the highest mark. R6 is a renamed instance of R5. R7 is a relation where R5.id doesn't contain the id of the student with the current highest mark. They used to be the students with the second highest marks. R8 is a table with the id of the students with the second highest mark. $\Pi_{id}(R5)$ contains all the id of the students that do not have the highest mark. $\Pi_{R5.id}(R7)$ contains the id of the students who do not have the highest or second highest marks. Therefore, $\Pi_{id}(R5) - \Pi_{R5.id}(R7)$ gets you the id of the students who have the second highest mark.

15. Finding "at least" using relational algebra.

Using the 2 schemas from 12, find the id of the students taking at least 2 courses.

Soln:

$R_1 = \Pi_{id, class}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = R_1 \bowtie (R1.id = R2.id \text{ and } R1.class != R2.class) R2$

You're finding all the rows of $R_1 \times R_2$ where R1.id equals to R2.id but R1.class doesn't equal to R2.class.

Using the schemas from 12, find the id of the students taking at least 3 classes.

Soln:

$R_1 = \Pi_{id, class}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = \rho_{R3}(R_1)$

$R_4 = R_1 \times R_2 \times R_3$

$R_5 = \sigma_{(R1.id = R2.id \text{ and } R1.id = R3.id \text{ and } R1.class != R2.class \text{ and } R1.class != R3.class \text{ and } R2.class != R3.class)}(R_4)$

16. Finding "exactly" using relational algebra.

Using the schemas from 12, find the id of the students taking exactly 2 courses.

Soln:

$R_1 = \Pi_{id, Class}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = \rho_{R3}(R_1)$

$R_4 = R_1 \bowtie (R1.id = R2.id \text{ and } R1.class != R2.class) R2$

$R_5 = R_1 \times R_2 \times R_3$

$R_6 = \sigma (R1.id = R2.id \text{ and } R1.id = R3.id \text{ and } R1.class != R2.class \text{ and } R1.class != R3.class \text{ and } R2.class != R3.class) (R5)$

$R_7 = \Pi_{R1.id}(R4) - \Pi_{R1.id}(R6)$

To find exactly n of something, find at least n and at least n+1 and subtract at least n from at least n+1. In this example, to find the ids of the students taking exactly 2 courses, we found the id of the students taking at least 2 courses and subtracted it from the id of the students taking at least 3 courses.

Introduction to SQL:

- So far, we have defined database schemas and queries mathematically.
  SQL (Structured Query Language) is a formal language for doing so with a DBMS.
- There are 2 parts to SQL:
  1. **DDL (Data Definition Language):** Used for defining schemas.
  2. **DML (Data Manipulation Language):** Used for writing queries and modifying the database.

  I.e. Whenever we are defining schemas, it's called DDL and whenever we are writing queries, it's called DML.
- SQL is a very high-level language.
- It provides **physical data independence** meaning that details of how the data is stored can change with no impact on your queries.
- You can focus on readability and because the DBMS optimizes your query, you get efficiency.
- SQL keywords/commands are not case sensitive.
  I.e. select = SELECT
  One convention is to use uppercase for keywords.
- Identifiers are not case-sensitive either.
  One convention is to use lowercase for attributes, and a leading capital letter followed by lowercase for relations.
- Literal strings are case-sensitive, and require single quotes.
- Whitespace (other than inside quotes) is ignored.
- E.g. Consider the query **SELECT surName FROM Student WHERE campus = 'StG';**
  SELECT, FROM and WHERE, which are all keywords, are all in capital letters.
  surName, and campus, which are identifiers for attributes, are lowercase.
  Student, which is an identifier for a relation, starts with an uppercase letter but everything else is lowercase.
  'StG', which is a literal string, is case sensitive and must be surrounded by single quotes.
- There are SQL statements, clauses, operators and functions. I will list the SQL statements in alphabetical order, followed by the SQL clauses in alphabetical order, then SQL operators in alphabetical order, and finally, SQL functions in alphabetical order.
  You need to end each of your SQL queries with a semicolon.
- **Note:** Anything you can write using relational algebra you can write in SQL, but not everything you can write in SQL can be written in relational algebra.

SQL Statements in Alphabetical Order:

Alter Table:

- The ALTER TABLE statement is used to add, delete (drop), or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.
- To add a column in a table, use the following syntax:
  ALTER TABLE table_name ADD column_name datatype;
- To delete a column in a table, use the following syntax:
  ALTER TABLE table_name DROP COLUMN column_name;
- To change the data type of a column in a table, use the following syntax:
  ALTER TABLE table_name ALTER COLUMN column_name datatype;

AS:

- The as statement creates an alias for a table or a column.
  SQL aliases are used to give a table, or a column in a table, a temporary name.
- Aliases are often used to make column names more readable.
- An alias only exists for the duration of the query.
- **Syntax for column: SELECT column_name AS alias_name FROM table_name;**
- E.g.
  Consider the table below:

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

  If I run the query **select FirstName as FName from Students;**, I get

	FName
1	Rick
2	ABC
3	Rick
4	Rick
5	ABC

- **Syntax for table: SELECT column_name(s) FROM table_name AS alias_name;**
- **Note:** There's another way we can rename tables, shown below.
  E.g. **SELECT e.name, d.name FROM employee e, department d WHERE d.name = 'marketing' AND e.name = 'Horton';**

Create Table:

- The create table statement creates a table with the specified table name and column name(s).
- **Syntax: CREATE TABLE table_name(column1 datatype, column2 datatype, column3 datatype, ..., column(n) datatype);**
- E.g.
  The query **create table Students(FirstName Text, LastName Text, StudentNumber INTEGER);**
  Creates this table

Table: Students

FirstName	LastName	StudentNumber
Filter	Filter	Filter

- The column parameters specify the names of the columns of the table.
- The datatype parameter specifies the type of data the column can hold (e.g. text, integer, date, etc).

Cross Join:

- Same as cartesian join.
- **Syntax: SELECT columns FROM Table1 CROSS JOIN Table2;**
- **Note:** You can do **SELECT columns FROM Table1,Table2;** to get the same result. When you have a comma between tables names, you are getting the cartesian join of the tables.
  I.e.

Delete:

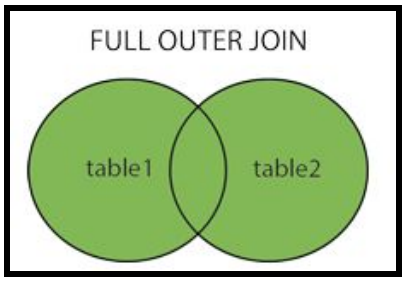
- The DELETE statement is used to delete existing records in a table.
- **Syntax: DELETE FROM table_name WHERE condition;**
- **Note:** Be careful when deleting records in a table. The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted.

Drop Table:

- The DROP TABLE statement is used to drop (delete) an existing table in a database.
- **Syntax: DROP TABLE table_name;**

Full Join:

- Also known as FULL OUTER JOIN.
- The FULL JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.
  I.e.



- **Note:** FULL OUTER JOIN can potentially return very large result-sets.
- **Note:** FULL OUTER JOIN and FULL JOIN are the same.
- **Syntax: SELECT column_name(s) FROM table1 FULL OUTER JOIN table2 ON table1.column_name = table2.column_name WHERE condition;**

Group By:

- The GROUP BY statement groups rows that have the same values into summary rows.
- The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
- **Syntax: SELECT column_name(s) FROM table_name WHERE condition GROUP BY column_name(s)**
- E.g. Consider the table below

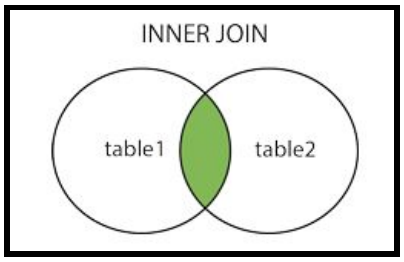
	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select AVG(Mark) from Marks group by StudentNumber;**, I get

	AVG(Mark)
1	82.3333333333333
2	75.5
3	71.5
4	82.0
5	52.0

Inner Join:

- The INNER JOIN keyword selects records that have matching values in both tables. I.e.



- **Syntax: SELECT column_name(s) FROM table1 INNER JOIN table2 ON table1.column_name = table2.column_name;**
- E.g.

Consider the tables

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

Table: Marks

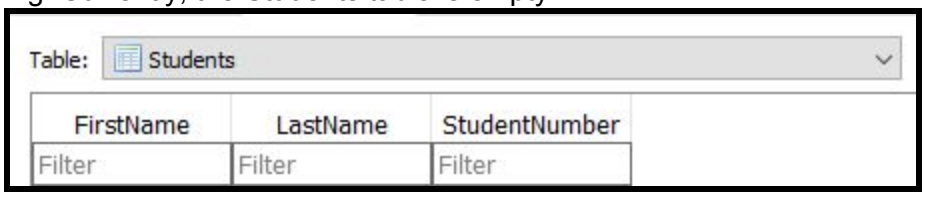
	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select * from Students inner join Marks on Students.StudentNumber = Marks.StudentNumber;**, I get

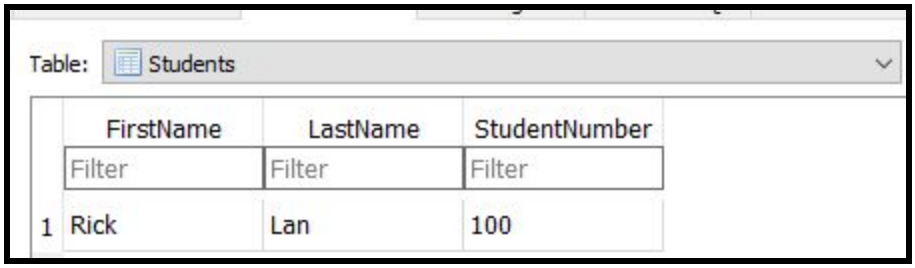
	FirstName	LastName	StudentNumber	StudentNumber	Mark
1	Rick	Lan	99	99	95
2	ABC	DEF	100	100	74
3	Rick	DEF	101	101	88
4	Rick	XYZ	102	102	82
5	ABC	XYZ	103	103	52

Insert Into:

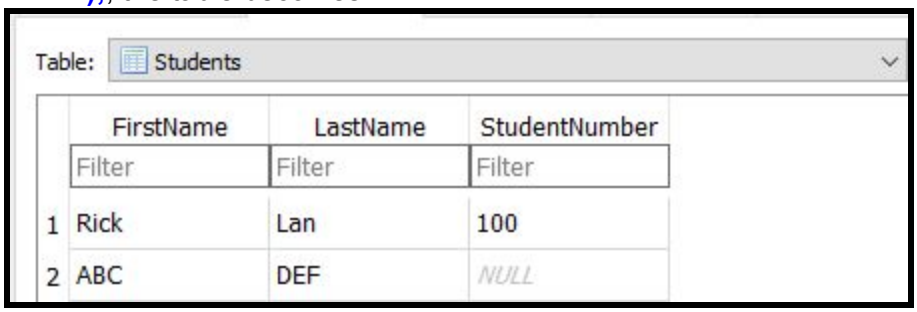
- The INSERT INTO statement is used to insert new records in a table.
- **Syntax #1: INSERT INTO table_name (column1, column2, column3, …, column(n)) VALUES (value1, value2, value3, …, value(n));**
- **Syntax #2: INSERT INTO table_name VALUES (value1, value2, value3, …, value(n));**
- **Syntax #3: INSERT INTO table_name (subquery);**
- The first way specifies both the column names and the values to be inserted.
- You use the second way if you are adding values for all the columns of the table. Here, you do not need to specify the column names. However, make sure the order of the values is in the same order as the columns in the table.
- Sometimes we want to insert tuples, but we don't have values for all attributes. If we name the attributes we are providing values for, the system will use NULL or a default for the rest.
- E.g. Currently, the Students table is empty.



However, if I run the query **insert into Students values ("Rick", "Lan", 100);**, then the table becomes

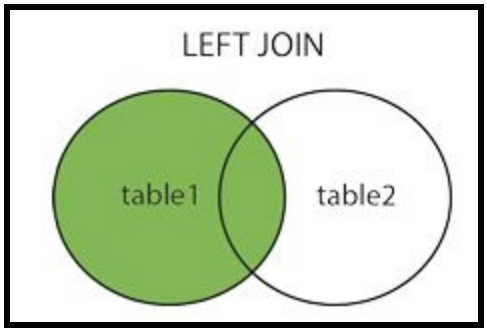


- E.g. If I run the query **insert into Students(FirstName, LastName) values ("ABC", "DEF");**, the table becomes



Left Join:

- Also known as LEFT OUTER JOIN.
- The LEFT JOIN statement returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.
  I.e.



- **Syntax: SELECT column_name(s) FROM table1 LEFT JOIN table2 ON table1.column_name = table2.column_name;**
- E.g.
  Consider the tables below:

Table: Marks

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	104	52

Table: Students

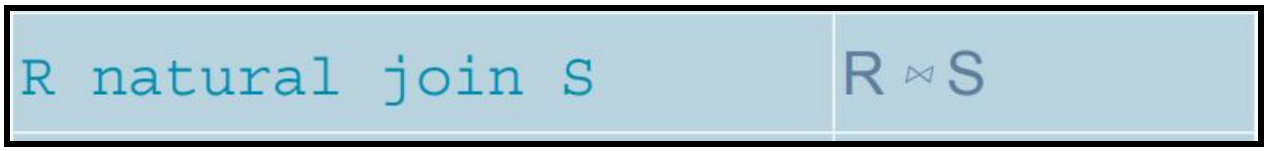
	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I do the query **select * from Students left join Marks on Students.StudentNumber = Marks.StudentNumber;**, I get

	FirstName	LastName	StudentNumber	StudentNumber	Mark
1	Rick	Lan	99	99	95
2	ABC	DEF	100	100	74
3	Rick	DEF	101	101	88
4	Rick	XYZ	102	102	82
5	ABC	XYZ	103	*NULL*	*NULL*

Natural Join:

- **Syntax:** **SELECT columns FROM Table1 NATURAL JOIN Table2;**
  I.e.



- E.g. Consider the tables below

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **SELECT * FROM Students NATURAL JOIN Marks;**, I get

	FirstName	LastName	StudentNumber	Mark
1	Rick	Lan	99	65
2	Rick	Lan	99	87
3	Rick	Lan	99	95
4	ABC	DEF	100	74
5	ABC	DEF	100	77
6	Rick	DEF	101	55
7	Rick	DEF	101	88
8	Rick	XYZ	102	82

- In practice, natural joins are brittle. A working query can be broken by adding a column to a schema. Furthermore, having implicit comparisons impairs readability. The best practise is not to use natural joins.
- **Note:** We can also do **SELECT columns FROM Table1 NATURAL LEFT|RIGHT FULL JOIN Table2;**
  E.g. If I do the query **select * from Students natural left join Marks;**, I get

	FirstName	LastName	StudentNumber	Mark
1	Rick	Lan	99	65
2	Rick	Lan	99	87
3	Rick	Lan	99	95
4	ABC	DEF	100	74
5	ABC	DEF	100	77
6	Rick	DEF	101	55
7	Rick	DEF	101	88
8	Rick	XYZ	102	82
9	ABC	XYZ	103	*NULL*

Order By:

- The ORDER BY statement is used to sort the result-set in ascending or descending order.
- The ORDER BY statement sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.
- The ordering is the last thing done before the SELECT, so all attributes are still available.
- **Syntax: SELECT column1, column2, …, column(n) FROM table_name ORDER BY column1, column2, ... ASC|DESC;**
- The attribute list can include expressions: e.g. **ORDER BY sales+rentals**.
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	99
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from students order by FirstName;**, I get

	FirstName	LastName	StudentNumber
1	ABC	DEF	99
2	ABC	XYZ	103
3	Rick	Lan	100
4	Rick	DEF	101
5	Rick	XYZ	102

If I run the query **select * from students order by FirstName, LastName;**, I get

	FirstName	LastName	StudentNumber
1	ABC	DEF	99
2	ABC	XYZ	103
3	Rick	DEF	101
4	Rick	Lan	100
5	Rick	XYZ	102

Note: Because I didn't specify ascending or descending in the query above, the default is ascending.

If I run the query **select * from students order by FirstName DESC;**, I get

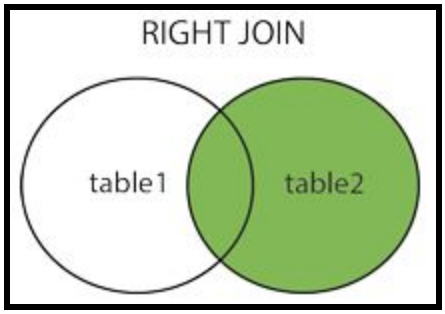
	FirstName	LastName	StudentNumber
1	Rick	Lan	100
2	Rick	DEF	101
3	Rick	XYZ	102
4	ABC	DEF	99
5	ABC	XYZ	103

Note: If you order by more than 1 column, you can order some columns in ascending order and others in descending order.
If I run the query **select * from students order by FirstName DESC, LastName ASC;**, I get

	FirstName	LastName	StudentNumber
1	Rick	DEF	101
2	Rick	Lan	100
3	Rick	XYZ	102
4	ABC	DEF	99
5	ABC	XYZ	103

Right Join:

- Also known as RIGHT OUTER JOIN.
- The RIGHT JOIN statement returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.
  I.e.



- **Syntax: SELECT column_name(s) FROM table1 RIGHT JOIN table2 ON table1.column_name = table2.column_name;**

Select:

- The select statement gets the specified columns from a table.
- **Syntax: select (column names) from table**
- **Note:** If you want to get all the columns from a table, you can do **select * from table(s)**
  A * in the SELECT clause means all attributes of this relation.
- E.g. Currently, the Students table looks like this

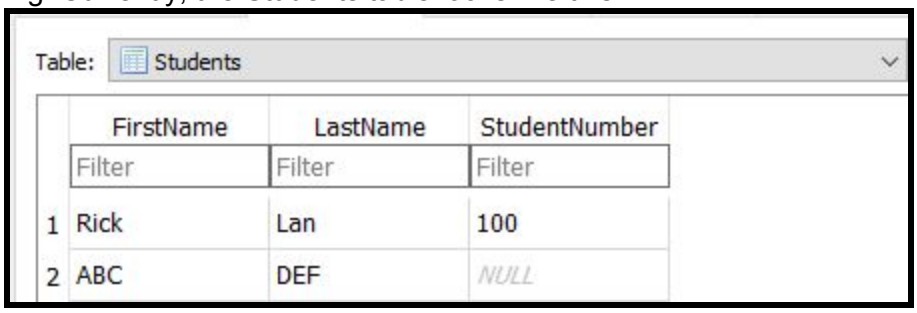
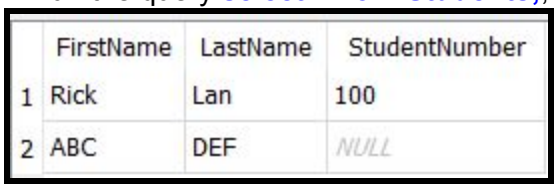


Table: Students

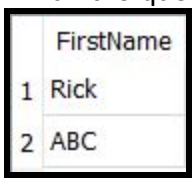
	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	NULL

If I run the query **select * from Students;**, I get



	FirstName	LastName	StudentNumber
1	Rick	Lan	100
2	ABC	DEF	NULL

If I run the query **select FirstName from Students;**, I get



	FirstName
1	Rick
2	ABC

- **Note:** If you have multiple tables after the from keyword, you will get a cartesian product of those tables.
  E.g. The query **SELECT cNum FROM Offering, Took WHERE Offering.id = Took.oid and dept = 'CSC';** is analogous to $\pi_{cNum}(\sigma_{Offering.id=Took.id \wedge dept='csc'}(Offering \times Took))$.
  In SQL, Offering, Took is the same as Offering x Took.
- **Note:** Instead of a simple attribute name, you can use an expression in a SELECT clause.
  E.g.
  SELECT sid, grade+10 AS adjusted FROM Took;
  SELECT dept||cnum FROM course; Note: || means concatenate.

Select Distinct:

- The SELECT DISTINCT statement is used to return only distinct (different) values.
- Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.
- **Syntax: SELECT DISTINCT column1, column2, …, column(n) FROM table_name;**

Self Join:

- A self JOIN is a regular join, but the table is joined with itself.
- **Syntax: SELECT column_name(s) FROM table1 T1, table1 T2 WHERE condition;**
- **Note:** T1 and T2 are different table aliases for the same table.
- E.g. **SELECT e1.name, e2.name FROM employee e1, employee e2 WHERE e1.salary < e2.salary;**

Theta Join:

- **Syntax: SELECT columns FROM Table1 JOIN Table2 ON condition;**
  I.e.

- E.g. Consider the tables below

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **SELECT FirstName, LastName, avg(Mark) FROM Students JOIN Marks on Students.StudentNumber = Marks.StudentNumber GROUP BY Marks.StudentNumber HAVING avg(Mark) > 80;**, I get:

	FirstName	LastName	avg(Mark)
1	Rick	Lan	82.3333333333333
2	Rick	XYZ	82.0

Update:

- The UPDATE statement is used to modify the existing records in a table.
- **Syntax: UPDATE table_name SET column1 = value1, column2 = value2, …, column(n) = value(n) where condition;**
- **Note:** Be careful when updating records in a table. The WHERE clause, while optional, specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated.
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	99
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **update Students set StudentNumber = 99 where FirstName = "Rick" and LastName = "Lan";**, the table becomes

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	99
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **update Students set StudentNumber = 100 where FirstName = "ABC" and LastName = "DEF";**, the table becomes

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

View:

- A **view** is a relation defined in terms of stored tables called **base tables** and other views.
- We can access a view like any table.
- There are 2 kinds of view:
    1. **Virtual:** No tuples are stored. The view is just a query for constructing the relation when needed.
    2. **Materialized:** The table is actually constructed and stored. It is expensive to maintain.
- A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.
- Views, which are a type of virtual tables allow users to do the following:
    - Break down a large query.
    - Provide another way of looking at the same data, e.g. for one category of user.
    - Structure data in a way that users or classes of users find natural or intuitive.
    - Restrict access to the data in such a way that a user can see and sometimes modify exactly what they need and no more.
    - Summarize data from various tables which can be used to generate reports.
- **Syntax: CREATE VIEW view_name AS (SELECT QUERY);**
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **CREATE VIEW StudentNames AS SELECT FirstName, LastName FROM Students;**, I get

Table: StudentNames

	FirstName	LastName
	Filter	Filter
1	Rick	Lan
2	ABC	DEF
3	Rick	DEF
4	Rick	XYZ
5	ABC	XYZ

- Generally, it is impossible to modify a virtual view, because it doesn't exist. Furthermore, most systems prohibit most view updates.
- A problem is that each time a base table changes, the materialized view may change and we can not afford to recompute the view with each change. A solution is to do periodic reconstructions of the materialized view, which is otherwise out of date.

