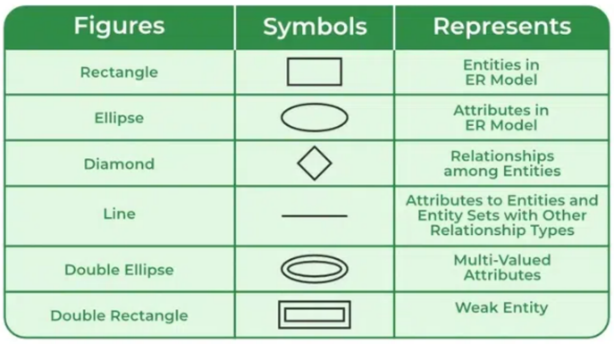
SRE DAY 3 SQL:

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

- **Rectangles:** Rectangles represent Entities in the ER Model.
- **Ellipses:** Ellipses represent Attributes in the ER Model.
- **Diamond:** Diamonds represent Relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double Ellipses represent Multi-Valued Attributes.
- **Double Rectangle:** Double Rectangle represents a Weak Entity.



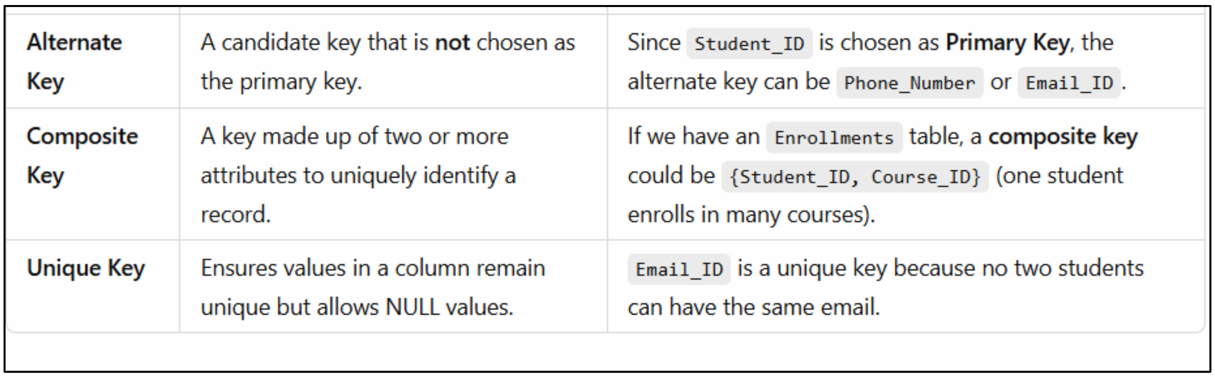
Figures	Symbols	Represents
Rectangle	▭	Entities in ER Model
Ellipse	⬭	Attributes in ER Model
Diamond	◇	Relationships among Entities
Line	—	Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse	◎	Multi-Valued Attributes
Double Rectangle	▣	Weak Entity

Student Table Example