SQL Clauses in Alphabetical Order:

Having:

- The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions (avg, count, max, min, sum, etc).
- **Syntax: SELECT column_name(s) FROM table_name WHERE condition GROUP BY column_name(s) HAVING condition**
- **Note:** The having clause is always used with the group by statement.
- E.g. Consider the table below:

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select StudentNumber from Marks group by StudentNumber having AVG(Mark) > 70;**, I get

	StudentNumber
1	99
2	100
3	101
4	102

- Outside subqueries, HAVING may refer to attributes only if they are either:
  - aggregated or
  - an attribute on the GROUP BY list.

Limit:

- The LIMIT clause is used to specify the number of records to return.
- E.g. With Select

  If I run the query **select * from Students limit 1;**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99

Where:

- The WHERE clause is used to extract only those records that fulfill a specified condition. We can build boolean expressions with operators that produce boolean results. Some comparison operators are:
  - <> (Means not equal to)
  - =
  - <
  - >
  - <=
  - >=
- It can be used with select, update, delete, and other statements.
- E.g. With Select
  Consider the table below:

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	*NULL*
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

  If I run the query **select FirstName from Students where FirstName = "Rick";**, I get

	FirstName
1	Rick
2	Rick
3	Rick

- If I run the query **select StudentNumber from Students where FirstName = "Rick";**, I get

	StudentNumber
1	100
2	101
3	102

- The WHERE clause can be combined with **AND**, **OR**, and **NOT** operators.
- The AND operator displays a record if all the conditions separated by AND are true.
- The OR operator displays a record if any of the conditions separated by OR is true.
- The NOT operator displays a record if the condition(s) is NOT true.

- E.g.
  Consider the table below:

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	*NULL*
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

  If I run the query **select LastName from Students where NOT FirstName = "Rick";**, I get

	LastName
1	DEF
2	XYZ

  If I run the query **update Students set StudentNumber = 99 where FirstName="ABC" and LastName="DEF";**, the table becomes

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	99
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

- The WHERE clause can also be used with the **LIKE** operator to search for a specified pattern in a column.
- There are two wildcards often used in conjunction with the LIKE operator:
  1. %: The percent sign represents zero, one, or multiple characters.
  2. _: The underscore represents a single character.
- **Syntax: SELECT column1, column2, …, column(n) FROM table_name WHERE column(N) LIKE pattern;**

- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from Students where FirstName like "A%";**, I get

	FirstName	LastName	StudentNumber
1	ABC	DEF	100
2	ABC	XYZ	103

If I run the query **select * from Students where FirstName like "R_CK";**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99
2	Rick	DEF	101
3	Rick	XYZ	102

- The WHERE clause can be used with the **IN** operator to specify multiple values. The IN operator is a shorthand for multiple OR conditions.
- **Syntax #1: SELECT column_name(s) FROM table_name WHERE column_name IN (value1, value2, ...);**
- **Syntax #2: SELECT column_name(s) FROM table_name WHERE column_name IN (SELECT STATEMENT);**
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from Students where StudentNumber in (99, 100, 101);**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101

- The WHERE clause can be used with the BETWEEN operator to select values within a given range. The values can be numbers, text, or dates.
  The BETWEEN operator is inclusive: begin and end values are included.
- **Syntax: SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2;**
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from Students where StudentNumber between 99 and 101;**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101

SQL Operators in Alphabetical Order:

All:

- The ALL operator is used with a WHERE or HAVING clause.
- The ALL operator returns true if all of the subquery values meet the condition.
- **Syntax: SELECT column_name(s) FROM table_name WHERE column_name operator ALL (SELECT column_name FROM table_name WHERE condition);**
- **Note:** The operator must be a standard comparison operator (=, <>, >, >=, <, or <=).

Any:

- The ANY operator is used with a WHERE or HAVING clause.
- The ANY operator returns true if any of the subquery values meet the condition.
- **Syntax: SELECT column_name(s) FROM table_name WHERE column_name operator ANY (SELECT column_name FROM table_name WHERE condition);**
- **Note:** The operator must be a standard comparison operator (=, <>, >, >=, <, or <=).
- The ANY operator is also called SOME.
  I.e. ANY is equivalent to SOME.

Except:

- The SQL EXCEPT operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in the second SELECT statement.
- Each SELECT statement within EXCEPT must have the same number of columns.
- The columns must also have similar data types.
- The columns in each SELECT statement must also be in the same order.
- **Syntax: (select column1, column2, …, column(n) from table1) except (select column1, column2, …, column(n) from table1);**
- E.g. Consider the tables below

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select StudentNumber from Students EXCEPT select StudentNumber from Marks;**, I get

	StudentNumber
1	103

Exists:

- The EXISTS operator is used to test for the existence of any record in a subquery.
- The EXISTS operator returns true if the subquery returns one or more records. The subquery is a SELECT statement. If the subquery returns at least one record in its result set, the EXISTS clause will evaluate to true and the EXISTS condition will be met. If the subquery does not return any records, the EXISTS clause will evaluate to false and the EXISTS condition will not be met.
- **Syntax: SELECT column_name(s) FROM table_name WHERE EXISTS (SELECT column_name FROM table_name WHERE condition);**

Intersect:

- The SQL INTERSECT operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.
- Each SELECT statement within INTERSECT must have the same number of columns.
- The columns must also have similar data types.
- The columns in each SELECT statement must also be in the same order.
- **Syntax: SELECT column_name(s) FROM table1 intersect SELECT column_name(s) FROM table2;**
- E.g.
  Consider the tables below:

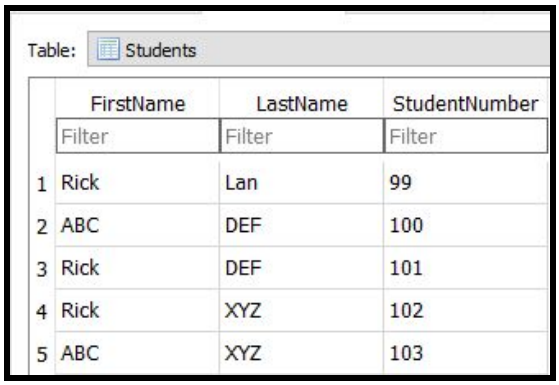


Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

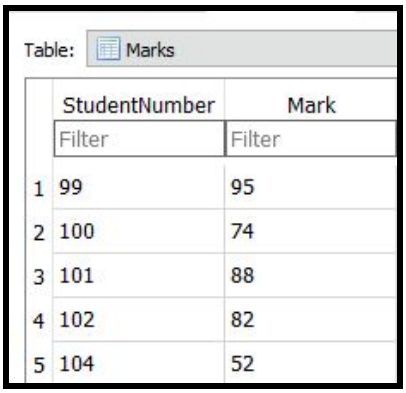
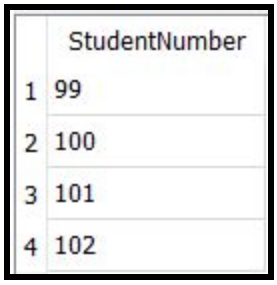


Table: Marks

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	104	52

If I run the query **select StudentNumber from Students intersect SELECT StudentNumber from Marks;**, I get



	StudentNumber
1	99
2	100
3	101
4	102

Union:

- The UNION operator is used to combine the result-set of two or more SELECT statements.
- Each SELECT statement within UNION must have the same number of columns.
- The columns must also have similar data types.
- The columns in each SELECT statement must also be in the same order.
- **Syntax: SELECT column_name(s) FROM table1 UNION SELECT column_name(s) FROM table2;**
- E.g.
  Consider the tables below:

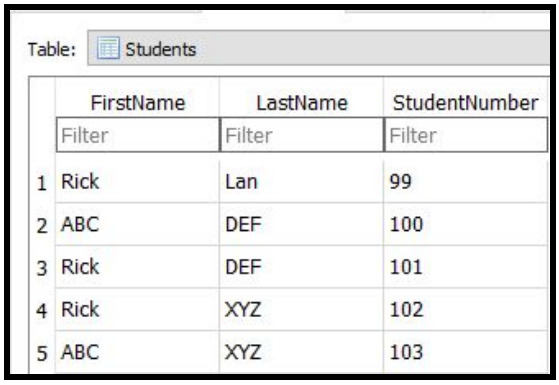


Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

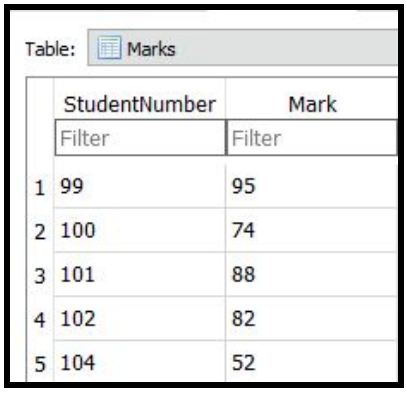
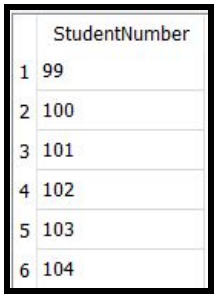


Table: Marks

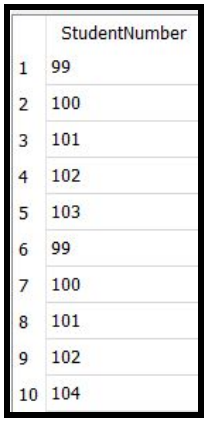
	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	104	52

If I run the query **select StudentNumber from Students union SELECT StudentNumber from Marks;**, I get



	StudentNumber
1	99
2	100
3	101
4	102
5	103
6	104

- The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.
  Syntax: SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;
- E.g. Using the 2 tables from above, if I run the query **select StudentNumber from Students union all SELECT StudentNumber from Marks;**, I get



	StudentNumber
1	99
2	100
3	101
4	102
5	103
6	99
7	100
8	101
9	102
10	104

SQL Functions in Alphabetical Order:

Note: These functions are called **aggregate functions**.

If any aggregation is used, then each element of the SELECT list must be either be:

1. aggregated or
2. an attribute on the GROUP BY list.

Otherwise, it doesn't even make sense to include the attribute.

AVG:

- The AVG() function returns the average value of a numeric column.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select avg(mark) from Marks;**, I get

	avg(mark)
1	78.2

COUNT:

- The COUNT() function returns the number of rows that matches a specified criterion.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select count(mark) from Marks;**, I get

	count(mark)
1	5

- You can do **COUNT(*)** to get the number of tuples.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select count(*) from Marks;**, I get

	count(*)
1	9

MAX:

- The MAX() function returns the largest value of the selected column.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select max(mark) from Marks;**, I get

	max(mark)
1	95

MIN:

- The MIN() function returns the smallest value of the selected column.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query select min(mark) from Marks;, I get

	min(mark)
1	52

SUM:

- The SUM() function returns the total sum of a numeric column.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I do the query **select sum(mark) from Marks;**, I get

	sum(mark)
1	391

Note: You can use the keyword **DISTINCT** with any of these functions to prevent counting duplicates.
E.g. **COUNT (DISTINCT column)** or **SUM(DISTINCT column)**, etc.
DISTINCT does not affect MIN or MAX.
E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select count(StudentNumber) from Marks;**, I get

	count(StudentNumber)
1	9

But, if I run the query **select count(DISTINCT StudentNumber) from Marks;**, I get

	count(DISTINCT StudentNumber)
1	5

Order of execution of a SQL query:

Query order	Execution order
SELECT	FROM
FROM	WHERE
WHERE	GROUP BY
GROUP BY	HAVING
HAVING	SELECT
ORDER BY	ORDER BY

Set Operations:

- Tables can have duplicates in SQL.
- A table can have duplicate tuples, unless this would violate an integrity constraint.
- SELECT-FROM-WHERE statements leave duplicates in unless you say not to. This is because:
  - Getting rid of duplicates is expensive.
  - We may want the duplicates because they tell us how many times something occurred.
- SQL treats tables as **bags/multisets** rather than sets.
- Bags are just like sets, but duplicates are allowed.
  E.g.
  {6, 2, 7, 1, 9} is a set (and a bag) while {6, 2, 2, 7, 1, 9, 9} is not a set but is a bag.
- Like with sets, order doesn't matter with bags.
- **Note:** In SQL, Union, Intersect and Except use set semantics by default. This means that duplicates are eliminated from the result.
- Consider Union, denoted as U for this example, Intersect, denoted as ∩ for this example, and Except, denoted as - for this example and suppose they use bag semantics instead of set semantics. Furthermore, suppose tuple t occurs m times in relation R and n times in relation S.
  Then, we have:

Operation	Number of occurrences of t in result
R ∩ S	min(m, n)
R U S	m + n
R - S	max(m-n, 0)

E.g. of union, intersect and except using bag semantics:

1. {1, 1, 1, 3, 7, 7, 8} ∪ {1, 5, 7, 7, 8, 8} = {1, 1, 1, 1, 3, 5, 7, 7, 7, 7, 8, 8, 8}
2. {1, 1, 1, 3, 7, 7, 8} ∩ {1, 5, 7, 7, 8, 8} = {1, 7, 7, 8}
3. {1, 1, 1, 3, 7, 7, 8} − {1, 5, 7, 7, 8, 8} = {1, 1, 3}

- **Note:** We can force the result of a Select-From-Where query to be a set by using **SELECT DISTINCT**.
- **Note:** We can force the result of a set operation to be a bag by using the keyword **ALL**.

Dangling tuples:

- With joins that require some attributes to match, tuples lacking a match are left out of the results. We say that they are **dangling**.
- An **outer join** preserves dangling tuples by padding them with NULL in the other relation.

  There are 3 types of outer joins.
  1. LEFT OUTER JOIN
  2. RIGHT OUTER JOIN
  3. FULL OUTER JOIN
-
- A join that doesn't pad with NULL is called an **inner join**.
- There are keywords INNER and OUTER, but you never need to use them.
- You get an outer join iff you use the keywords LEFT, RIGHT, or FULL.
- If you don't use the keywords LEFT, RIGHT, or FULL you get an inner join.
- Here is a chart to show comparisons:

	Theta Join	**Natural Join**
Inner Join	A JOIN B ON C	A NATURAL JOIN B
Outer Join	A {LEFT|RIGHT|FULL} JOIN B ON C	A NATURAL {LEFT|RIGHT|FULL} JOIN B

- E.g. Consider the tables R and S below:

R

A	B
1	2
4	5

S

B	C
2	3
6	7

This is **R NATURAL JOIN S**:

A	B	C
1	2	3

This is **R NATURAL FULL JOIN S**:

A	B	C
1	2	3
4	5	NULL
NULL	6	7

This is **R NATURAL LEFT JOIN S**:

A	B	C
1	2	3
4	5	NULL

This is **R NATURAL RIGHT JOIN S**:

A	B	C
1	2	3
NULL	6	7

Null Values:

- There are 2 common scenarios for null values:
    1. Missing value
       E.g. We know a student has some email address, but we don't know what it is.
    2. Inapplicable attribute
       E.g. The value of the attribute spouse is inapplicable for an unmarried person.
- One possibility for representing missing information is to use a special value as a placeholder. However, a better solution is to use a value not in any domain. We call this a **null value**.

- Tuples in SQL relations can have NULL as a value for one or more components.
- You can compare an attribute value to NULL with the following clauses:
    1. **IS NULL**
    2. **IS NOT NULL**

  E.g. **SELECT * FROM Course WHERE breadth IS NULL;**
- Because of NULL, we need three truth-values:
    1. If one or both operands to a comparison is NULL, the comparison always evaluates to UNKNOWN.
    2. True
    3. False
- We need to know how the three truth-values combine with AND, OR and NOT.
- To do so, we can think in terms of numbers:
    - TRUE = 1
    - FALSE = 0
    - UNKNOWN = 0.5
- AND is min, OR is max, NOT x is (1-x).
- **Note:** A tuple is in a query result iff the WHERE clause is TRUE. UNKNOWN is not good enough.

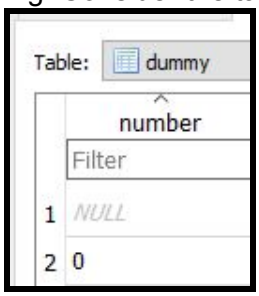
  E.g. Consider the table below:

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	104	52
6	99	87
7	100	77
8	99	65
9	101	55
10	101	*NULL*

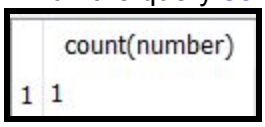
If I run the query **select * from Marks where Mark > 70;**, I get

	StudentNumber	Mark
1	99	95
2	100	74
3	101	88
4	102	82
5	99	87
6	100	77

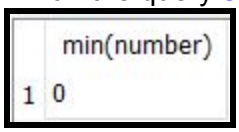
- **Note:** The aggregate functions ignores NULL.
- NULL never contributes to a sum, average, or count, and NULL can never be the minimum or maximum of a column unless every value is NULL.
- If there are no non-NULL values in a column, then the result of the aggregation is NULL. An exception is that COUNT of an empty set is 0.
- E.g. Consider the table below



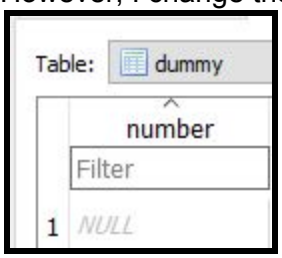
If I run the query **select count(number) from dummy;**, I get



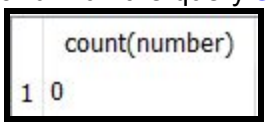
If I run the query **select min(number) from dummy;**, I get



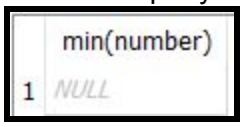
However, I change the table such that all the rows are null,



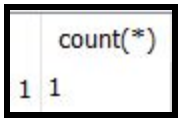
and I run the query **select count(number) from dummy;**, I get



If I run the query **select min(number) from dummy;**, I get



If I run the query **select count(*) from dummy;**, I get



- Here's a table of the summary of aggregate functions on NULL

	some nulls in A	All nulls in A
min(A)	ignore the nulls	null
max(A)		
sum(A)		
avg(A)		
count(A)		0
count(*)	all tuples count	

- Other corner cases to think about:
  1. SELECT DISTINCT: Are 2 NULL values equal?
     (For the most part, for select distinct, 2 null values are equal.)
  2. NATURAL JOIN: Are 2 NULL values equal?
     (For the most part, for natural join, 2 null values are not equal.)
  3. SET OPERATIONS: Are 2 NULL values equal?
     (For the most part, for set operations, 2 null values are equal.)
  4. UNIQUE Constraint: Do 2 NULL values violate it?

  However, this behaviour may vary across DBMSs.
  Different DBMSs have different implementations.

Subqueries:

Can go:

- In a FROM clause:
  - In place of a relation name in the FROM clause, we can use a subquery.
  - The subquery must be parenthesized.
  - We must name the result, so you can refer to it in the outer query.
  - E.g.

    SELECT sid, dept||cnum as course, grade FROM Took,
    (SELECT * FROM Offering WHERE instructor='Horton') Hoffering
    WHERE Took.oid = Hoffering.oid;
- In a WHERE clause:
  - If a subquery is guaranteed to produce exactly one tuple, then the subquery can be used as a value.
  - The simplest situation is that one tuple has only one component.
  - E.g.

    SELECT sid, surname FROM Student WHERE cgpa >
    (SELECT cgpa FROM Student WHERE sid = 99999);
  - When a subquery can return multiple values, we can make comparisons using a quantifier by using the ALL, ANY/SOME, IN, and EXISTS operators.
- As operands to UNION, INTERSECT or EXCEPT.

Scope:

- Queries are evaluated from the inside out.
- If a name might refer to more than one thing, use the most closely nested one.
- If a subquery refers only to names defined inside it, it can be evaluated once and used repeatedly in the outer query.

- If it refers to any name defined outside of itself, it must be evaluated once for each tuple in the outer query. These are called **correlated subqueries**.
  Think of this as a nested loop.
- Renaming can make scope explicit.
  E.g.
  SELECT instructor FROM Offering Off1 WHERE NOT EXISTS
  (SELECT * FROM Offering Off2 WHERE Off2.oid <> Off1.oid AND Off2.instructor = Off1.instructor);

Q1. Given the schema below, answer the following questions.

Schema

Student(<u>sID</u>, surName, firstName, campus, email, cgpa)
Course(<u>dept, cNum</u>, name, breadth)
Offering(<u>oID</u>, dept, cNum, term, instructor)
Took(<u>sID, oID</u>, grade)

Offering[dept, cNum] ⊆ Course[dept, cNum]
Took[sID] ⊆ Student[sID]
Took[oID] ⊆ Offering[oID]

1. Answer each of the following questions with an arithmetic expression.
   Suppose a row occurs n times in table R and m times in table S.
   a. Using bag semantics, how many times will it occur in table R ∪ S?

      Soln:
      m+n
      Recall that bag semantics allows for duplicate entries/tuples.
      In bag semantics, {1, 1, 1, 3, 7, 7, 8} ∪ {1, 5, 7, 7, 8, 8} = {1, 1, 1, 1, 3, 5, 7, 7, 7, 7, 8, 8, 8}.
      Hence, if a row occurs n times in table R and m times in table S, when we take the union of R and S, we get all those rows.
      Hence, it will occur m+n times.

   b. Using bag semantics, how many times will it occur in table R ∩ S?

      Soln:
      min(m, n)
      In bag semantics, {1, 1, 1, 3, 7, 7, 8} ∩ {1, 5, 7, 7, 8, 8} = {1, 7, 7, 8}.
      This shows that for intersection, we always take the least amount of occurrences in either set.
      Hence, it will occur min(m, n) times.

   c. Using bag semantics, how many times will it occur in table R − S?

      Soln:
      max(m-n, 0)
      In bag semantics, {1, 1, 1, 3, 7, 7, 8} − {1, 5, 7, 7, 8, 8} = {1, 1, 3}.
      Hence, if the row exists in R, we have m-n occurrences.
      If the row does not exist in R, we have 0 occurrences.
      Hence, it will occur max(m-n, 0) times.

2. Use a set operation to find all terms when Jepson and Suzuki were both teaching.
   Include every occurrence of a term from the result of both operands.

   Soln:

```
Answer(term) :=
        (SELECT term FROM Offering WHERE instructor = "Jepson")
                INTERSECT ALL
        (SELECT term FROM Offering WHERE instructor = "Suzuki");
```

3. Find the sID of students who have earned a grade of 85 or more in some course, or who have passed a course taught by Atwood. Ensure that no sID occurs twice in the result.

   Soln:

```
(SELECT sID FROM Took WHERE grade >= 85)
        UNION
(SELECT sID FROM Took NATURAL JOIN Offering WHERE grade>=50
AND instructor = “Atwood”);
```

4. Find all terms when csc369 was not offered.

   Soln:

```
(SELECT term FROM Offering)
        EXCEPT
(SELECT term FROM Offering WHERE dept=“CSC” AND cNUM=369);
```

5. Make a table with two columns: oID and results. In the results column, report either “high” (if that offering had an average grade of 80 or higher), or “low” (if that offering had an average under 60). Offerings with an average in between will not be included.

   Soln:

```
(SELECT oID, “high” AS results FROM Took GROUP BY oID HAVING
avg(grade) >= 80)
        UNION
(SELECT oID, “low” AS results FROM Took GROUP BY oID HAVING
avg(grade) < 60);
```

Q2. Given the schema below, answer the following questions.

Schema

Student(<u>sID</u>, surName, firstName, campus, email, cgpa)
Course(<u>dept, cNum</u>, name, breadth)
Offering(<u>oID</u>, dept, cNum, term, instructor)
Took(<u>sID, oID</u>, grade)

Offering[dept, cNum] ⊆ Course[dept, cNum]
Took[sID] ⊆ Student[sID]
Took[oID] ⊆ Offering[oID]

1. Write a query to find the average grade, minimum grade, and maximum grade for each offering.

   Soln:

   Max grade:

```
SELECT max(grade) FROM Took GROUP BY oID;
```

   Min grade:

```
SELECT min(grade) FROM Took GROUP BY oID;
```

   Average grade:

```
SELECT avg(grade) FROM Took GROUP BY oID;
```

2. Which of these queries is legal?

```
SELECT surname, sid
FROM Student, Took
WHERE Student.sid = Took.sid
GROUP BY sid;
```

```
SELECT surname, Student.sid
FROM Student, Took
WHERE Student.sid = Took.sid
GROUP BY campus;
```

```
SELECT instructor, max(grade),
    count(Took.oid)
FROM Took, Offering
WHERE Took.oid = Offering.oid
GROUP BY instructor;
```

```
SELECT Course.dept, Course.cnum,
    count(oid), count(instructor)
FROM Course, Offering
WHERE Course.dept = Offering.dept and
    Course.cnum = Offering.cnum
GROUP BY Course.dept, Course.cnum
ORDER BY count(oid);
```

Soln:
SELECT instructor, max(grade), count(Took.oid)
FROM Took, Offering
WHERE Took.oid = Offering.oid
GROUP BY instructor;
and
SELECT Course.dept, Course.cnum, count(oid), count(instructor)
FROM Course, Offering
WHERE Course.dept = Offering.dept and Course.cnum = Offering.cnum
GROUP BY Course.dept, Course.cnum
ORDER BY count(oid);
are legal.
Note: For the other 2 queries, you cannot get the column(s) that you are not grouping by or using an aggregate function.

For
SELECT surname, sid
FROM Student, Took
WHERE Student.sid = Took.sid
GROUP BY sid;
you cannot select surname unless you use an aggregate function with it or you also group by it.

For
SELECT surname, Student.sid
FROM Student, Took
WHERE Student.sid = Took.sid
GROUP BY campus;
you cannot get surname or Student.sid unless you use an aggregate function with it or you also group by it.

3. Find the sid and minimum grade of each student with an average over 80.

   Soln:
   SELECT sID, min(grade) FROM Took GROUP BY sID HAVING avg(grade) > 80;

4. Find the sid, surname, and average grade of each student, but keep the data only for those students who have taken at least 10 courses.

   Soln:
   SELECT sID, surName, avg(grade) FROM Took NATURAL JOIN Offering NATURAL JOIN Student GROUP BY sID, surName HAVING count(oID) >= 10;

5. For each student who has passed at least 10 courses, report their sid and average grade on the courses that they passed.

   Soln:
   // This first query creates a view that contains the sID of all students who have
   // passed at least 10 courses.
   CREATE VIEW students_who_passed_at_least_10_courses(sID) AS
   SELECT sID FROM Took WHERE grade >= 50 GROUP BY sID
   HAVING count(grade) >= 10;

   // This second query creates a view that contains the sID of all students who have
   // passed at least 10 courses as well as the oID of the courses they passed.
   CREATE VIEW courses_passed(sID, oID) AS
   SELECT sID, oID FROM Took NATURAL JOIN
   students_who_passed_at_least_10_courses;

   // This third query gets the sID of all students who have passed at least 10 courses as
   // well as the average grade of the courses they passed.
   SELECT sID, avg(grade) FROM Took NATURAL JOIN courses_passed
   GROUP BY sID;

6. For each student who has passed at least 10 courses, report their sid and average grade on all of their courses.

   Soln:
   // This first query creates a view of all the students who passed at least
   // 10 courses.
   CREATE VIEW students_who_passed_at_least_10_courses(sID) AS
   SELECT sID FROM Took WHERE grade >= 50 GROUP BY sID
   HAVING count(grade) >= 10;

   // This second query gets the sID and average grade of all students who passed
   // at least 10 courses.
   SELECT sID, avg(grade) FROM Took NATURAL JOIN
   students_who_passed_at_least_10_courses GROUP BY sID;

7. Which of these queries is legal?

```
SELECT dept
FROM Took, Offering
WHERE Took.oID = Offering.oID
GROUP BY dept
HAVING avg(grade) > 75;
```

```
SELECT Took.oID, avg(grade)
FROM Took, Offering
WHERE Took.oID = Offering.oID
GROUP BY Took.oID
HAVING avg(grade) > 75;
```

```
SELECT Took.oID, dept, cNum, avg(grade)
FROM Took, Offering
WHERE Took.oID = Offering.oID
GROUP BY Took.oID
HAVING avg(grade) > 75;
```

```
SELECT oID, avg(grade)
FROM Took
GROUP BY sID
HAVING avg(grade) > 75;
```

Soln:

```
SELECT dept
FROM Took, Offering
WHERE Took.oID = Offering.oID
GROUP BY dept
HAVING avg(grade) > 75;
```

```
SELECT Took.oID, avg(grade)
FROM Took, Offering
WHERE Took.oID = Offering.oID
GROUP BY Took.oID
HAVING avg(grade) > 75;
```

Q3. Given the schema below, answer the following questions.

Schema

Student(sID, surName, firstName, campus, email, cgpa)

Course(dept, cNum, name, breadth)

Offering(oID, dept, cNum, term, instructor)

Took(sID, oID, grade)

Offering[dept, cNum] ⊆ Course[dept, cNum]

Took[sID] ⊆ Student[sID]

Took[oID] ⊆ Offering[oID]

1. Which of these queries is legal?

(a)
```
SELECT count(distinct dept), count(distinct instructor)
FROM Offering
WHERE term >= 20089;
```

(b)
```
SELECT distinct dept, distinct instructor
FROM Offering
WHERE term >= 20089;
```

(c)
```
SELECT distinct dept, instructor
FROM Offering
WHERE term >= 20089;
```

Soln:

a and c are both legal.
b is not legal because the keyword “distinct” is repeated.
Distinct cannot be used twice.

2. Under what conditions could these two queries give different results? If that is not possible, explain why.

```
SELECT surName, campus                    SELECT distinct surName, campus
FROM Student;                             FROM Student;
```

Soln:
If there are multiple rows with the same surName and campus, then the first query will list all of them while the second query only lists one of them.

3. For each student who has taken a course, report their sid and the number of different departments they have taken a course in.

Soln:
SELECT sID, count(DISTINCT dept) FROM Course NATURAL JOIN Took GROUP BY sID;

4. Suppose we have two tables with content as follows:

```
SELECT *                                  SELECT *
FROM One;                                 FROM Two;

 a  |  b                                     b  |  c
----+-----                                -----+-----
  1 |   2                                   2 |   3
  6 |  12                                 100 | 101
    | 100                                  20 |  21
 20 |                                       2 |   4
(4 rows)                                    2 |   5
                                          (5 rows)
```

a. What query could produce this result?

```
 a |   b  |   c
---+------+-----
 1 |    2 |   3
 1 |    2 |   4
 1 |    2 |   5
   |   20 |  21
   |  100 | 101
(5 rows)
```

Soln:
SELECT * FROM One RIGHT JOIN Two;

b. What query could produce this result?

```
 a  |  b  |  c
----+-----+-----
  1 |   2 |   3
  1 |   2 |   4
  1 |   2 |   5
  6 |  12 |
    | 100 | 101
 20 |     |
(6 rows)
```

Soln:
SELECT * FROM One LEFT JOIN Two;

Q4. Given the schema below, answer the following questions.

Schema

Student(sID, surName, firstName, campus, email, cgpa)
Course(dept, cNum, name, breadth)
Offering(oID, dept, cNum, term, instructor)
Took(sID, oID, grade)

Offering[dept, cNum] ⊆ Course[dept, cNum]
Took[sID] ⊆ Student[sID]
Took[oID] ⊆ Offering[oID]

1. What does this query do? (Recall that the || operator concatenates two strings.)

```
SELECT sid, dept || cnum as course, grade
FROM Took,
        (SELECT *
         FROM Offering
         WHERE instructor = 'Horton') Hoffering
WHERE Took.oid = Hoffering.oid;
```

Soln:
The query gets all the students who have taken some courses with instructor Horton and all the courses taught by instructor Horton.

2. What does this query do?

```
SELECT sid, surname
FROM Student WHERE cgpa >
        (SELECT cgpa
        FROM Student
        WHERE sid = 99999);
```

Soln:
This gets the sid and surname of all students who have a cgpa greater than the cgpa of the student with the sid 99999.

3. What does this query do?

```
SELECT sid, dept || cnum AS course, grade
FROM Took JOIN Offering ON Took.oid = Offering.oid
WHERE
        grade >= 80 AND
        (cnum, dept) IN (
                SELECT cnum, dept
                FROM Took JOIN Offering ON Took.oid = Offering.oid
                                JOIN Student ON Took.sid = Student.sid
                WHERE surname = 'Lakemeyer');
```

Soln:
The inner query gets all the courses taken by students with the surname Lakemeyer. The outer query gets the sids of all students who have gotten a mark of 80 or higher in a course taken by students with the surname Lakemeyer as well as the courses.

4.

a. Suppose we have these relations: R(a, b) and S(b, c). What does this query do?

```
SELECT a
FROM R
WHERE b in (SELECT b FROM S);
```

Soln:
This query gets you all the a's from relation R whose corresponding b value is also in S.

b. Can we express this query without using subqueries?

Soln:
Yes. The query is given below.

```
SELECT a from R NATURAL JOIN S;
```

5. What does this query do?

```
SELECT instructor
FROM Offering Off1
WHERE NOT EXISTS (
        SELECT *
        FROM Offering
        WHERE
                oid <> Off1.oid AND
                instructor = Off1.instructor);
```

Soln:

The inner query gets all offerings that are not Off1.oid but both have the same instructor.
I.e. The inner query is essentially that an instructor has taught multiple courses.
The outer query gets all instructors who did not teach multiple courses.
I.e. The outer query gets all instructors who only taught one course.

Types:

- Table attributes have types.
- When creating a table, you must define the type of each attribute.
  This is analogous to declaring a variable's type in a program.
- Built-in Types:
  - **CHAR(n):** This is a fixed-length string of n characters. It can be padded with blanks if necessary.
  - **VARCHAR(n):** This is a variable-length string of up to n characters.
  - **TEXT:** This is a variable-length string with an unlimited number of characters. While this is not in the SQL standard, PSQL and others support it.
  - **INT:** INTEGER
  - **FLOAT:** REAL
  - **BOOLEAN**
  - **DATE**
  - **TIME**
  - **TIMESTAMP:** This is date plus time.

  E.g.
  - Strings: 'ABC'
    Note: Strings must be surrounded with single quotes.
  - INT: 37
  - FLOAT: 1.49, 37.96e2
  - BOOLEAN: TRUE, FALSE
  - DATE: '2011-09-22'
  - TIME: '15:00:02', '15:00:02.5'
  - TIMESTAMP: 'Jan-12-2011 10:25'
- These are not the only built-in types. There are many more built-in types.

User-defined types:

- Defined in terms of a built-in type.
- You make it more specific by defining constraints and perhaps a default value.
- E.g.
  create domain Grade as int default null check (value>=0 and value <=100);

  The check happens every time a user tries to put in a value of type Grade. It checks that the number is between 0 and 100 inclusive.
- E.g.
  create domain Campus as varchar(4) default 'StG' check (value in ('StG','UTM','UTSC'));
- Constraints on a type are checked every time a value is assigned to an attribute of that type.
- You can use these to create a powerful type system.
- The default value for a type is used when no value has been specified.
- This is useful because you can run a query and insert the resulting tuples into a relation even if the query does not give values for all attributes.
- Table attributes can also have default values.
  The difference between attribute default and type default is that:
  - with **attribute default**, it is only for that one attribute in that one table.
  - with **type default**, it is for every attribute defined to be of that type.

Key Constraints:

- **Primary Key Constraints:**
- A **PRIMARY KEY constraint** uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values and cannot contain NULL values.

- A table can have at most one primary key and this primary key can consist of one or multiple columns.
- Declaring that a set of one or more attributes is the PRIMARY KEY for a relation means:
  - they form a key (they are unique).
  - their values will never be null (you don't need to separately declare that).
- Primary keys are a big hint to the DBMS. They optimize for searches by this set of attributes.
- Every table must have 0 or 1 primary key.
  A table can have no primary key, but in practise, every table should have one. This is because if you have duplicate rows in a table, it can make queries useless and it can take up space, unnecessarily.
  You cannot declare more than one primary key.
- There are 2 ways to declare a primary key:
  1. For a single-attribute key, can be part of the attribute definition.
     E.g.
     ```
     create table Test (
             ID integer primary key,
             name varchar(25)
     );
     ```
  2. Or they can be at the end of the table definition. This is the only way for multi-attribute keys. The brackets are required.
     E.g.
     ```
     create table Test (
             ID integer,
             name varchar(25),
             primary key (ID)
     );
     ```
- **Unique Constraints:**
- The **UNIQUE constraint** ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.
- Declaring that a set of one or more attributes is UNIQUE for a relation means:
  - They form a key (They are unique.)
  - Their values can be null. This is because null ≠ null.
    Note: If they mustn't be null, you need to separately declare that.
- **Note:** You can declare more than one set of attributes to be UNIQUE.
- There are 2 ways to declare uniqueness:
  1. If only one attribute is involved, can be part of the attribute definition.
     E.g.
     ```
     create table Test (
             ID integer unique,
             name varchar(25)
     );
     ```

2. Or they can be at the end of the table definition. This is the only way if multiple attributes are involved. The brackets are required.

```
create table Test (
        ID integer,
        name varchar(25),
        unique (ID)
);
```

- For uniqueness constraints, no two nulls are considered equal.
- E.g. Consider

```
create table Testunique (
        first varchar(25),
        last varchar(25),
        unique(first, last)
);
```

This would prevent two insertions of ('Diane', 'Horton'), but it would allow two insertions of (null, 'Schoeler').
This can't occur with a primary key because primary keys can't be null.

- **Foreign Key Constraints:**
- A **FOREIGN KEY** is a key used to link two tables together.
- A FOREIGN KEY is a column or collection of columns in one table that refers to the PRIMARY KEY or unique columns in another table.
- The table containing the foreign key is called the **child table**, and the table containing the primary key is called the **referenced/parent table**.
- The **FOREIGN KEY constraint** is used to prevent actions that would destroy links between tables.
- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.
- E.g. Consider **foreign key (sID) references Student**
  This means that the attribute sID is a foreign key that references the primary key of table Student.
  Every value for sID in this table must actually occur in the Student table.
- The requirement for foreign keys is that they must be declared either primary key or unique in the "home" table.
  I.e. The attribute(s) the foreign key references to must be either a primary key or unique.
- There are 2 ways to declare foreign keys:
  Suppose we have the table people as defined here:

```
create table People (
        SIN integer primary key,
        name text,
        OHIP text unique
);
```

1. If only one attribute is involved, can be part of the attribute definition.
   E.g.

```
create table Volunteers (
        email text primary key,
        OHIPnum text references People(OHIP)
);
```

2. Or they can be at the end of the table definition. This is the only way if multiple attributes are involved. The brackets are required.

```
create table Volunteers (
        email text primary key,
        OHIPnum text,
        FOREIGN KEY (OHIPnum) references People(OHIP)
);
```

- Suppose there is a foreign-key constraint from relation R to relation S.
  When must the DBMS ensure that:
  - The referenced attributes are PRIMARY KEY or UNIQUE?
  - The values actually exist?

  Also, what could cause a violation?
  You get to define what the DBMS should do. This is called specifying a **reaction policy**.
- **Check Constraints:**
- The **CHECK constraint** is used to limit the value range that can be placed in a column, in a tuple, in a relation or in a user-defined type.
- If you define a CHECK constraint on a single column it allows only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.
- The CHECK constraint can also be used for a user-defined type, as stated before.
  - **Attribute-based check constraints:**
  - Defined with a single attribute and constrains its value in every tuple.
  - Can only refer to that attribute.
  - Can include a subquery.
  - E.g.

```
CREATE TABLE Persons (
        ID int,
        LastName varchar(255),
        FirstName varchar(255),
        Age int CHECK (Age>=18)
);
```

  - E.g.

```
CREATE TABLE Student (
        ID int,
        LastName varchar(255),
        FirstName varchar(255),
        Program varchar(5) CHECK (program in (select post from P))
);
```

    Note: The condition can be anything that could go in a WHERE clause.
  - The condition is checked only when a tuple is inserted into that relation, or its value for that attribute is updated.
  - If a change somewhere else violates the constraint, the DBMS will not notice.
    E.g.
    If a student's program changes to something not in table P, we get an error.
    But if table P drops a program that some student has, there is no error.
    - **Not Null Constraints:**
    - By default, a column can hold NULL values.
    - The NOT NULL constraint enforces a column to NOT accept NULL values.

  - This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.
  - You can declare that an attribute of a table is NOT NULL.
  - E.g.

```
create table Course(
        cNum integer,
        name varchar(40) not null,
        dept Department,
        wr boolean,
        primary key (cNum, dept)
);
```

  - E.g.

```
create table User(
        username varchar(10) not null,
        password varchar(10) not null,
        first_name text not null,
        last_name text not null,
);
```

  - In practise, many attributes should be not null.
  - This is a very specific kind of attribute-based constraint.

- **Tuple-based check constraints:**
- Defined as a separate element of the table schema, so it can refer to any attributes of the table.
- Again, the condition can be anything that could go in a WHERE clause, and can include a subquery.
- E.g.

```
create table Student (
        sID integer,
        age integer,
        year integer,
        college varchar(4) check college in (select name from Colleges),
        check (year = age - 18),
);
```

  Here, both age and years are columns. However, with “check (year = age - 18)”, we cannot, for example, put age = 30 and year = 40 as that would violate the constraint.

- The constraint(s) are checked only when a tuple is inserted into that relation, or updated.
- Again, if a change somewhere else violates the constraint, the DBMS will not notice.
- **How nulls affect check constraints:**
- A check constraint only fails if it evaluates to false.
- It is not picky like a WHERE condition.

- E.g. Suppose we have check (age > 0)

age	Value of condition	CHECK outcome	WHERE outcome
19	TRUE	pass	pass
-5	FALSE	fail	fail
NULL	unknown	pass	fail

- Suppose you created this table:

```
create table Frequencies(
        word varchar(10),
        num integer check (num > 5)
);
```

It would allow you to insert ('hello', null) since null passes the constraint check (num > 5).

If you need to prevent that, use a "not null" constraint.
I.e.

```
create table Frequencies(
        word varchar(10),
        num integer not null check (num > 5)
);
```

- **Naming your constraints:**
- If you name your constraint, you will get more helpful error messages.
- This can be done with any of the types of constraint we've seen.
- To add a name to a constraint, do:
  Add constraint «name» before the check («condition»)
- E.g.

```
create domain Grade as smallint
        default null
        constraint gradeInRange
                check (value>=0 and value <=100));
```

- E.g.

```
create domain Campus as varchar(4)
        not null
        constraint validCampus
                check (value in ('StG', 'UTM', 'UTSC'));
```

- E.g.

```
create table Offering(...
        constraint validCourseReference
        foreign key (cNum, dept) references Course);
```

- The order of constraints doesn't matter, and doesn't dictate the order in which they're checked.
- **Assertions:**
- Check constraints can't express complex constraints across tables.
  Check constraints are good for checking data types or information within 1 table.
- Assertions are schema elements at the top level, so they can express cross-table constraints.
- Syntax: **create assertion (<name>) check (<predicate>);**

- E.g. Suppose we have:
    - Every loan has at least one customer, who has an account with at least $1,000.
    - For each branch, the sum of all loan amounts < the sum of all account balances.

  Both of these require multiple tables. Hence, we need to use assertions instead of checks.
- Assertions are powerful but costly.
- SQL has a fairly powerful syntax for expressing the predicates, including quantification.
- Assertions are costly because:
    1. They have to be checked upon every database update, although a DBMS may be able to limit this.
    2. Each check can be expensive.
- Testing and maintenance are also difficult.
- So assertions must be used with great care.
- **Triggers:**
- Assertions are powerful, but costly.
- Check constraints are less costly, but less powerful.
- Triggers are a compromise between these extremes:
    1. They are cross-table constraints, as powerful as assertions.
    2. But you control the cost by having control over when they are applied.
- A trigger is a database object that is associated with the table, it will be activated when a defined action is executed for the table.The trigger can be executed when we run the following statements:
    1. INSERT
    2. UPDATE
    3. DELETE

  And it can be invoked before or after the event.
  Syntax:

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

  E.g.

```
create trigger t1  before  UPDATE on sailors
for each row
begin
  if new.age>60 then
    set new.age=old.age;
  else
    set new.age=new.age;
  end if;
end;
```

- Differences between assertions and triggers:

ASSERTIONS	TRIGGERS
We can use assertions when we know that the given particular condition is always true.	We can use triggers even if a particular condition may or may not be true.
When the SQL condition is not met then there are chances for an entire table or even Database to get locked up.	Triggers can catch errors if the condition of the query is not true.
Assertions are not linked to specific tables or events. It performs tasks specified or defined by the user.	Triggers help in maintaining the integrity constraints in the database tables, especially when the primary key and foreign key constraint are not defined.
Assertions do not maintain any track of changes made in table.	Triggers maintain track of all changes occurring in the table.
Assertions have small syntax compared to triggers.	They have large syntax to indicate each and every specific of the created trigger.

- **Reaction Policies:**
- Suppose we have 2 relations, R and S, where R = Took and S = Student.
  Can you delete a student from S without making a change to R first? (Answer: No)
  Consider if you could. Say student R1 took CSCA08 and CSCA67. If you delete R1 from S without deleting its corresponding values in R, you'll have CSCA08 and CSCA67 in R but you can't find the student who's taking it.
- Your reaction policy can specify one of these reactions:
  1. **Cascade:**
  - Cascade propagates the change to the referring table.
  - **DELETE CASCADE:** When we create a foreign key using this option, it deletes the referencing rows in the child table when the referenced row is deleted in the parent table which has a primary key.
    Example:

```
create table Took (
        …
        foreign key (sID) references Student on delete cascade
        …
);
```

  - **UPDATE CASCADE:** When we create a foreign key using UPDATE CASCADE the referencing rows are updated in the child table when the referenced row is updated in the parent table which has a primary key.
    Example:

```
create table Took (
        …
        foreign key (sID) references Student on update cascade
        …
);
```

- Note the asymmetry.
  Suppose table R refers to table S.
  You can define fixes that propagate changes backwards from S to R.
  You define them in table R because it is the table that will be affected.
  However, you cannot define fixes that propagate forward from R to S.
- Add your reaction policy where you specify the foreign key constraint, as shown above.

2. **Set Null:**

- Sets the referring attribute(s) to null.
  I.e. Set the corresponding value in the referring tuple to null.

3. **Restrict:**

- Don't allow the deletion/update.

Note: There are more methods.
Note: If you say nothing, the default is to forbid the change in the parent table.

- Your reaction policy can specify what to do on:

1. **On delete:**

- This is when a deletion creates a dangling reference.
- **On Delete Cascade:** When data is removed from a parent table, the foreign key associated cell will be deleted in the child table.
- **On Delete Set Null:** When data is removed from a parent table, the foreign key associated cell will be null in the child table.
- **On Delete Restrict:** When data is removed from a parent table, and there is a foreign key associated with the child table, it gives an error and you can not delete the record.

2. **On update:**

- **On Update Cascade:** If the parent primary key is changed, the child value will also change to reflect that.
- **On Update Set Null:** The SQL Server sets the rows in the child table to NULL when the corresponding row in the parent table is updated. Note that the foreign key columns must be nullable for this action to execute.
- **On Update Restrict:** When data is updated from a parent table, and there is a foreign key associated with the child table, it gives an error and you can not update the record.

3. **Both:**

- Just put them one after the other.
- E.g. on delete restrict on update cascade

- **Semantics of Deletion:**
- Consider the following cases:
  1. What if deleting a tuple violates a foreign key constraint?
  2. What if deleting one tuple violates a foreign key constraint, but deleting others does not?
  3. What if deleting one tuple affects the outcome for a tuple encountered later?

  To prevent such interactions, deletion proceeds in two stages:
  1. Mark all tuples for which the WHERE condition is satisfied.
  2. Go back and delete the marked tuples.
- **Note:** If you drop a table that is referenced by another table, you must specify cascade.

Functional Dependencies:

- A **functional dependency (FD)** is a relationship between two attributes, X and Y, if for every valid instance of X, that value of X uniquely determines the value of Y. This relationship is denoted as **X → Y**.
  I.e. If column X of a table uniquely identifies column Y of the same table then it can be represented as **X → Y**.
  A functional dependency **X → Y** in a relation holds if two or more tuples having the same value for X also have the same value for Y.
- The left side of the above FD notation is called the **determinant**, and the right side is the **dependent**.
- E.g.
  a. **SIN → Name, Birth date, Address** means that SIN determines Name, Address and Birthdate. Given a SIN, we can determine any of the other attributes within the table.
  b. **ISBN → Title** means that ISBN determines Title.
- **Types of functional dependencies:**
  1. **Multivalued dependency:**
  - **Multivalued dependency** occurs when there are more than one independent multivalued attributes in a table.
  - E.g.

Car_model	Maf_year	Color
H001	2017	Metallic
H001	2017	Green
H005	2018	Metallic
H005	2018	Blue
H010	2015	Metallic
H033	2012	Gray

In this example, maf_year and color are independent of each other but dependent on car_model. In this example, these two columns are said to be multivalued dependent on car_model.

This dependence can be represented like this:
car_model → maf_year
car_model → colour

2. **Trivial Functional dependency:**
- The dependency of an attribute on a set of attributes is known as **trivial functional dependency** if the set of attributes includes that attribute.
- **Note: A → A** is always a trivial functional dependency.
- E.g.
  Consider a table with two columns Student_id and Student_Name.

  {Student_Id, Student_Name} → Student_Id is a trivial functional dependency as Student_Id is a subset of {Student_Id, Student_Name}.

  Furthermore, **Student_Id → Student_Id** & **Student_Name → Student_Name** are trivial dependencies too.

3. **Non-trivial Functional Dependency:**

- A **non-trivial functional dependency** occurs when **A → B** where B is not a subset of A.
  I.e. If a functional dependency **X → Y** holds true where Y is not a subset of X then this dependency is called a non-trivial functional dependency.
- E.g.
  Consider an employee table with three attributes: emp_id, emp_name, and emp_address.
  The following functional dependencies are non-trivial:
  emp_id → emp_name (emp_name is not a subset of emp_id)
  emp_id → emp_address (emp_address is not a subset of emp_id)

  However, **{emp_id, emp_name} → emp_name** is trivial because emp_name is a subset of {emp_id, emp_name}.

4. **Transitive Dependency:**

- A functional dependency is said to be **transitive** if it is indirectly formed by two functional dependencies.
- **X → Z** is a transitive dependency if the following three functional dependencies hold true:
  - **X → Y**
  - Y does not → X
  - **Y → Z**
- E.g.

Book	Author	Author_age
Game of Thrones	George R. R. Martin	66
Harry Potter	J. K. Rowling	49
Dying of the Light	George R. R. Martin	66

  {Book} → {Author}
  {Author} does not → {Book}
  {Author} → {Author_age}
  Therefore as per the rule of transitive dependency:
  {Book} → {Author_age} should hold. That makes sense because if we know the book name we can know the author's age.
- **Note:** A transitive dependency can only occur in a relation of three or more attributes.

- **Axioms of functional dependency:**
  1. If we have X → Y and all the values in X are unique, then we know for sure that there is a valid functional dependency between X and Y.
  2. Similarly, if we have X → Y and all the values in Y are the same, then we know for sure that there is a valid functional dependency between X and Y.
  3. **Reflexive Axiom:** If X is a set of attributes and Y ⊆ X, then X → Y.
  4. **Augmentation Axiom:** If X → Y and Z is a set of attributes, then XZ → YZ.
  5. **Transitivity Axiom:** If X → Y and Y → Z, then X → Z.
  6. **Union Axiom:** If X → Y and X → Z, then X → YZ
  7. **Decomposition Axiom:** If X → YZ, then X → Y and X → Z.
  8. **Pseudo Transitivity Axiom:** If X → Y and WY → Z, then WX → Z.

9. **Composition Axiom:** If X → Y and Z → W, then XZ → YW.

Note: The reflexive, augmentation and transitivity axioms are called the **Armstrong Axioms**.

- **Closure/Attribute Closure:**
- Defined as "Given a set of attributes, what are the other attributes that can be fetched from it."
- The closure of an attribute, A, is denoted as A^+.
- **Equivalence of functional dependencies:**
- Let FD1 and FD2 are two FD sets for a relation R.
    1. If all FDs of FD1 can be derived from FDs present in FD2, we can say that FD1 ⊆ FD2.
    2. If all FDs of FD2 can be derived from FDs present in FD1, we can say that FD2 ⊆ FD1.
    3. If 1 and 2 both are true, FD1 = FD2.
- **Irreducible set of functional dependencies/Canonical Form:**
- Whenever a user updates the database, the system must check whether any of the functional dependencies are getting violated in this process. If there is a violation of dependencies in the new database state, the system must roll back. Working with a huge set of functional dependencies can cause unnecessary added computational time. This is where the canonical cover comes into play.
- A canonical cover of a set of functional dependencies F is a simplified set of functional dependencies that has the same closure as the original set F.
- **Examples:**
    1. Given the table and the functional dependencies below, show and explain which functional dependencies are valid and which are invalid.

A	B	C	D	E
a	2	3	4	5
a	2	3	6	5
2	a	3	4	5

A → BC
DE → C
C → DE
BC → A

Soln:

1. A → BC
   This one is valid because if you look at the table, under column A, there are 2 a's, and they both correspond to 2 in column B and 3 in column C.

2. DE → C
   This one is valid because there are 2 instances of {D: 4, E:5} and they both correspond to 3 in C.

3. C → DE
   This one is invalid because there are 3 instances of {C:3} but they correspond to different values in DE.
   In the second row, C = 3 corresponds to D = 6 and E = 5 while in rows 1 and 3, C = 3 corresponds to D = 4 and E = 5.

4. BC → A
   This one is valid because there are 2 instances of B = 2 and C = 3 and both times, they correspond to A = a.

2. Given a relational R with attributes A, B and C, R(A,B,C), and the following functional dependencies, find the closure of A.

   A → B
   B → C

   Soln:
   A^+ = {A, B, C} because A can determine A, and B. Furthermore, B can determine C.

3. Given a relational R with attributes A, B, C, D, E, and F, R(A,B,C,D,E,F), and the following functional dependencies, find the closure of D and DE.

   A → B
   C → DE
   AC → F
   D → AF
   E → CF

   Soln:
   D^+ = {A, B, D, F} because D can determine A, D and F. Furthermore, A can determine B.

   $(DE)^+$ = {A, B, C, D, E, F} because D can determine A, D and F. Furthermore, A can determine B. E can determine E, C and F.

4. Given R(A, B, C, D, E, F, G) and the following functional dependencies, find the closure of AC.

   A → B
   BC → DE
   AEG →G

   Soln:
   $(AC)^+$ = {A, C, B, D, E} because AC can determine A and C. Then, A can determine B. Then, BC can determine D and E.

5. Given R(A, B, C, D, E) and the following functional dependencies, find the closure of B.

   A → BC
   B → D
   CD → E
   E →A
   Soln:
   B^+ = {B, D} because B can determine B and D.

6. Given R(A, B, C, D, E, F) and the following functional dependencies, find the closure of AB.

   AB → C
   BC → DE
   D → E
   CA → B

   Soln:
   $(AB)^+$ = {A, B, C, D, E}

7. Given R(A, B, C, D, E, F, G, H) and the following functional dependencies, find the closure of BCD.

   A → BC
   CD → E
   E → C
   D → AEH
   ABH → BD
   DH → BC
   BCD → H

   Soln:
   $(BCD)^+$ = {B, C, D, H, E, A}

8. Given R(A, C, D, E, H) and the following 2 sets of functional dependencies
   Set 1:
   $A \rightarrow C$
   $AC \rightarrow D$
   $E \rightarrow ADH$

   Set 2:
   $A \rightarrow CD$
   $E \rightarrow AH$

   We want to know if the 2 sets of functional dependencies are equivalent.

   Soln:
   Step 1: Check if all of the FDs of Set 1 are in Set 2.
   To do so, I will compute the closures of A, AC and E using the functional dependencies of Set 2.

   A^+ = {A, C, D} (Knowing A, I can get A, C and D.)
   $(AC)^+$ = {A, C, D} (Knowing A, I can get A, C and D. Knowing C, I can get C.)
   E^+ = {E, A, H, C, D} (Knowing E, I can get E, A and H. Knowing A, I can get C and D.)

   Since the FDs of Set 1 are in the closure of each LHS item computed using the FDs of set 2, we know that Set 1 $\subseteq$ Set 2.
   I.e.
   A^+ in Set 1 = {A, C} but A^+ computed using the FDs of Set 2 = {A, C, D}.
   $(AC)^+$ in Set 1 = {A, C, D} but $(AC)^+$ computed using the FDs of Set 2 = {A, C, D}.
   E^+ in Set 1 = {E, A, D, H} but E^+ computed using the FDs of Set 2 = {E, A, H, C, D}.
   Hence, Set 1 $\subseteq$ Set 2.

   Step 2: Check if all of the FDs of Set 2 are in Set 1.
   To do so, I will compute the closures of A and E using the functional dependencies of Set 1.

   A^+ = {A, C, D} (Knowing A, I can get A and C. Knowing AC, I can get D.)
   E^+ = {E, A, D, H, C} (Knowing E, I can get E, A, D and H. Knowing A, I can get C.)

   A^+ in Set 2 = {A, C, D} but A^+ computed using the FDs of Set 1= {A, C, D}.
   E^+ in Set 2 = {E, A, H} but A^+ computed using the FDs of Set 1 = {E, A, D, H, C}.
   Hence, Set 2 $\subseteq$ Set 1.

   Since Set 1 $\subseteq$ Set 2 and Set 2 $\subseteq$ Set 1, Set 1 = Set 2.

9. Given R(P, Q, R, S) and the following 2 sets of functional dependencies
   Set 1:
   $P \rightarrow Q$
   $Q \rightarrow R$
   $R \rightarrow S$

   Set 2:
   $P \rightarrow QR$
   $R \rightarrow S$

   We want to know if the 2 sets of functional dependencies are equivalent.

   Soln:
   Step 1:
   $P^+ = \{P, Q, R, S\}$ (Knowing P, I can get P, Q and R. Knowing R, I can get S.)
   $Q^+ = \{Q\}$ (Knowing Q, I can get Q.)
   $R^+ = \{R, S\}$ (Knowing R, I can get R and S.)
   Here, Set 1 $\not\subseteq$ Set 2 because in Set 1, $Q^+ = \{Q, R\}$ while in Set 2, $Q^+ = \{Q\}$.

   Step 2:
   $P^+ = \{P, Q, R, S\}$ (Knowing P, I can get P and Q. Knowing Q, I can get R. Knowing R, I can get S.)
   $R^+ = \{R, S\}$ (Knowing R, I can get R and S.)
   Here, Set 2 $\subseteq$ Set 1.

   Therefore, Set 2 $\subseteq$ Set 1.

10. Given R(A, B, C) and the following 2 sets of functional dependencies
    Set 1:
    $A \rightarrow B$
    $B \rightarrow C$
    $C \rightarrow A$
    Set 2:
    $A \rightarrow BC$
    $B \rightarrow A$
    $C \rightarrow A$
    We want to know if the 2 sets of functional dependencies are equivalent.

    Soln:
    Step 1:
    $A^+ = \{A, B, C\}$
    $B^+ = \{B, A, C\}$
    $C^+ = \{C, A, B\}$
    Here, Set 1 $\subseteq$ Set 2.
    Step 2:
    $A^+ = \{A, B, C\}$
    $B^+ = \{B, C, A\}$
    $C^+ = \{C, A, B\}$
    Here, Set 2 $\subseteq$ Set 1.

    Therefore, Set 1 = Set 2.

11. Given R(V, W, X, Y, Z) and the following 2 sets of functional dependencies
    Set 1:
    $W \rightarrow X$
    $WX \rightarrow Y$
    $Z \rightarrow WY$
    $Z \rightarrow V$

    Set 2:
    $W \rightarrow XY$
    $Z \rightarrow WX$

    We want to know if the 2 sets of functional dependencies are equivalent.

    Soln:
    Step 1:
    $W^+ = \{W, X, Y\}$
    $(WX)^+ = \{W, X, Y\}$
    $Z^+ = \{Z, W, X, Y\}$
    Here, Set 1 $\not\subseteq$ Set 2. (V is not in Z^+.)

    Step 2:
    $W^+ = \{W, X, Y\}$
    $Z^+ = \{Z, W, Y, V, X\}$
    Here, Set 2 $\subseteq$ Set 1.

    Therefore, Set 2 $\subseteq$ Set 1.

12. Given R(W, X, Y, Z) and the following set of functional dependencies

    $X \rightarrow W$
    $WZ \rightarrow XY$
    $Y \rightarrow WXZ$

    We want to check for redundancy.

    Soln:
    The redundancy can occur at $\propto$, β or $\propto \rightarrow \beta$.

    Step 1: We will remove redundancies at the β level.
    To do this, we will apply the decomposition rule.
    $X \rightarrow W$
    $WZ \rightarrow X$
    $WZ \rightarrow Y$
    $Y \rightarrow W$
    $Y \rightarrow X$
    $Y \rightarrow Z$

Now, we will find the closure of each item on the LHS, first with the FD and second without the FD.

Variable	**With FD**	**Without FD**
X^+	{X, W}	{X} (Without X → W)
$(WZ)^+$	{W, Z, X, Y}	{W, Z, X, Y} (Without WZ → X) **Redundant**
$(WZ)^+$	{W, Z, X, Y}	{W, Z, X} (Without WZ → Y)
Y^+	{Y, W, X, Z}	{Y, X, Z, W} (Without Y → W) **Redundant**
Y^+	{Y, W, X, Z}	{Y, Z, W, X} (Without Y → X) **Redundant**
Y^+	{Y, W, X, Z}	{Y, X, W} (Without Y → Z)

A FD is redundant if it can be recreated some other way.
Hence, the following FDs are redundant:
WZ → X
Y → W
Y → X

The canonical form are the following FDs:
X → W
WZ → Y
Y → Z

Step 2: We will remove redundancies at the ∝ level.
Note: We can't decompose WZ because if we do, we will get different closures.
$(WZ)^+$ = {W, Z, X, Y}
W^+ = {W}
Z^+ = {Z}
Since we can't decompose WZ, nothing changes.

13. Given R(A, B, C, D) and the following set of functional dependencies

A → B
C → B
D → ABC
AC →D

We want to check for redundancy.

Soln:
Step 1: We will remove redundancies at the β level.
A → B
C → B
D → A
D → B
D → C
AC →D

Variable	**With FD**	**Without FD**
A^+	{A, B}	{A} (Without A → B)
C^+	{C, B}	{C} (Without C → B)
D^+	{D, A, B, C}	{D, B, C} (Without D → A)
D^+	{D, A, B, C}	{D, A, C, B} (Without D → B) **Redundant**
D^+	{D, A, B, C}	{D, A, B} (Without D → C)
$(AC)^+$	{A, C, D, B}	{A, C, B} (Without AC → D)

The following FD is redundant:
D → B

The canonical form are the following FDs:
A → B
C → B
D → A
D → C
AC →D

Step 2: We will remove redundancies at the ∝ level.
Note: We can't decompose AC because if we do, we will get different closures.
$(AC)^+$ = {A, C, D, B}
A^+ = {A, B}
C^+ = {C, B}
Since we can't decompose AC, nothing changes.

14. Given R(V, W, X, Y, Z) and the following set of functional dependencies

V → W
VW → X
Y → VXZ

We want to check for redundancy.

Soln:
Step 1: We will remove redundancies at the β level.
V → W
VW → X
Y → V
Y → X
Y → Z

Variable	**With FD**	**Without FD**
V^+	{V, W, X}	{V} (Without V → W)
$(VW)^+$	{V, W, X}	{V, W} (Without VW → X)
Y^+	{Y, V, X, Z, W}	{Y, X, Z} (Without Y → V)
Y^+	{Y, V, X, Z, W}	{Y, V, Z, W, X} (Without Y → X) **Redundant**
Y^+	{Y, V, X, Z, W}	{Y, V, X, W} (Without Y → Z)

The following FD is redundant:
Y → X

The canonical form are the following FDs:
V → W
VW → X
Y → V
Y → Z

Step 2: We will remove redundancies at the ∝ level.
$(VW)^+$ = {V, W, X}
V^+ = {V, W, X}
W^+ = {W}

Hence, we can rewrite VW → X into V → X.

The canonical form are the following FDs:
V → W
V → X
Y → V
Y → Z

Tutorial Notes:

E.g. 1. Determine if the 2 sets of FDs are equivalent.

Set 1
$A \rightarrow BC$

Set 2
$A \rightarrow B$
$A \rightarrow C$

Soln:
Step 1: Check if all the FDs of Set 1 are in Set 2.
To do so, compute the closure of A using the FDs of Set 2.
$A^+ = \{A, B, C\}$
We can see that A^+ in Set 1 is a subset of A^+ computed using the FDs of Set 2.
Hence, Set 1 $\subseteq$ Set 2.

Step 2: Check if all the FDs of Set 2 are in Set 1.
To do so, compute the closure of A using the FDs of Set 1.
$A^+ = \{A, B, C\}$
We can see that A^+ in Set 2 is a subset of A^+ computed using the FDs of Set 1.
Hence, Set 2 $\subseteq$ Set 1.

Since Set 1 $\subseteq$ Set 2 and Set 2 $\subseteq$ Set 1, Set 1 = Set 2, meaning they are equivalent.

E.g. 2. Suppose the FD $BC \rightarrow D$ holds for relation R. Create an instance of relation R that breaks the FD. R(A, B, C, D)

Soln:

A	B	C`	D
1	1	2	3
2	1	2	4

We see that there are 2 instances of 1|2 for B|C, but their D values are different. Hence, we cannot use B|C to uniquely determine D.

E.g. 3. Determine if the 2 sets of FDs are equivalent.

Set 1
$PQ \rightarrow R$

Set 2
$P \rightarrow R$
$Q \rightarrow R$

Soln:
Step 1: Get the closure of PQ using the FDs from Set 2.
$(PQ)^+ = \{P, Q, R\}$

Step 2: Get the closure of P and Q using the FDs from Set 1.
$(P)^+ = \{P\}$
$(Q)^+ = \{Q\}$

We can see that Set 1 $\subseteq$ Set 2 but Set 2 $\not\subseteq$ Set 1.
Hence, they are not equivalent.

E.g. 4. Determine if the 2 sets of FDs are equivalent.

Set 1
$PQ \rightarrow R$

Set 2
$P \rightarrow Q$
$P \rightarrow R$

Soln:
Step 1: Get the closure of PQ using the FDs from Set 2.
$(PQ)^+ = \{P, Q, R\}$

Step 2: Get the closure of P using the FDs from Set 1.
$(P)^+ = \{P\}$

We can see that Set 1 $\subseteq$ Set 2 but Set 2 $\not\subseteq$ Set 1.
Hence, they are not equivalent.

E.g. 5. Given R(A, B, C, D, E, F) and the FDs
$AC \rightarrow F$
$CEF \rightarrow B$
$C \rightarrow D$
$DC \rightarrow A$

a. Does the FD $C \rightarrow F$ hold?

   Soln:
   Another way of thinking about this is “Does the closure of C include F?”
   The closure of C = {C, D, A, F}.
   Yes, $C \rightarrow F$ holds.

b. Does the FD $ACD \rightarrow B$ hold?

   Soln:
   The closure of ACD = {A, C, D, F}.
   Hence, $ACD \rightarrow B$ does not hold.

Projection:
E.g. 1. Given R(A, B, C, D, E) and the FDs
$A \rightarrow C$
$C \rightarrow E$
$E \rightarrow BD$

Project these FDs onto R1(A, B, C).

Soln:
Let's find the closure of A, B and C.
$A^+ = \{A, C, E, B, D\}$ However, since R1 only has the attributes A, B and C, we get the FD $A \rightarrow BC$. We ignore $A \rightarrow A$ because it's a trivial FD.
$B^+ = \{B\}$ This is a trivial FD, so we ignore it.
$C^+ = \{C, E, B, D\}$ However, since R1 only has the attributes A, B and C, we get the FD $C \rightarrow B$.

The projection of the FDs onto R1 is $\{A \rightarrow BC, C \rightarrow B\}$.

E.g. 2. Given R(A, B, C, D, E) and the FDs
$A \rightarrow C$
$C \rightarrow E$
$E \rightarrow BD$

Project these FDs onto R1(A, D, E).

Soln:
Let's find the closure of A, D and E.
$A^+ = \{A, C, E, B, D\}$ However, since R1 only has the attributes A, D and E, we get the FD $A \rightarrow DE$. We ignore $A \rightarrow A$ because it's a trivial FD.
$D^+ = \{D\}$ This is a trivial FD, so we ignore it.
$E^+ = \{E, B, D\}$ However, since R1 only has the attributes A, D and E, we get the FD $E \rightarrow D$.

The projection of the FDs onto R1 is $\{A \rightarrow DE, E \rightarrow D\}$.

E.g. 3. Given R(A, B, C) and the FDs
$A \rightarrow B$
$B \rightarrow C$

Project these FDs onto R1(A, C).

Soln:
Let's find the closure of A and C.
$A^+ = \{A, B, C\}$ However, since R1 only has the attributes A and C, we get the FD $A \rightarrow C$. We ignore $A \rightarrow A$ because it's a trivial FD.
$C^+ = \{C\}$ This is a trivial FD, so we ignore it.

The projection of the FDs onto R1 is $\{A \rightarrow C\}$.

E.g. 4. Given R(A, B, C, D) and the FDs
A → B
B → C
C → D

Project these FDs onto R1(A, C, D).

Soln:
Let's find the closure of A, C, and D.
A^+ = {A, B, C, D} However, since R1 only has the attributes A, C and D, we get the FDs A → C and A → D. We ignore A → A because it's a trivial FD.
C^+ = {C, D} However, since R1 only has the attributes A, C and D, we get the FD C → D. We ignore C → C because it's a trivial FD.
D^+ = {D} This is a trivial FD, so we ignore it.

The projection of the FDs onto R1 is {A → C, A → D and C → D}.

E.g. 5. Given R(A, B, C, D, E, F) and the FDs
A → BC
C → DE
E → A

Project these FDs onto R1(A, C, E).

Soln:
Let's find the closure of A, C, and E.
A^+ = {A, B, C, D, E} However, since R1 only has the attributes A, C and E, we get the FDs A → C and A → E. We ignore A → A because it's a trivial FD.
C^+ = {C, D, E, A} However, since R1 only has the attributes A, C and E, we get the FDs C → E and C → A. We ignore C → C because it's a trivial FD.
E^+ = {E, A, B, C, D} However, since R1 only has the attributes A, C and E, we get the FDs E → A and E → C. We ignore E → E because it's a trivial FD.

The projection of the FDs onto R1 is {A → C, A → E, C → A, C → E, E → A and E → C}.

Super Keys:

- A **super key** is a set of one or more attributes, which can uniquely identify a row in a table.
- A super key may have additional attributes that are not needed for unique identification.
- **E.g. 1.** Suppose that we have a relation called Students with the attributes id, first_name, last_name and average and {id} is a super key.
  Since {id} is a super key, then the following are also super keys:
  - {id, first_name}
  - {id, last_name}
  - {id, average}
  - {id, first_name, last_name}
  - {id, first_name, average}
  - {id, last_name, average}
  - {id, first_name, last_name, average}

  This is because since id can uniquely identify a row in Students, anything else we add to the set can also uniquely identify a row in Students.
- Another way to define super keys is through closure. The closure of a super key should give back the entire relation.
- **E.g. 2.** Given R(A, B, C) and A → BC. Determine if A is a super key.

  Soln:
  The closure of A, A^+ = {A, B, C}.
  Since the closure of A gives back the entire relation R, it is a super key.
- **E.g. 3.** Given R(A, B, C, D) and
  ABC → D
  AB → CD
  A → BCD

  What are the super key(s) if any exist?

  Soln:
  Since we don't have A on the RHS of any fd, we know that our super key must contain at least A.
  Furthermore, we see that A^+ = {A, B, C, D}. Hence, A is a super key. This means that the following are all super keys:
  {A, B}
  {A, C}
  {A, D}
  {A, B, C}
  {A, B, D}
  {A, C, D}
  {A, B, C, D}

Candidate Keys:

- A **candidate key** is a minimal super key.
  I.e. It is the minimal set of attributes needed to uniquely identify a row in a table.
  In example 1, only {id} is a candidate key.
  In example 3, only {A} is a candidate key.
- Properties of candidate keys:
  - It must contain unique values.
  - It may have multiple attributes.
  - It must not contain null values.

    - It should contain the minimum fields to ensure uniqueness.
    - It should uniquely identify each record in a table.
- A table can have multiple candidate keys.
- All candidate keys are super keys but not all super keys are candidate keys.
- **E.g. 4.** Given R(A, B, C, D) and
  B → ACD
  ACD → B

  List out all the candidate keys, if there are any.

  Soln:
  B^+ = {A, B, C, D}
  $(ACD)^+$ = {A, B, C, D}.
  Hence, both {B} and {A, C, D} are both super keys.
  However, because they are both minimal, they are also candidate keys.
  Note that for ACD, you cannot break it down and still get all the attributes in R.
  A^+, C^+, D^+, $(AC)^+$, $(AD)^+$, $(CD)^+$ do not give you all the relations in R.
  Hence, ACD is minimal.
  So in this case, we have 2 candidate keys for the relation R.
- **E.g. 5.** Given R(A, B, C, D) and
  AB → C
  C → BD
  D → A

  List all the candidate keys, if there are any.

  Soln:
  $(AB)^+$ = {A, B, C, D}
  C^+ = {A, B, C, D}
  D^+ = {A, D}

  In this example, both {A, B} and {C} are candidate keys.
- **E.g. 6.** Given R(A, B, C, D) and
  A → B
  B → C
  C → A

  List all the candidate keys, if there are any.

  Soln:
  First, notice that neither A, B nor C can get you column D. Hence, we know that our candidate key must contain D.

  A^+ = {A, B, C}
  B^+ = {A, B, C}
  C^+ = {A, B, C}

  Hence, the candidate keys are {A, D}, {B, D}, and {C, D}.

- **E.g. 7.** Given R(A, B, C, D) and
  AB → CD
  D → A
  List all the candidate keys, if there are any.

  Soln:
  $(AB)^+$ = {A, B, C, D}
  D^+ = {A, D} ← Notice that the closure of D has all the relations except for B and C. We know that AB gets us B and C and we already have A.
  $(BD)^+$ = {A, B, C, D}

  Hence, {A, B} and {B, D} are candidate keys.
- **E.g. 8.** Given R(A, B, C, D, E, F) and
  AB → C
  C → D
  B → AE

  List all the candidate keys, if there are any.

  Soln:
  $(AB)^+$ = {A, B, C, D, E}
  B^+ = {A, B, C, D, E} ← Only missing column F.
  C^+ = {C, D}

  Hence, {B, F} is the only candidate key.
- **E.g. 9.** Given R(A, B, C, D) and
  AB → CD
  C → A
  D → B

  List all the candidate keys, if there are any.

  Soln:
  $(AB)^+$ = {A, B, C, D}
  C^+ ={A, C} ← Missing B and D. We know that AB gets us CD, so {B, C} is a candidate key as we already have A.
  D^+ = {B, D} ← Missing A and C. We know that AB gets us CD, so {A, D} is a candidate key as we already have B.

  Hence, {A, B}, {B, C}, {C, D} and {A, D} are the candidate keys.

Primary Keys:

- A **primary key** is a chosen candidate key.
  I.e. There could be multiple candidate keys. From the options, we choose one to use. The one that we chose to use is the primary key.
- Rules for defining primary keys:
  - Two rows can't have the same primary key value.
  - The primary key field cannot be null.
  - The value in a primary key column can never be modified or updated if any foreign key refers to that primary key.
- **Prime attributes** are the attributes of the candidate key(s).
- **Non-prime attributes** are the attributes of a table not in the candidate key(s).

Normalization:

- **Normalization** is a database design technique that reduces data redundancy and eliminates undesirable characteristics like insertion, update and deletion anomalies.
- Normalization divides larger tables into smaller tables and links them using relationships.
- The purpose of normalization is to eliminate repetitive data and ensure data is stored logically.
- **E.g. 10.** Consider the table below:

  Student

SID	Name	Program	Department Head	Department Head's Phone Number
1	A	CSC	X	100-100-1000
1	A	MAT	Y	100-100-1001
2	B	CSC	X	100-100-1000
3	C	MAT	Y	100-100-1001
4	D	STA	Z	100-100-1002

  Here are some problems with this design:

  1. Suppose we enroll a new student who's not in any program. Then, the program, department head and department head's phone number will be blank. This is an example of **insertion anomaly**.
  2. Suppose that a department head gets changed. Then, we would have to change that information for multiple students, and if by mistake we miss any record, it will lead to data inconsistency. This is an example of **updation anomaly**.
  3. We see that the department head and department head's phone number information are repeated for the students who are in that program. This is an example of **data redundancy**.
  4. Suppose that student D graduated and all rows pertaining to student D gets deleted. If student D is the only student in the stats program, then we lose important information, such as student D's program, the program's department chair and the department chair's phone number, when we delete all rows pertaining to student D. This is an example of **deletion anomaly**.

- Anomalies are caused when there is too much redundancy in the database's information.
- **Update anomaly** happens when there are multiple entries of the same data in the db and when we update that data, one or more entries do not get updated. Then, we will have data inconsistency.
- **Insertion anomaly** happens when inserting vital data into the database is not possible because other data is not already there.
- **Deletion anomaly** happens when the deletion of unwanted information causes desired information to be deleted as well.
- There are a few normalization rules we can use:
  - 1NF (First Normal Form)
  - 2NF (Second Normal Form)
  - 3NF (Third Normal Form)
  - BCNF (Boyce and Codd Normal Form)
  - 4NF (Fourth Normal Form)

- **1NF (First Normal Form):**
- For a table to be in the First Normal Form, it must follow the following rules:
    1. Each table cell should contain a single value.
    2. Each record needs to be unique.
    3. Values stored in a column should be of the same domain
    4. All the columns in a table should have unique names.
- **2NF (Second Normal Form):**
- For a table to be in the Second Normal Form, it must follow the following rules:
    1. It is already in First Normal Form.
    2. It must not have **partial dependency**. **Partial dependency** occurs when a non-prime attribute in a table depends on only a part of the candidate key and not on the whole candidate key.
       I.e. A partial dependency occurs when we have P → NP where P is 1 or more prime attributes but is not a candidate/primary key and NP is 1 or more non-prime attributes.
       E.g. 11. Consider R(A, B, C, D) and
       AB → D
       B → C

       We see that the candidate key is {A, B}. However, R is not in 2NF because the attribute C only depends on B and not A & B. This is an example of partial dependency.

       To change R to 2NF, we have to decompose it so that the partial dependencies are its own tables.

       For this example, we decompose R(A, B, C, D) into
       R1(A, B, D) and
       R2(B, C)
       Note: When you decompose R into smaller relations, you always want a relation with the primary keys. In this case, we have R1, so we don't need an additional table.
- **E.g. 12.** Consider R(A, B, C) and
  AB → C
  B → C

  We see that the candidate key is {A, B}. We see that B → C is a partial dependency.

  To change R to 2NF, we have to decompose it so that the partial dependencies are its own tables.

  For this example, we decompose R(A, B, C) into
  R1(A, B) and
  R2(B, C)

  We don't have C in R1 because we already have C in R2.
- **E.g. 13.** Consider R(A, B, C, D, E) and
  AB → C
  D → E

We see that the candidate key is {A, B, D}. We see that AB → C and D → E are partial dependencies.

To change R to 2NF, we have to decompose it so that the partial dependencies are its own tables.

For this example, we decompose R(A, B, C, D, E) into
R1(A, B, C)
R2(D, E)
R3(A, B, D)

- **E.g. 14.** Consider R(A, B, C, D, E) and
  A → B
  B → E
  C → D

  We see that the candidate key is {A, C}. We see that A → B and C → D are partial dependencies.

  To change R to 2NF, we have to decompose it so that the partial dependencies are its own tables.

  For this example, we decompose R(A, B, C, D, E) into
  R1(A, B, E)
  R2(C, D)
  R3(A, C)

  Note: B → E is not a partial dependency as B is not part of the primary key. For 2NF, we simply find the relation that contains B, which is R1, and add E to it.
- **3NF (Third Normal Form):**
- A table is in third normal form if:
  1. It is in 2nd normal form.
  2. It must not have **transitive dependencies**.

  Recall: A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies.
  I.e. If A → B and B → C, then A → C is a transitive dependency.
  Another way to think about transitive dependency is that it occurs when a non-prime attribute depends on other non-prime attributes. So, a transitive dependency occurs when you have NP → NP.
- The normalization of 2NF relations to 3NF relations involves the removal of transitive dependencies. If a transitive dependency exists, we remove the transitively dependent attribute(s) from the relation by placing the attribute(s) in a new relation along with a copy of the determinant.
  Recall: The left side of a functional dependency is called the determinant.
- **E.g. 15.** Consider R(A, B, C) and
  A → B
  B → C

  We see that A^+ = {A, B, C}, so {A} is the candidate key and A is a prime attribute. Furthermore, we see that we have a transitive dependency B → C.

What we do is we split R into 2 relations:
R1(A, B)
R2(B, C)

- Let P be prime attribute(s) and NP be non-prime attribute(s) and suppose that {P} is not a candidate/primary key. Then, we have
    1. **Partial dependency** if we have P → NP.
    2. **Transitive dependency** if we have NP → NP.

If we have P/NP → P, we know for sure that it is in 3NF.

- **E.g. 16.** Consider R(A, B, C, D, E) and
A → B
B → E
C → D

We see that $(AC)^+$ = {A, B, C, D, E}, so {A, C} is a candidate key.
We see that we have
1. A → B (Partial dependency)
2. C → D (Partial dependency)
3. B → E (Transitive dependency)

To turn R into 3NF, we will break it down into the following relations:
R1(A, B, E) ← Since B → E, we put E here. However, B → E is a transitive dependency, so we have to split up R1. We will split R1 up into R11 and R12.
R11(A, B)
R12(B, E)
R2(C, D)

R3(A, C) ← **Note:** When you decompose R into smaller relations, you always want a relation with the primary keys. In this case, we need to create a new relation to get a relation with the primary keys.

The final decomposition of R is:
R11(A, B)
R12(B, E)
R2(C, D)
R3(A, C)

- **E.g. 17.** Consider R(A, B, C, D, E, F, G, H, I, J) and
AB → C
A → DE
B → F
F → GH
D → IJ

A candidate key is {A, B} as the closure of (AB) gets back all attributes in R.
We see that
1. A → DE is a partial dependency (pd).
2. B → F is a pd.
3. F → GH is a transitive dependency (td).
4. D → IJ is a td.

We want to decompose R so that it is in 3NF.

We start with R1(A, D, E, I, J). We know that D → IJ, so we put I and J here. However, D → IJ is a td, so we have to split up R1 into R11 and R12.
R11(A, D, E)
R12(D, I, J)
Next, we have R2(B, F, G, H). We know that F → GH, so we put G and H here. However, F → GH is a td, so we split up R2 into R21 and R22.
R21(B, F)
R22(F, G, H)

R3(A, B, C)

The final decomposition of R is:
R11(A, D, E)
R12(D, I, J)
R21(B, F)
R22(F, G, H)
R3(A, B, C)

- **E.g. 18.** Consider R(A, B, C, D, E) and
AB → C
B → D
D → E

A candidate key is {A, B} as the closure of (AB) gets back all attributes in R.
We see that B → D is a pd and that D → E is a td.
We need to decompose R.
We start with R1(B, D, E). We know that D → E, so we put E here. However, D → E is a td, so we have to split R1 into R11 and R12.
R11(B, D)
R12(D, E)

Next, we have R2(A, B, C).

The final decomposition of R is:
R11(B, D)
R12(D, E)
R2(A, B, C)

- **BCNF (Boyce and Codd Normal Form):**
- For a table to be in BCNF, following conditions must be satisfied:
    1. It must be in 3NF.
    2. For each functional dependency (X → Y), X must be a super key.
- **E.g. 19.** Consider R(A, B, C) with the fds
AB → C
C → B

We see that {A, B} and {A, C} are candidate keys.
Hence, A, B and C are all prime attributes.
AB → C is neither a pd nor td.
C → B is neither pd or td because both the LHS and the RHS have prime attributes.
Hence, we see R is in 3NF.
However, R is not in BCNF because in C → B, C is not a super key.

To fix this, I'll decompose R into
R1(C, B)
R2(A, C) ← We chose (A, C) over (A, B) to prevent loss of data when joining R1 and R2.

- **E.g. 20.** Given R(A, B, C, D, E, F, G, H) and
AB → C
A → DE
B → F
F → GH

What form is it?

Soln:
We see that a candidate key is {A, B}.
We see that A → DE is a pd. Hence, R is in 1NF only.

- **E.g. 21.** Given R(A, B, C, D, E) and
CE → D
D → B
C → A

What form is it?

Soln:
We see that a candidate key is {C, E}.
We see that C → A is a pd. Hence, R is in 1NF only.

- **4NF (Fourth Normal Form):**
- For a table to satisfy the Fourth Normal Form, it should satisfy the following two conditions:
    1. It should be in the **Boyce-Codd Normal Form**.
    2. For each non-trivial multi-valued dependency A -->> B, A is a key.
- A **multi-valued dependency** occurs when two attributes in a table are independent of one another, but both depend on a third attribute.
- Here is the formal definition for multi-valued dependency:
Let R be a relation. Let A, B and rest be attributes. Let t, u, and v be tuples.
$\forall t, u \in R$ if t[A] = u[A], then $\exists v \in R$ s.t. v[A] = t[A] and v[B] = t[B] and v[rest] = u[rest].

We can show a picture of this.
R

	A	B	Rest
t	a	b1	r1
u	a	b2	r2
v	a	b1	r2
w	a	b2	r1

In our definition, we said that "For all tuples t and u in relation R, if t[A] equals to u[A], then there exists a tuple v in R such that v[A] = t[A] and v[B] = t[B] and v[rest] = u[rest]." In the table above, we can see that t[A] = u[A] = a. Furthermore, we didn't specify that t[B] = u[B] or t[rest] = u[rest], so we have different values for those. We can create tuple

v based on the definition and tuples t and u. By swapping the roles of t and u, we can create tuple w.

- A table is said to have multi-valued dependency, if the following conditions are true:
    1. For a dependency A → B, if for a single value of A, multiple values of B exists, then the table may have multi-valued dependency.
    2. A table should have at-least 3 columns for it to have a multi-valued dependency.
    3. For a relation R(A,B,C), if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.
- **Note:** Every FD is an MVD. This is because if X → Y, then swapping Y's between tuples that agree on X does not create new tuples.
  I.e. X → Y implies X -->> Y
- **Note:** If X -->> Y, then X -->> R - Y - X.
- I.e. If we have a relation R(A, B, C, D) and we have A -->> B, then we also have A -->> CD.
- The main idea of 4NF is to eliminate redundancy due to the multiplicative effect of MVDs.
- **E.g. 22.** Consider the table below:

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

We see that course and hobby are independent of each other, but are dependent on stu_id. Furthermore, we see that for stu_id value of 21, there's 2 different corresponding course values and 2 different corresponding hobby values. Hence, the table has multi-valued dependency.

- We use -->> to denote a multi-valued dependency.
  I.e. For the table above in example 22, **stu_id -->> course** and **stu_id -->> hobby**.
- **E.g. 23.** Consider the relation Apply(SSN, collegeName, hobby) and the fact that SSN -->> collegeName (cName) and SSN -->> hobby.

Based on the information above, we can create a table below:

SSN	cName	Hobby
123	Stanford	Trumpet
123	Berkeley	Tennis
123	Stanford	Tennis
123	Berkeley	Trumpet

Notice how all possible combinations of cName and hobby are listed.

- A **trivial multivalued functional dependency** occurs when X -->> Y and

1. $Y \subseteq X$ (Y is a subset of X) or
2. $X \cup Y$ gets back all the attributes of the relation.
   I.e. There's no "rest."

- A **non-trivial multi-valued functional dependency** occurs otherwise.
- Rules of multi-valued functional dependency:
  1. If we have $A \rightarrow B$, then we also have A -->> B.
     Proof:
     Consider the template table below and the fact that $A \rightarrow B$.

	A	B	Rest
t	a1	b1	r1
u	a1	b2	r2

     We want to prove that there exists a tuple v with the following values:

	A	B	C
v	a1	b1	r2

     We know that v exists for the following reasons:
     a. Since $A \rightarrow B$, and we have a1 | b1 and a1 | b2, we know that b1 = b2.
     b. Since b1 = b2, go back to v and rewrite b1 as b2. You'll see that now, you have a1 | b2 | r2. This row exists and is tuple u.
     c. Hence, tuple v exists.
     d. Hence, if $A \rightarrow B$, then A -->> B.
  2. **Intersection Rule:** If A -->> B and A -->> C, then A -->> B∩C.
  3. **Transitive Rule:** If A -->> B and B -->> C, then A -->> B-C.
- Functional dependencies are a subset of multi-valued dependencies.
  This means that any rules for multi-valued dependencies apply to functional dependencies but rules for functional dependencies may not apply to multi-valued dependencies.
- Here's the algorithm to decompose a relation into Fourth Normal Form:
  Input: Relation R + FDs for R + MVDs for R
  Output: Decomposition of R into 4NF relations with lossless joins.
  Steps:
  1. Compute the candidate keys for R.
  2. Repeat until all relations are in 4NF:
     a. Pick any R' with nontrivial A -->> B that violates 4NF.
     b. Decompose R' into R1(A, B) and R2(A, Rest)
     c. Compute FDs and MVDs for R1 and R2.
     d. Compute keys for R1 and R2.
- **E.g. 24.** Consider the relation Apply(SSN, collegeName, hobby) and the fact that SSN -->> collegeName (cName). Decompose Apply such that all relations are in 4NF.

  Solution:
  A1(SSN, cName)
  A2(SSN, hobby)

- **E.g. 25.** Consider the relation Apply(SSN, collegeName, date, major, hobby) and the fact that
  SSN, cName → date
  SSN, cName, date -->> major
  Decompose Apply such that all relations are in 4NF.

  Solution:
  A1(SSN, cName, date, major)
  A2(SSN, cName, date, hobby)

  We need to break up A1 and A2 further. I'll break A1 into A11 and A12 and A2 into A21.

  A11(SSN, cName, date)
  A12(SSN, cName, major)
  A21(SSN, cName, hobby)

Tutorial Notes:
E.g. 1. Given R(A, B, C, D) and the FDs
A → BC
AC → D
D → ABC
C → A

List out all the candidate keys.

Soln:
First, we try to look for any attribute(s) that are not on the RHS on any FD. If such attribute(s) exist, we know that all candidate keys must include those attribute(s) since the only way to derive those attribute(s) is from themselves. However, in this example, all attributes are listed on the RHS of some FD.

We see that A^+ = {A, B, C, D}, so {A} is a candidate key.
We see that C^+ = {A, B, C, D}, so {C} is a candidate key.
We see that D^+ = {A, B, C, D}, so {D} is a candidate key.

Hence, the candidate keys are {A, C, D}.
Lossy Decomposition:

- The decomposition of relation R into R1 and R2 is **lossy** when the join of R1 and R2 does not yield the same relation as in R.
- **Note:** A **lossy decomposition** does not necessarily mean that you lost data. You could also have gained false/incorrect data when you join the tables. It simply means that when you join the tables, it does not yield the same relation as the original table.
- One of the disadvantages of decomposition into two or more relations is that some information is lost during retrieval of original relation or table.
- Decomposition is **lossless** if it is feasible to reconstruct relation R from decomposed tables using Joins.

- **E.g. 2.** Consider the table below:

R

utorid	**name**	**grade**
g3tout	Amy	91
g4foobar	David	78
c0zhang	David	85

Suppose we broke R into the following 2 tables:

R1

utorid	**name**
g3tout	Amy
g4foobar	David
c0zhang	David

R2

name	**grade**
Amy	91
David	78
David	85

Lets see what happens when we join R1 and R2.

utorid	**name**	**grade**
g3tout	Amy	91
g4foobar	David	78
g4foobar	David	85
c0zhang	David	78
c0zhang	David	85

We see that we get 2 extra rows, the underlined rows, that weren't there before. This is an example of a lossy decomposition. Notice that we didn't lose any of the original data but we got false/incorrect data.

First Normal Form (1NF):

- For a table to be in 1NF, every cell in the relation can only take on a single value.
- **E.g. 3.** Consider the table below:

SID	**Major**
123	Math, Computer Science

Since there are 2 values under the Major column, this is not in 1NF. To fix it, we can do this:

SID	**Major**
123	Math
123	Computer Science

Second Normal Form (2NF):

- For a table to be in 2NF:
  1. It must already be in 1NF and
  2. It must not have any **partial dependencies**.
- A **partial dependency** occurs when non-prime attributes depend on a proper subset of the prime attributes.
  I.e. Let P be a proper subset of the prime attributes and let NP be some non-prime attributes. If we have P → NP, we have partial dependency.
- Here are the steps on how you can decompose R into smaller relations that are in 2NF:
  1. Identify all the candidate keys.
  2. Identify the prime and non-prime attributes.
  3. Identify the partial dependencies.
  4. Decompose the relation for the candidate keys and all partial dependencies.
- **E.g. 4.** Given R(A, B, C, D, E) and the FDs
  AB → C
  D → E

  Determine if R is in 2NF, and if it isn't decompose it so that the relations are in 2NF.

  Soln:
  Notice how the RHS of the FDs do not contain A, B and D. Hence, we know that all candidate keys must contain at least A, B and D. The closure of ABD is {A, B, C, D, E}, so in this case, the candidate key is {A, B, D}.

  Since the candidate key is {A, B, D}, the prime attributes are A, B and D and the non-prime attributes are C and E.

  We see that we have 2 partial dependencies:
  1. AB → C
  2. D → E

  I will decompose R into:
  R1(A, B, D) ← Relation with the prime attributes
  R2(A, B, C)
  R3(D, E)

Third Normal Form (3NF):

- A table is in 3NF if:
    1. It is in 2NF and
    2. It must not have any **transitive dependencies**.
- If you have a NP → NP, you have a transitive dependency.
- Here are the steps on how you can decompose R into smaller relations that are in 3NF:
    1. Identify all the candidate keys.
    2. Identify the prime and non-prime attributes.
    3. Identify the partial dependencies and the transitive dependencies.
    4. Decompose the relation for the candidate keys, all partial dependencies and all transitive dependencies.
- **E.g. 5.** Given R(A, B, C, D, E, F, G, H, I, J) and the FDs
  AB → C
  AD → GH
  BD → EF
  A → I
  H → J

  Determine if R is in 3NF, and if it isn't decompose it so that the relations are in 3NF.

  Soln:
  The candidate key is {A, B, D}.
  Hence, the prime attributes are A, B and D.
  The non-prime attributes are C, E, F, G, H, I, J.
  We see that
    - AB → C is a pd
    - AD → GH is a pd
    - BD → EF is a pd
    - A → I is a pd
    - H → J is a td.

  We can decompose R into:
  R1(A, B, C)
  R2(A, D, G, H)
  R3(B, D, E, F)
  R4(A, I)
  R5(H, J)
  R6(A, B, D)
- **Note:** If you have (P+NP) → NP, that is also a 3NF violation.
  E.g. If we have the FD AH → J, we know that A is a prime attribute, but H is a non-prime attribute and J is also a non-prime attribute. This would still be a transitive dependency and would violate 3NF.

Boyce-Codd Normal Form (BCNF/3.5NF):

- A table is in BCNF if:
    1. It is in 3NF and
    2. For every non-trivial FD X → Y, X must be a super key.
- **E.g. 6.** Suppose we have R(A, B, C, D, E, F) and the FDs
  A → B
  CD → E
  AC → F

Is R in BCNF?

Soln:
Let's look at A → B. The closure of A is {A, B}. Hence, A is not a super key, which means that R is not in BCNF.

- For every X → Y that violates BCNF, we create 2 tables
  R1(R - Y)
  R2(X, Y)

More Examples:

E.g. 7. Create an instance of R(A, B, C, D, E) that violates the FD ABC → DE.

Soln:

A	B	C	D	E
1	2	3	4	5
1	2	3	5	7

E.g. 8. Suppose we have a relation R(A, B, C, D, E). Does the instance below violate the FD DB → A?

A	B	C	D	E
5	3	2	1	6
5	8	3	1	2

Soln:
No, because A has the same values for all rows. It doesn't matter what DB is, it will always uniquely determine A.

E.g. 9. Suppose we have R(A, B, C, D, E) and the FDs
A → BD
D → E

Is R in BCNF?

Soln:
No. Let's look at the closure of D. It is {D, E}. Hence, D is not a super key, which means that R is not in BCNF.

E.g. 10. Given R(A, B, C, D) and the FDs
A → B
AC → D
Is R in BCNF? If it's not, decompose it so that the relations are in BCNF.

Soln:
We see that R is not in BCNF because in A → B, A is not a super key.
We will decompose R into
R1(A, C, D)
R2(A, B)

Summary of Normal Forms:

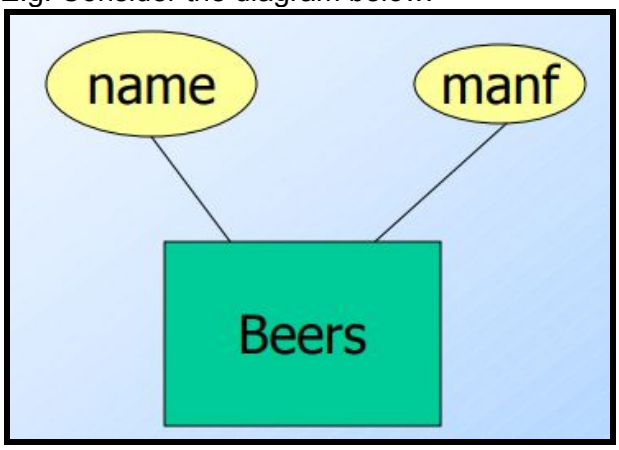
Form	Explanation	Decomposition
1NF (First Normal Form)	Each cell can only contain 1 value.	If a cell contains multiple values, create a new row for each value.
2NF (Second Normal Form)	Must be in 1NF. Cannot have partial dependencies (pds).	Identify all the candidate keys. Identify the prime and non-prime attributes. Identify the partial dependencies. Decompose the relation for the candidate keys and all pds.
3NF (Third Normal Form)	Must be in 2NF. Cannot have transitive dependencies (tds).	Identify all the candidate keys. Identify the prime and non-prime attributes. Identify the partial dependencies and the transitive dependencies. Decompose the relation for the candidate keys, all pds and all tds.
BCNF (Boyce-Codd Normal Form)	Must be in 3NF. For each non-trivial FD $X \rightarrow Y$, X must be a super key.	decompose (R, $X \rightarrow Y$): R1 (R - Y) R2 (X + Y) project FDs onto R1 and R2 recursively call decompose on R1 and R2 for BCNF violations OR decompose (R, $X \rightarrow Y$): R1 (X+) R2 (R - (X+ - X)) project FDs onto R1 and R2 recursively call decompose on R1 and R2 for BCNF violations
4NF (Fourth Normal Form)	Must be in BCNF. Cannot have multi-valued dependencies (mvds).	decompose (R, X -->> Y): R1 = XY R2 = X union (R - Y) Repeat on R1 and R2 until all relations are in 4NF

Introduction to Entity-Relationship Models:

- An **ER model** is a high-level data model. This model is used to define the data elements and relationship for a specified system.
- An ER model describes the structure of a database with the help of an **entity-relationship diagram (ER diagram)**. An ER model is a design or blueprint of a database that can later be implemented as a database. The main components of a ER model are the **entity set** and **relationship set**.
- The purpose of an ER model is that it allows us to sketch database schema designs, called ER diagrams.
  Note: ER models may include some constraints, but not operations.
- ER models are very useful in planning and communicating database schemas. Sketching the key components is an efficient way to develop a working database.
- Later, we can convert ER models to relational database designs.

Introduction to Entities:

- **Entity:** A real-world thing which can be distinctly identified. It is an object which is distinguishable from others. In a table, an entity is a tuple.
  E.g. If we have the table Student(SID, First_Name, Last_Name) then each student in that table is an entity and can be uniquely identified by their SID.
- **Entity Set:** A collection of similar entities.
  The current value of an entity set is the set of entities that belong to it.
  In the ER diagram, an entity set is represented as a rectangle.
- **Attribute:** A property of the entities of an entity set. In a table, an attribute is a column. Note that attributes are simple values such integers or character strings. They are not structs, sets, etc.
  In the ER diagram, an attribute is represented as an oval, with a line to the rectangle representing its entity set.
- E.g. Consider the diagram below:



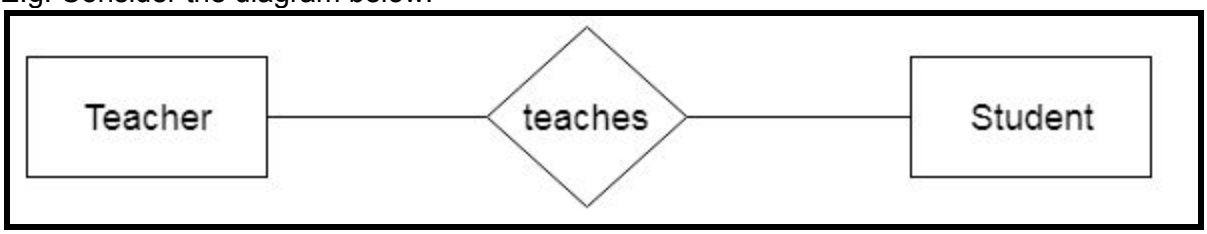
We see that:

- Beers is an entity set, because it's denoted by a rectangle.
- name and manf are attributes of Beers because they are denoted by an oval and they are connected to Beers by a line.

Each Beers entity has values for these two attributes.

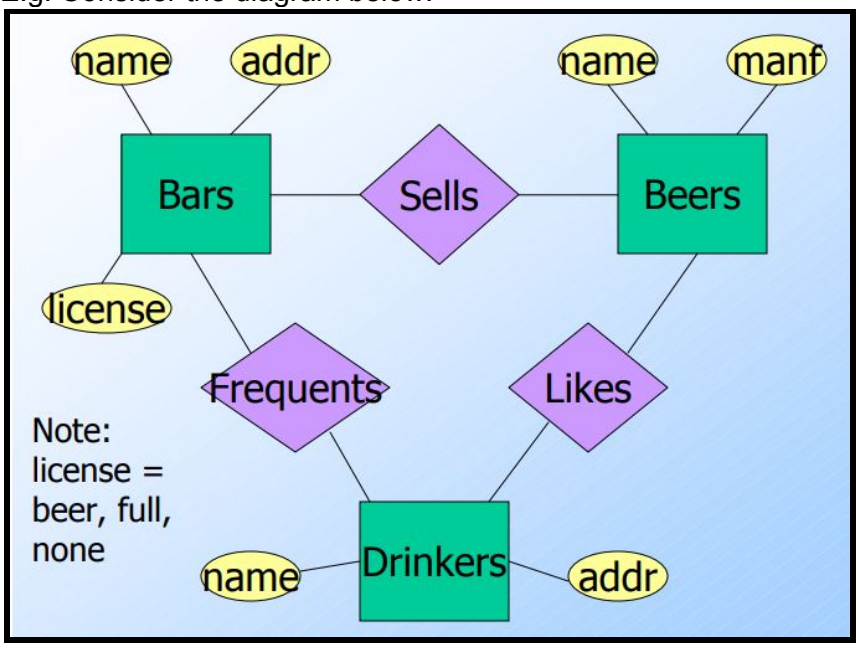
Introduction to Relationships:

- A **relationship** connects two or more entity sets. It is represented by a diamond, with lines to each of the entity sets involved.
- A relationship between two entities signifies that the two entities are associated with each other. Think of relationships as joins.
- E.g. Consider the diagram below:



We can see that the Teacher and Student entity sets are connected by the relationship teaches.

- E.g. Consider the diagram below:



We can see that:

- Bars sell some beers.
- Drinkers like some beers.
- Drinkers frequent some bars.

- The value of a relationship is a **relationship set**, which is a set of tuples with one component for each related entity set.

- E.g. Consider the Sells relationship from above. The below picture is a possible relationship set.

Bar	Beer
Joe's Bar	Bud
Joe's Bar	Miller
Sue's Bar	Bud
Sue's Bar	Pete's Ale
Sue's Bar	Bud Lite

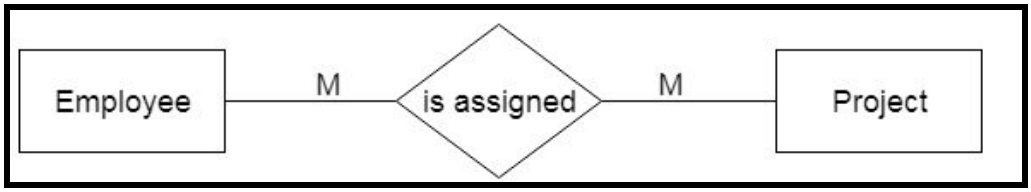
- The **degree of a relationship set** is the number of different entity sets participating in a relationship set.
    - When there is only one entity set participating in a relation, the relationship is called a **unary relationship**.
    - When there are two entities set participating in a relation, the relationship is called a **binary relationship**.
    - When there are n entities set participating in a relation, the relationship is called a **n-ary relationship**.

Cardinality of Relationships:

- **Cardinality** specifies how many instances of an entity relate to one instance of another entity.
  There are 4 types of cardinality:
    1. Many-to-Many
    2. Many-to-One
    3. One-to-One
    4. One-to-Many
- In a **many-many relationship**, an entity of either set can be connected to many entities of the other set.
  I.e. A many-to-many relationship refers to the relationship between two entities X and Y in which X may be linked to many instances of Y and vice versa.

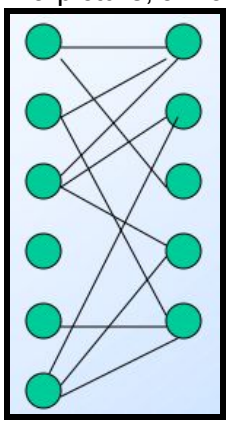
  E.g. Sells is a many-to-many relationship because a bar sells many beers and a beer is sold by many bars.

  E.g. Consider the diagram below:



Is assigned is a many-to-many relationship because an employee can be assigned many projects and a project can have many employees working on it.

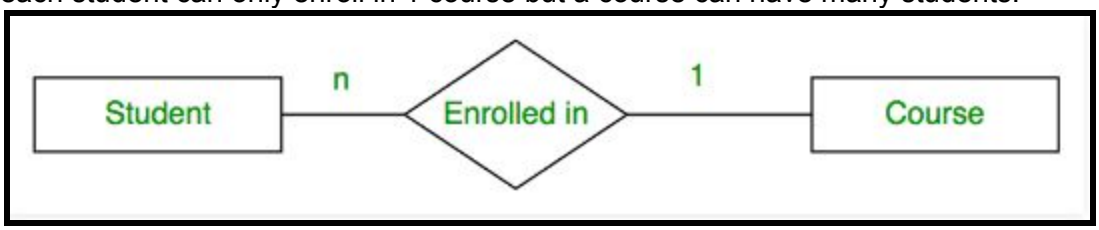
In a picture, a many-to-many relationship looks like:



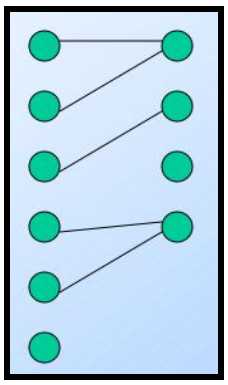
- A **many-to-one relationship** occurs when more than one instance of the entity on the left and only one instance of an entity on the right associates with the relationship.
  Note: In a many-to-one relationship, each entity of the first set is connected to at most one entity of the second set but an entity of the second set can be connected to zero, one, or many entities of the first set.

E.g. Suppose there is a Favourites relationship between entity sets Drinkers and Beers. Favourites is a many-to-one relationship because a drinker can only have 1 favourite beer but a beer can be the favourite of many drinkers.

E.g. Assume that at UTSC, each student can enroll in 1 course at most.
Then, the Enrolled in relationship shown below is a many-to-one relationship because each student can only enroll in 1 course but a course can have many students.



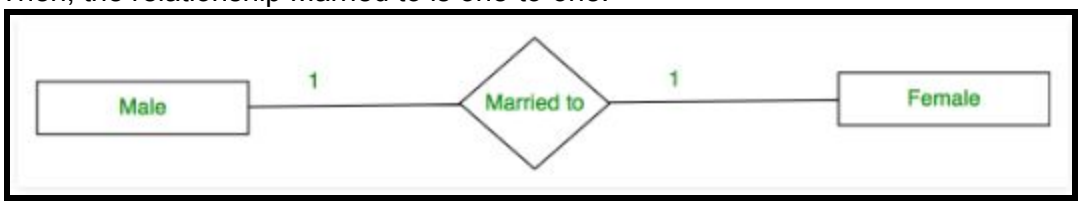
In a picture, a many-to-one relationship looks like:



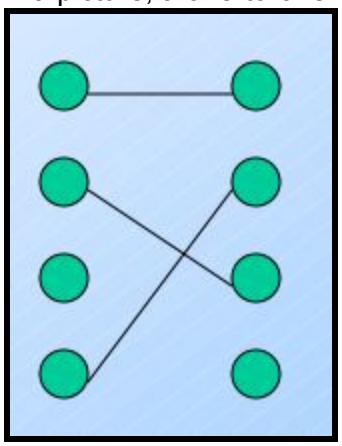
- A **one-to-one relationship** occurs when each entity in each entity set can take part only once in the relationship.
  I.e. In a one-one relationship, each entity of either entity set is related to at most one entity of the other set.

  E.g. Suppose we have a relationship Best-Seller between Manufacturer and Beers. Best-Seller is a one-to-one relationship because a manufacturer can only have 1 best selling beer (assume no ties) and a beer can only be made by 1 manufacturer.

  E.g. Assume that a male can marry to one female and a female can marry to one male. Then, the relationship Married to is one-to-one.

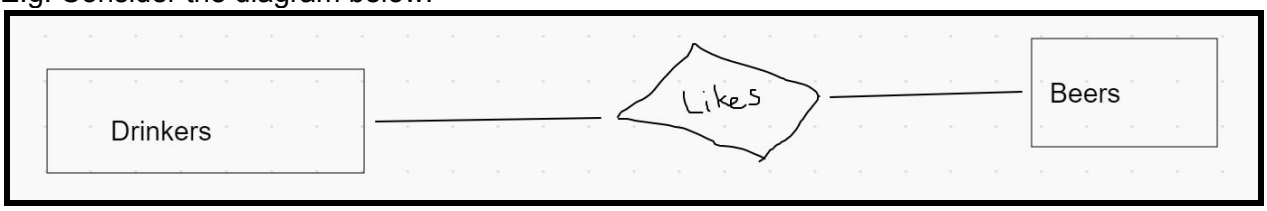


  In a picture, a one-to-one relationship looks like:



- We can represent these relationships in the ER diagram by using various lines.
  - We can show a many-to-one relationship by an arrow entering the “one” side.
  - We can show a one-to-one relationship by arrows entering both entity sets.
  - A rounded arrow means exactly one.
    I.e. Each entity of the first set is related to exactly one entity of the target set.

  E.g. Consider the diagram below.



  This is a many-to-many diagram. We can tell because there’s no arrow going into either Drinkers or Beers.

To interpret it, start from one entity set and follow the line to the second entity set. Then, start from the second entity set and follow the line to the first entity set.

We can interpret this as:

- A drinker likes some beers.
  (We started from drinker and followed the arrow to beers.)
- A beer is liked by some drinkers.
  (We started from beer and followed the arrow to drinker.)

E.g. Consider the diagram below.

This is a many-to-one diagram. We can tell because there's an arrow going into Beers while there's no arrow going into Drinkers.

We can interpret this as:

1. A drinker has at most 1 favourite beer.
2. A beer is the favourite of some drinkers.

Note: In the first 2 examples, two relationships are connecting the same entity sets, but are different.

E.g. Consider the diagram below.

Note: The arrow going to Beers is a rounded arrow, meaning exactly one.

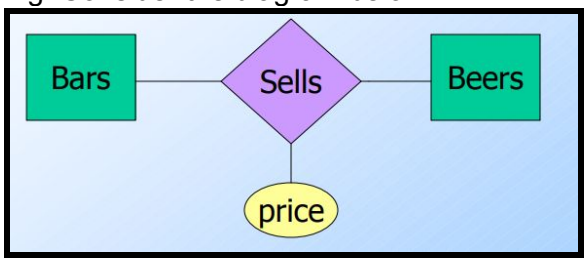
We can interpret this as:

1. A manufacturer has exactly one best-selling beer.
2. A beer is the best seller of at most one manufacturer.

Attributes on Relationships:

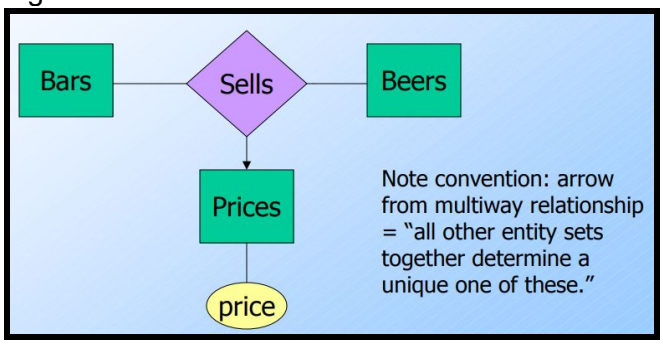
- Sometimes it is useful to attach an attribute to a relationship.
- We can think of this attribute as a property of tuples in the relationship set.

- E.g. Consider the diagram below.



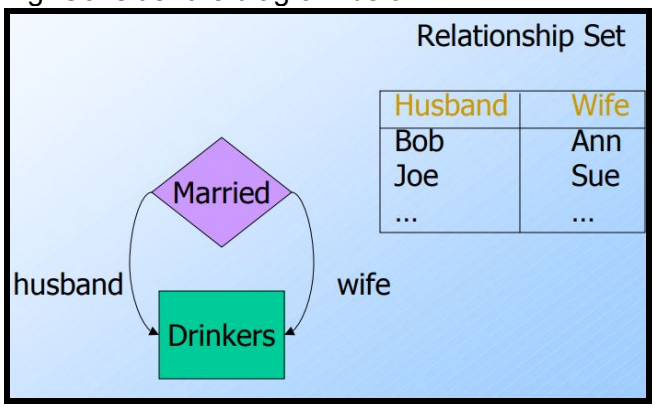
Price is a function of both the bar and the beer, not of one alone.

- An equivalent way of showing the diagram but without having attributes on relationships is to create an entity set representing values of the attribute and make that entity set participate in the relationship.
- E.g.



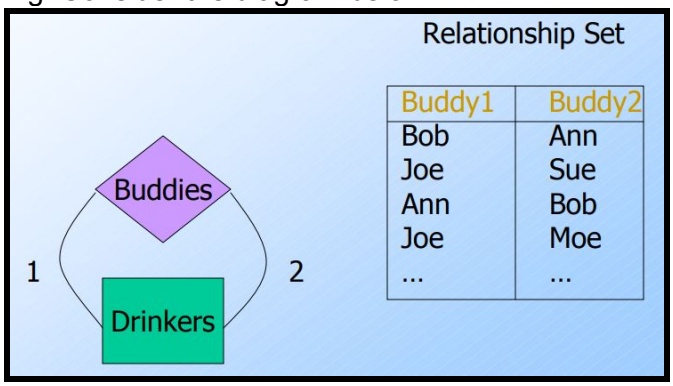
Roles:

- Entity sets of a relationship do not need to be distinct and sometimes an entity set appears more than once in a relationship.
- To show this on an ER diagram, we label the edges between the relationship and the entity set with names called **roles**.
- Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.
- Role labels are optional, and are used to clarify semantics of the relationship.
- E.g. Consider the diagram below:



In this example, husband and wife are the roles.

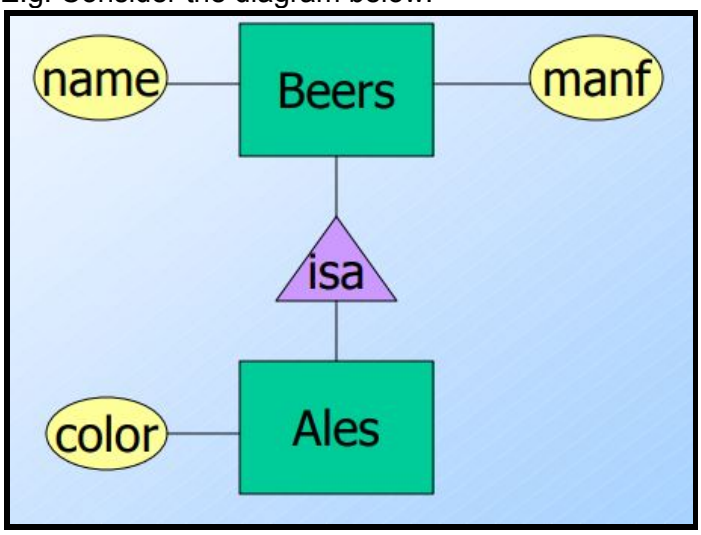
- E.g. Consider the diagram below.



The roles in this example are buddy1 and buddy2.

Subclasses:

- An entity set may contain entities that have special properties not associated with all members of the set. These entities are called **subclasses**.
  I.e. A subclass is a subgroup of entities with special properties.
- E.g. Ales are a kind of beer. Not every beer is an ale, but some are.
- Assume subclasses form a tree in the ER diagram.
  I.e. There's no multiple inheritance.
- Isa triangles indicate the subclass relationship. They always point to the superclass.
- E.g. Consider the diagram below.

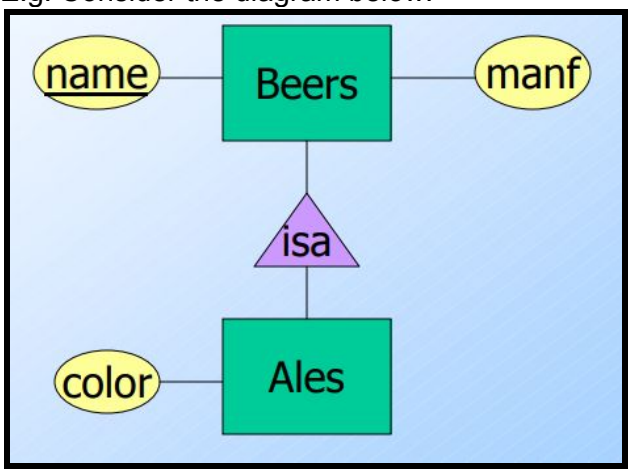


- ER entities have representatives in all subclasses to which they belong.
  Rule: If entity e is represented in a subclass, then e is represented in the superclass and recursively up the tree.

Keys:

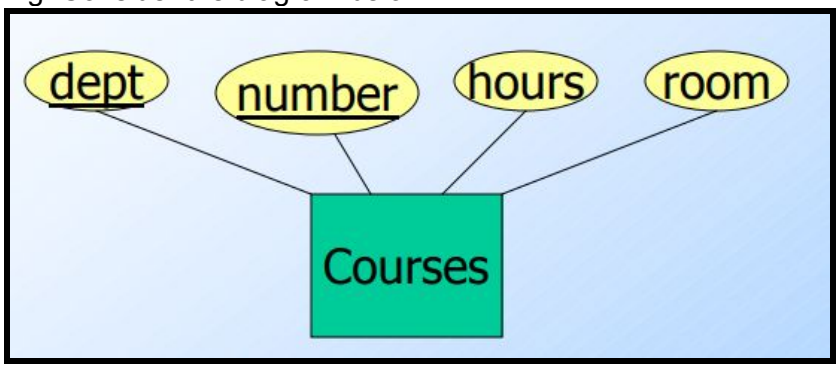
- A **key** is a set of attributes for one entity set such that no two entities in this set agree on all the attributes of the key.
- **Note:** It is allowed for two entities to agree on some, but not all, of the key attributes.
- We must designate a key for every entity set.

- There could be multiple keys, but we choose one to use.
- To represent a key in an ER diagram, we underline the attribute(s) that make up the key.
- In an Isa hierarchy, only the root entity set has a key, and it must serve as the key for all entities in the hierarchy.
- E.g. Consider the diagram below.



In this case, name is key for Beers and for Ales because Ales is a subclass of Beers.

- E.g. Consider the diagram below.

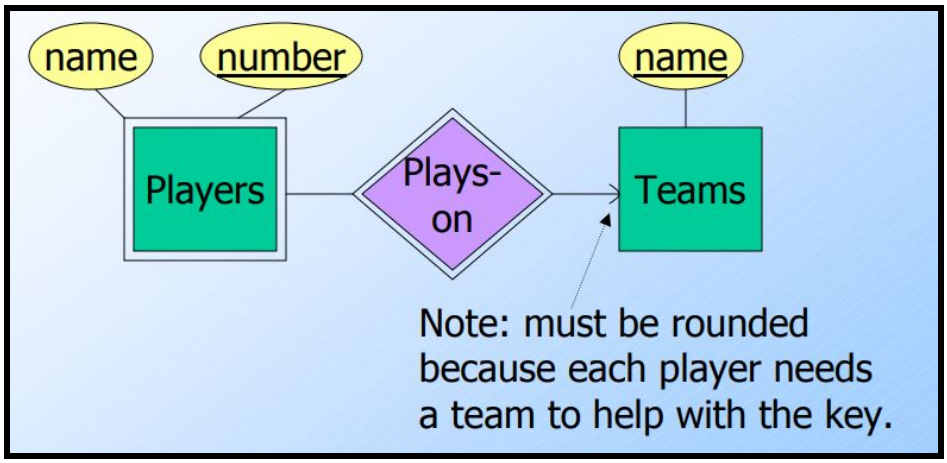


Here, we see that dept and number form the key.
Note that hours and room could also serve as a key, but we must select only one key.

Weak Entity Set:

- A **weak entity** is an entity that depends on another entity.
- Entity set E is said to be weak if in order to identify entities of E uniquely, we need to follow one or more many-one relationships from E and include the key of the related entities from the connected entity sets.
- An entity set that does not have a primary key is referred to as a weak entity set.
  I.e. A weak entity set does not have a primary key.
- The weak entity is represented by a double rectangle in an ER diagram.
- The relationships connecting a weak entity set to a strong entity set is represented by a double diamond in an ER diagram.
- E.g.
  name is almost a key for football players, but there might be two with the same name.
  number is certainly not a key, since players on two teams could have the same number.

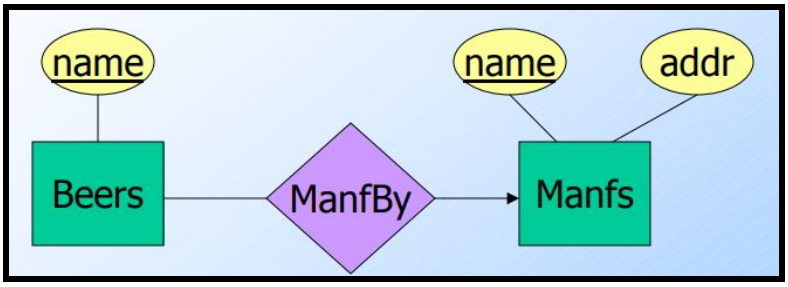
But number, together with the team name related to the player by Plays-on should be unique.
I.e.



- Weak entity set rules:
    1. A weak entity set has one or more many-one relationships to other supporting entity sets.
        - **Note:** Not every many-one relationship from a weak entity set needs to be supporting.
        - All supporting relationships must have a rounded arrow pointing towards its “one” end.
    2. The key for a weak entity set is its own underlined attributes and the keys for the supporting entity sets.

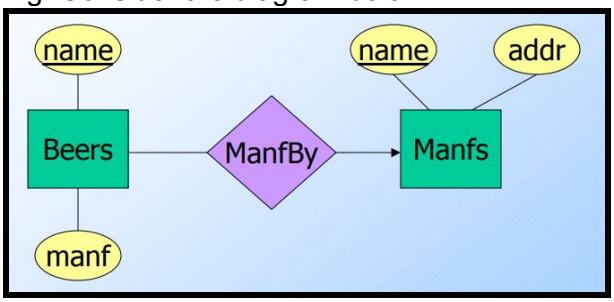
Design Techniques:

- **Avoid redundancy:**
    - Redundancy is saying the same thing in two or more different ways.
    - It wastes space and more importantly encourages inconsistency.
    - Two representations of the same fact become inconsistent if we change one and forget to change the other.
      Recall the anomalies from function dependencies.
    - E.g. Consider the diagram below.



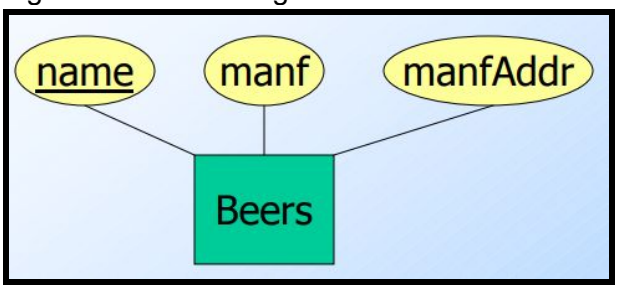
This is an example of good design because there is no duplicate information.

- E.g. Consider the diagram below.



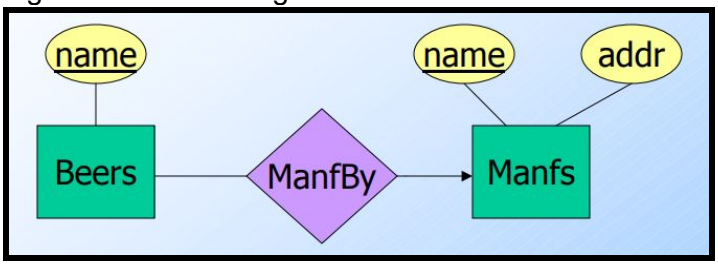
This is an example of bad design because the manf information is duplicated. This design states the manufacturer of a beer twice: once as an attribute and second as a related entity.

- E.g. Consider the diagram below.



This is an example of bad design because this design repeats the manufacturer's address once for each beer. Suppose that a manufacturer temporarily stopped producing beers. We would lose its address.

- **Limit the use of weak entity sets:**
  - We use weak entity sets if there is no global authority capable of creating unique IDs.
  - E.g. It is unlikely that there could be an agreement to assign unique player numbers across all football teams in the world.
- **Don't use an entity set when an attribute will do:**
  - An entity set should satisfy at least one of the following conditions:
    1. It is more than the name of something. It has at least one non-key attribute.
    2. It is the "many" in a many-one or many-many relationship.
  - E.g. Consider the diagram below.

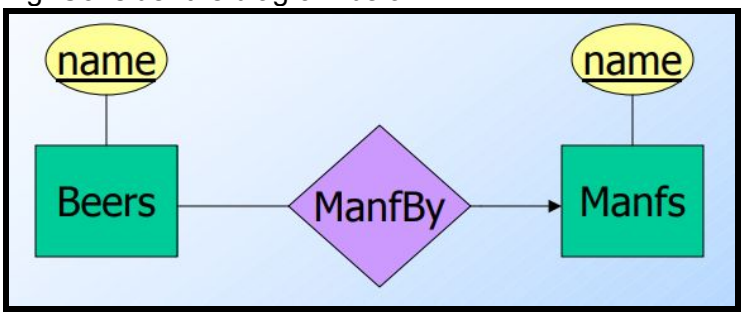


This is an example of good design because:

1. Manfs deserves to be an entity set because of the nonkey attribute addr.

2. Beers deserves to be an entity set because it is the “many” of the many-one relationship ManfBy.

- E.g. Consider the diagram below.



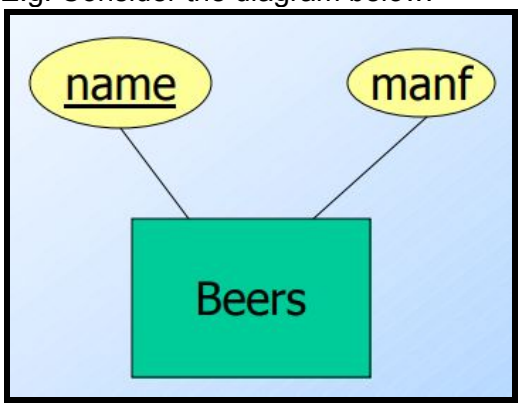
This design is bad because:

1. Manfs is just a name.
2. Manfs is not at the “many” end.

Hence, Manfs should not be an entity set.

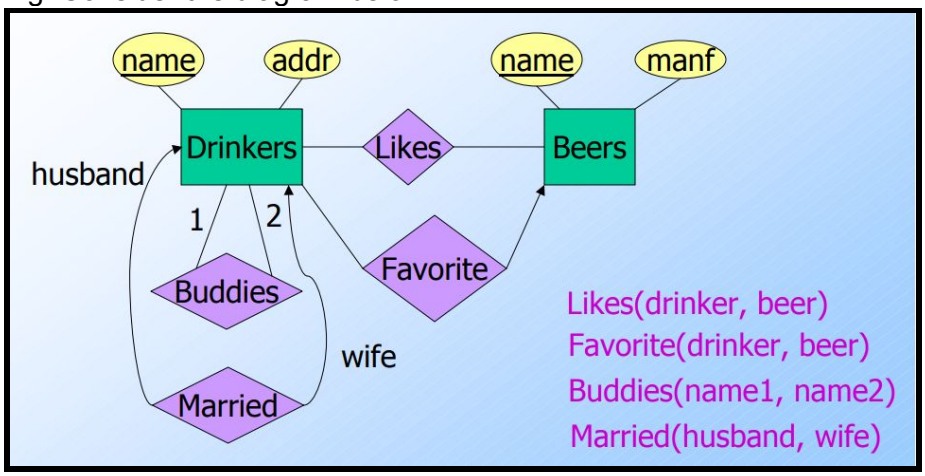
ER Diagrams to Relations:

- Entity set → relation.
- Attributes → attributes.
- Relationships → relations whose attributes are only:
    1. The keys of the connected entity sets. Or
    2. Attributes of the relationship itself.
- E.g. Consider the diagram below.



The corresponding relation is Beers(name, manf).

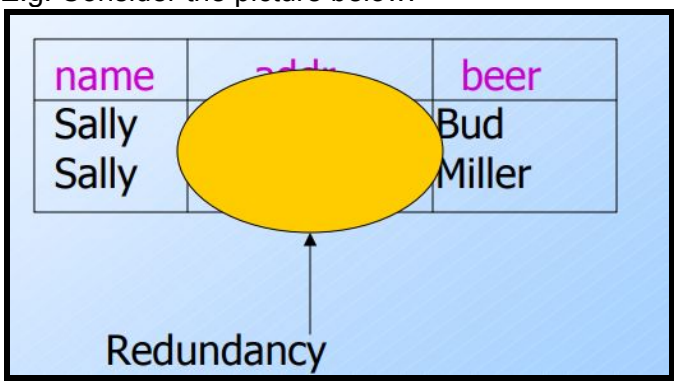
- E.g. Consider the diagram below.



The relations we get from the picture are:

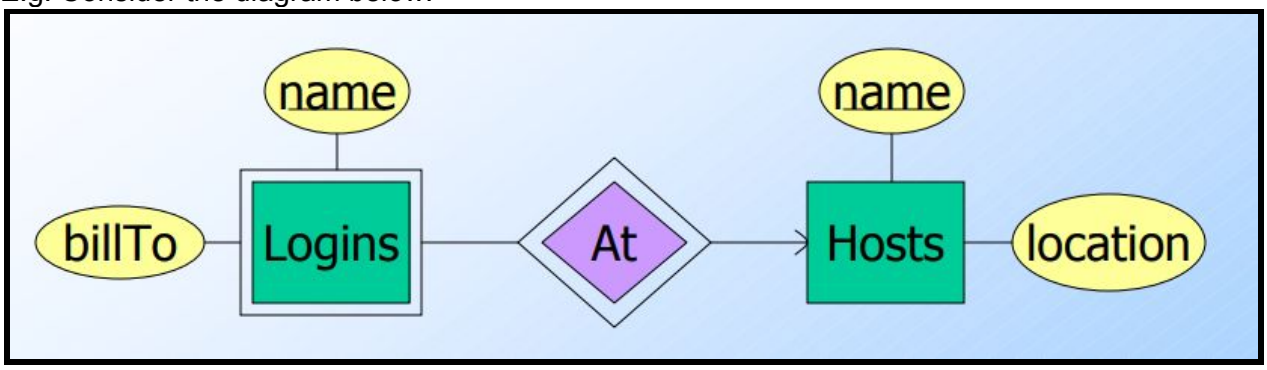
1. Drinkers(name, addr)
2. Beers(name, manf)
3. Married(husband, wife)
4. Buddies(name1, name2)
5. Likes(drinker, beer)
6. Favorite(drinker, beer)

- We can also combine relations. It is ok to combine the following 2 relations into one relation:
    1. The relation for an entity-set E
    2. The relations for many-one relationships of which E is the "many."
- E.g.
  Drinkers(name, addr) and Favorite(drinker, beer) combine to make Drinker1(name, addr, favBeer).
- **Note:** The reason why we don't combine many-to-many relationships is because it could lead to redundancy.
- E.g. Consider the picture below.



We see that Sally's address is used twice.

- We can turn a weak entity set into a relation too.
- A relation for a weak entity set must include attributes for its complete key including those belonging to other entity sets, as well as its own, nonkey attributes.
- A supporting relationship is redundant and yields no relation unless it has attributes.
- E.g. Consider the diagram below.



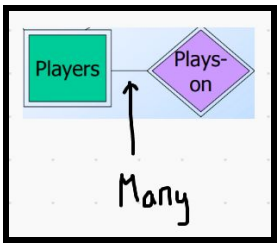
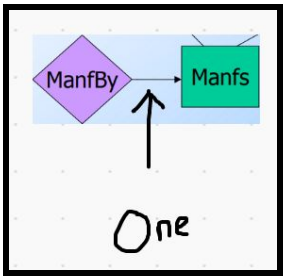
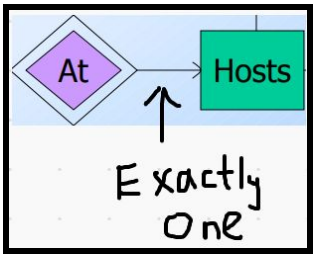
We have 2 relations:

1. Hosts(<u>hostName</u>, location)
2. Logins(<u>loginName</u>, <u>hostName</u>, billTo)

We don't need a relation for At as it does not have any attributes.

<u>Summary of ER Diagram Items:</u>

Item	**Definition**	**Shape**	**Example**
Entity Set	A collection of similar entities.	Rectangle	
Weak Entity Set	Entity set E is said to be weak if in order to identify entities of E uniquely, we need to follow one or more many-one relationships from E and include the key of the related entities from the connected entity sets. An entity set that does not have a primary key is referred to as a weak entity set. I.e. A weak entity set does not have a primary key.	Double Rectangle	
Attribute	A property of the entities of an entity set. In a table, an attribute is a column. Note that attributes are simple values such integers or character strings. They are not structs, sets, etc. **Note:** You underline the attribute(s) that form the key.	Oval	

Relationship	A relationship connects two or more entity sets. A relationship between two entities signifies that the two entities are associated with each other. Think of relationships as joins.	Diamond	
Supporting Relationship	The relationship connecting a weak entity set to a strong entity set.	Double Diamond	
Isa Triangle	Isa triangles indicate the subclass relationship. They always point to the superclass.	Triangle	
Many	0 or more	Line with no arrows	
One	At most 1	Line with filled in arrow.	
Exactly One	Exactly 1	Line with rounded arrow	

XML:

- XML stands for eXtensible Markup Language.
- **Note:** XML is not a database.
- XML is useful for moving data between databases.
- XML was designed to store and transport data.
- XML was designed to be both human and machine-readable.
- **XPath** is a language for navigating in XML documents.
- **XQuery** is a language for querying XML documents.
- **XSLT** is a language for transforming XML documents.

Well-Formed XML:

- An XML document with the correct syntax is called **Well-Formed**.
- An XML document validated against a DTD is both Well-Formed and valid.
- **DTD** stands for **Document Type Definition**.
- A DTD defines the structure and the legal elements and attributes of an XML document.
- **Well-Formed XML** allows you to invent your own tags.
- Valid XML conforms to a certain DTD.
  You can think of DTD as a schema for your XML file.
- Start the document with a declaration, surrounded by **<?xml … ?>**.
- A normal declaration is: **<?xml version = ”1.0” standalone = “yes” ?>**.
  Note that standalone = “yes” means no DTD is provided.
  standalone = “no” means that a DTD is provided.
- The balance of document is a root tag surrounding nested tags.

XML Tags:

- Tags are normally matched pairs such as **<FOO> … </FOO>**.
- Unmatched tags are also allowed such as **<FOO/>**.
- Tags may be nested arbitrarily.
  E.g.

```
<Student>
        <Name> … </Name>
        <Student_ID> … </Student_ID>
</Student>
```

- XML tags are case-sensitive.

DTD Structure:

- The purpose of a DTD is to define the structure and the legal elements and attributes of an XML document.
- The structure of a DTD looks like this:

```
<!DOCTYPE <root tag> [
   <!ELEMENT <name> (<components>)>
   More elements
]>
```

- E.g.

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

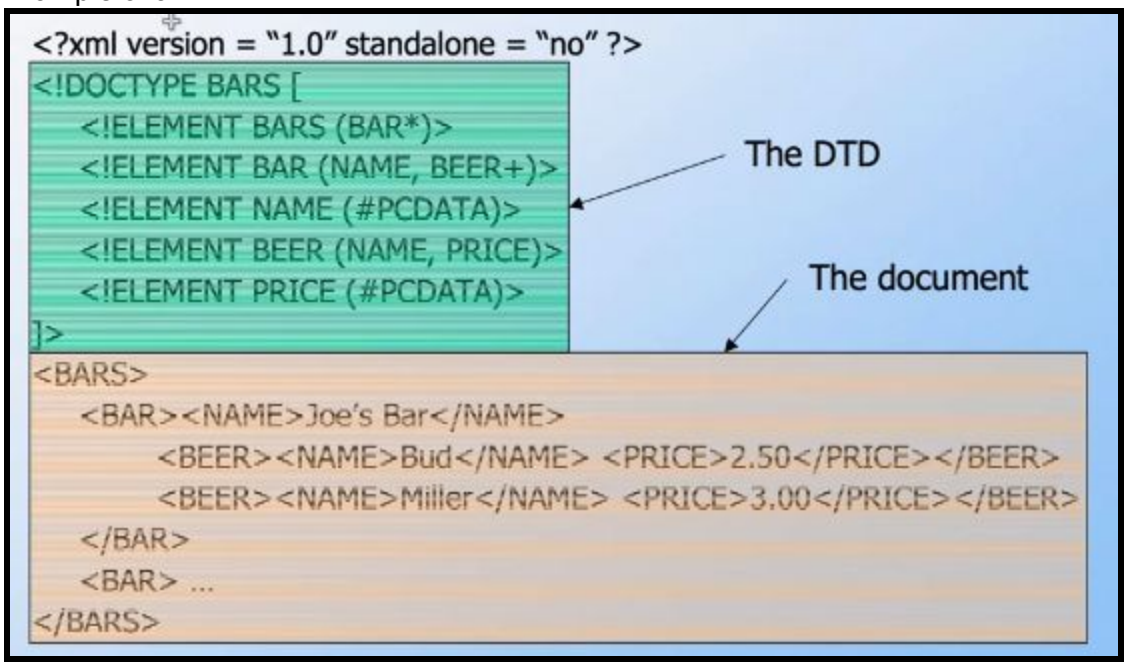
- The description of an DTD element consists of its name (tag), and a parenthesized description of any nested tags. It also includes the order of subtags and their multiplicity.

- Leaves (text elements) have #PCDATA (Parsed Character DATA) in place of nested tags.
- Subtags must appear in order shown.
- A tag may be followed by a symbol to indicate its multiplicity:
  - * = zero or more
  - + = one or more
  - ? = zero or one.
- The symbol | can connect alternative sequences of tags.
- E.g. In the example below, a name is an optional title, a first name, and a last name, in that order, or it is an IP address.

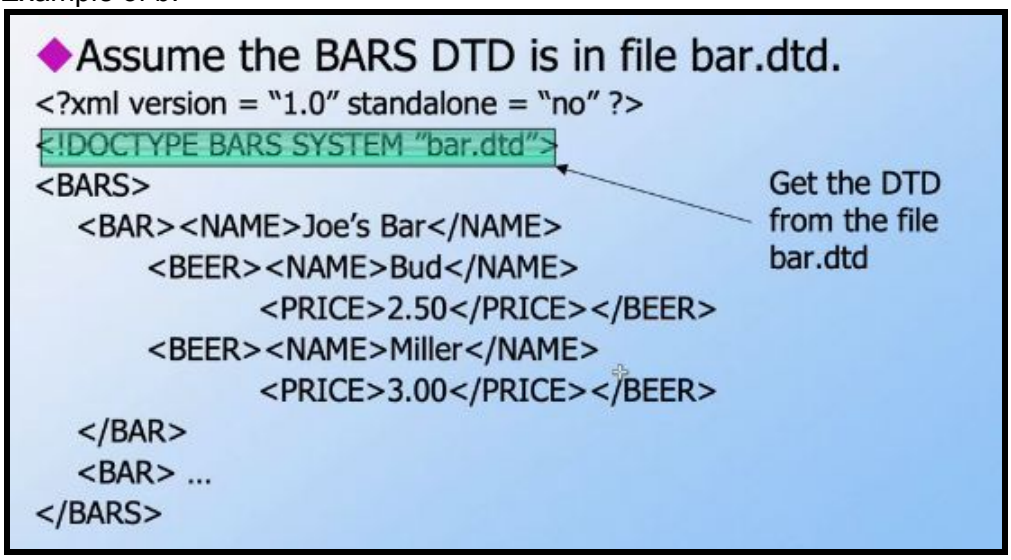
```
<!ELEMENT NAME (
  (TITLE?, FIRST, LAST) | IPADDR
)>
```

It is an optional title because we have a ? after TITLE.
It could just be an IP address because we have a |.

- If you want to use a DTD, you have to set standalone = “no”.
- We can either:
  a. Include the DTD as a preamble of the XML document.
     I.e. The DTD is included in the XML document.
     OR
  b. Follow DOCTYPE and the by SYSTEM and a path to the file where the DTD can be found.
     The DTD is a separate document but there’s a path to it in the XML document.
- Example of a:



- Example of b:



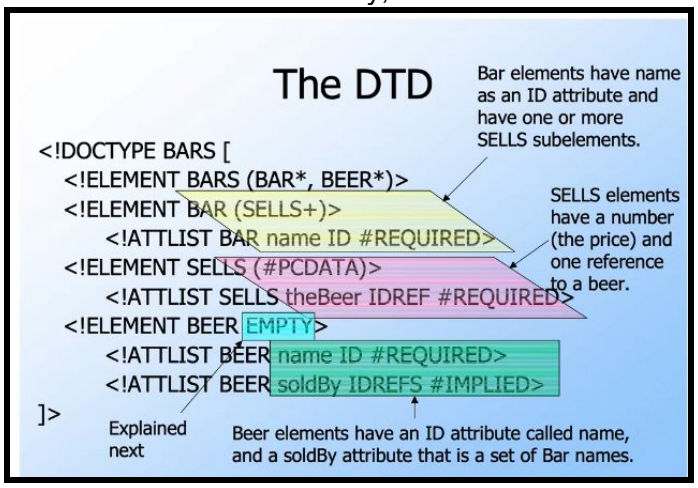
- An attribute declaration in DTD has the following syntax:
  <!ATTLIST element-name attribute-name attribute-type attribute-value>.
- E.g.
  <!ATTLIST payment type CDATA "check">
  XML example:
  <payment type="check"/>
- Table of attribute-type:

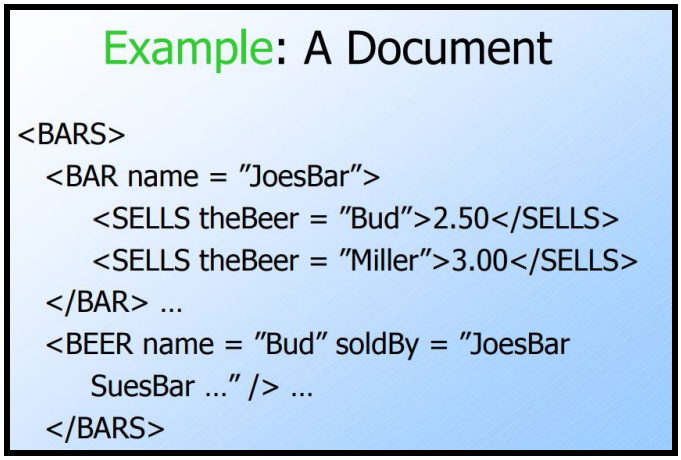
Type	**Description**
CDATA	The value is character data
(en1|en2|..)	The value must be one from an enumerated list
ID	The value is a unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
xml:	The value is a predefined xml value

- Table of attribute-value:

Value	**Explanation**
value	The default value of the attribute
#REQUIRED	The attribute is required
#IMPLIED	The attribute is optional
#FIXED value	The attribute value is fixed

- Attributes can be pointers from one object to another.
- IDs and IDREFs allow the structure of an XML document to be a general graph, rather than just a tree.
- E.g.
  A new BARS DTD includes both BAR and BEER subelements.
  BARS and BEERS have ID attributes name.
  BARS have SELLS subelements, consisting of a number (the price of one beer) and an IDREF theBeer leading to that beer.
  BEERS have attribute soldBy, which is an IDREFS leading to all the bars that sell it.



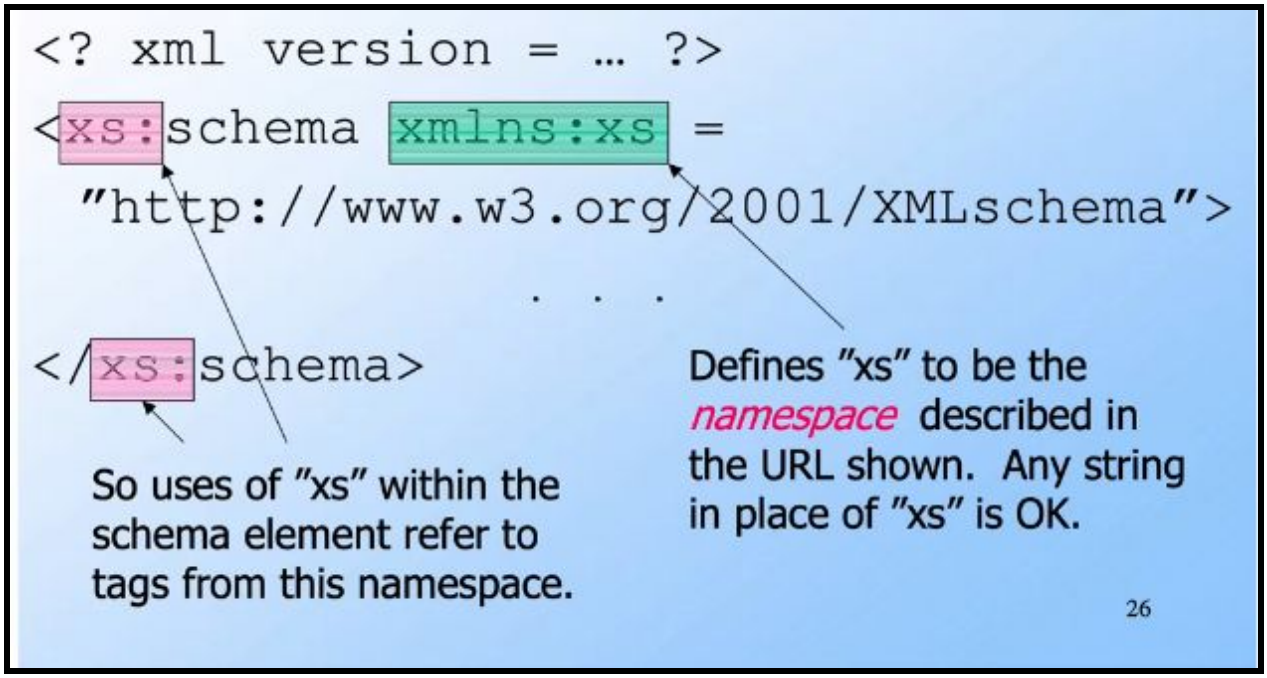


- Empty elements are declared with the category keyword EMPTY.

- E.g.
  <!ELEMENT element-name EMPTY>
- With empty elements, we can do all the work of an element in its attributes.

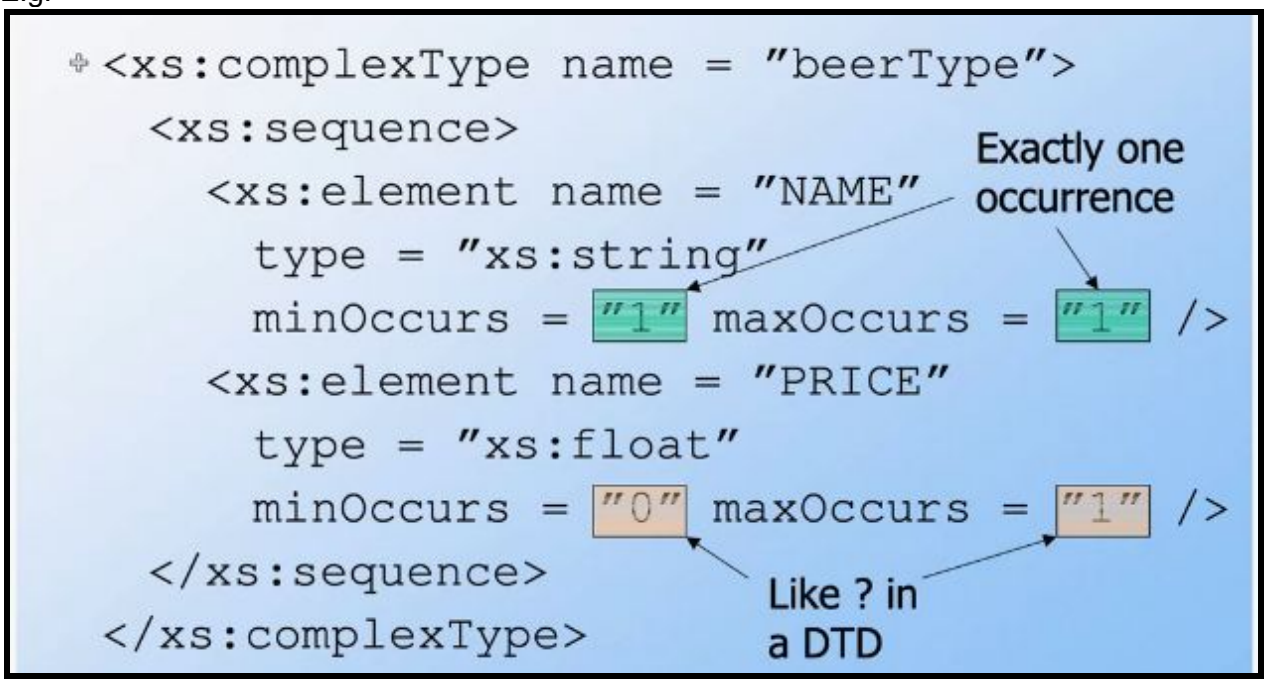
XML Schema:

- A more powerful way to describe the structure of XML documents.
- XML-Schema declarations are themselves XML documents.
  They describe “elements” and the things doing the describing are also “elements.”
- Here is the structure of an XML schema:



- A **simple element** is an XML element that can contain only text. The text can be of many different types, such as string, decimal, integer, boolean, etc.
  The syntax for defining a simple element is:
  <xs:element name="xxx" type="yyy"/>
  where xxx is the name of the element and yyy is the data type of the element.
  The data type could be the name of a type defined in the document itself.
  XML Schema has a lot of built-in data types. The most common types are:
  - xs:string
  - xs:decimal
  - xs:integer
  - xs:boolean
  - xs:date
  - xs:time
- E.g.
  <xs:element name = ”NAME” type = ”xs:string” />
- To describe elements that consist of subelements, we use xs:complexType.
- The complexType element defines a complex type. A complex type element is an XML element that contains other elements and/or attributes.
- The attribute name gives a name to the type.
- A typical subelement of a complex type is xs:sequence, which itself has a sequence of xs:element subelements.
- We can use the minOccurs and maxOccurs attributes to control the number of occurrences of an xs:element.

- E.g.



- xs:attribute elements can be used within a complex type to indicate attributes of elements of that type.
- attributes of xs:attribute:
  - name and type as for xs:element.
  - use = ”required” or ”optional”.
- E.g.
  <xs:attribute name="lang" type="xs:string" use="required"/>
- E.g.

```
<xs:complexType name = "beerType">
  <xs:attribute name = "name"
     type = "xs:string"
     use = "required" />
  <xs:attribute name = "price"
     type = "xs:float"
     use = "optional" />
</xs:complexType>
```

- An xs:element can have an xs:key subelement. This means that within this element, all subelements reached by a certain selector path will have unique values for a certain combination of fields.
- An xs:keyref subelement within an xs:element says that within this element, certain values (defined by selector and field(s), as for keys) must appear as values of a certain key.

XPath:

- In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes.
- XML documents are treated as trees of nodes.
  The topmost element of the tree is called the **root element**.
- E.g. Consider the XML snippet below.

```
<?xml version="1.0" encoding="UTF-8"?>

<bookstore>
  <book>
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

  <bookstore> is the root element node.
  <author>J K. Rowling</author> is an element node.
  lang="en" is an attribute node.

- Table of XPath expressions

Expression	Description
node-name	Select all nodes with the given name "nodename"
/	Selection starts from the root node.
//	Selection starts from the current node that matches the selection.
.	Selects the current node.
..	Selects the parent of the current node.
@	Selects attributes.

- E.g. Consider the XML snippet below.

```
<?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book>
  <title lang="en">Harry Potter</title>
  <price>29.99</price>
</book>
```

```
<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
</book>

</bookstore>
```

The below table shows examples of XPath expressions.

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore".
/bookstore	Selects the root element bookstore. **Note:** If the path starts with a slash (/) it always represents an absolute path to an element.
bookstore/book	Selects all book elements that are children of the bookstore.
//book	Selects all book elements no matter where they are in the document.
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element.
//@lang	Selects all attributes that are named lang.

XQuery:

- XQuery is to XML what SQL is to databases.
- XQuery is designed to query XML data.
- XQuery is an expression language.
  Each expression operates on and returns sequences of elements.
- XQuery is case-sensitive.
- XQuery elements, attributes, and variables must be valid XML names.
- An XQuery string value can be in single or double quotes.
- An XQuery variable is defined with a $ followed by a name. E.g. $bookstore
- XQuery comments are delimited by (: and :). E.g. (: This is a XQuery Comment :)
- XQuery is built on XPath expressions.
- **FLWOR Expression:**
  - Syntax:
    For $var in expr
    Let $var := expr
    Where condition
    Order By expr
    Return Expr
  - For: Selects a sequence of nodes.
  - Let: Binds a sequence to a variable.
  - Where: Filters the nodes.
  - Order by: Sorts the nodes.
  - Return: What to return (gets evaluated once for every node).
  - **Note:** Everything except the Return clause is optional.
  - **Note:** The For and Let clauses can be repeated and interleaved.

- We can mix query results with xml data.
  E.g. <result> {query result} </result>
- **Comparison Operators:**

	Value Comparison	General Comparison
equals	eq	=
not equals	ne	!=
less than	lt	<
greater than	gt	>
less than or equal to	le	<=
greater than or equal to	ge	>=

- **General comparison operators** can be used to compare atomic values, sequences, or any combination of the two.
- When you are comparing two sequences by using general comparison operators and a value exists in the second sequence that compares True to a value in the first sequence, the overall result is True. Otherwise, it is False.
  E.g. (1, 2, 3) = (3, 4) is True, because the value 3 appears in both sequences.
- **Value comparison operators** are used to compare atomic values.
- If the two values compare the same according to the chosen operator, the expression will return True. Otherwise, it will return False. If either value is an empty sequence, the result of the expression is False.

XSLT:

- **XSL (eXtensible Stylesheet Language)** is a styling language for XML.
- XSLT stands for XSL Transformations.
- It is similar to XML as CSS is to HTML.

XSLT Template:

- The <xsl:template> element is used to build templates.
- Syntax:

```
<xsl:template
  name = Qname
  match = Pattern
  priority = number>
...
</xsl:template>
```

- The match attribute is used to associate a template with an XML element.
  The value of the match attribute is an XPath expression.
  Note: match="/" defines the whole document.
- Table of attributes:

Name	Description
Name	The name of the element on which the template is to be applied. If this is present, the match attribute becomes optional.
Match	The match attribute is used to associate a template with an XML element.

Priority	If there are several xsl:template elements that all match the same node, the one that is chosen is determined by the optional priority attribute. The template with highest priority wins. The priority is written as a floating-point number. The default priority is 1. If two matching templates have the same priority, the one that appears last in the stylesheet is used.

XSLT Value-of:

- The <xsl:value-of> element is used to extract the value of a selected node.
- The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output stream of the transformation.
- Syntax:
  <xsl:value-of select = XPath_Expression/>
- Table of attributes:

Name	Description
Select	The XPath expression to be evaluated in the current context. I.e. The select attribute contains an XPath expression.

- E.g.
  <xsl:value-of select="title"/>

XSLT For-Each:

- The <xsl:for-each> element allows you to do looping in XSLT.
- The XSL <xsl:for-each> element can be used to select every XML element of a specified node-set.
- Syntax:
  <xsl:for-each select = XPath_Expression>
  ...
  </xsl:for-each>
- Table of attributes:

Name	Description
Select	The XPath Expression to be evaluated in the current context to determine the set of nodes to be iterated. I.e. The value of the select attribute is an XPath expression.

- E.g.

```
<xsl:for-each select="catalog/cd">
        <xsl:value-of select="title"/>
        <xsl:value-of select="artist"/>
</xsl:for-each>
```

XSLT If:

- The <xsl:if> element is used to put a conditional test against the content of the XML file. I.e. The <xsl:if> tag specifies a conditional test against the content of nodes.
- Syntax:

```
<xsl:if test = boolean-expression>
...
</xsl:if>
```

- Table of attributes:

Name	Description
Test	The condition in the xml data to test. The value of the required test attribute contains the expression to be evaluated.

- E.g.

```
<xsl:if test="price &gt; 10">
        <xsl:value-of select="title"/></td>
        <xsl:value-of select="artist"/></td>
        <xsl:value-of select="price"/></td>
</xsl:if>
```

XSLT Comparison Operators:

Operator	Description
$gt;	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
=	Equals
!=	Not equal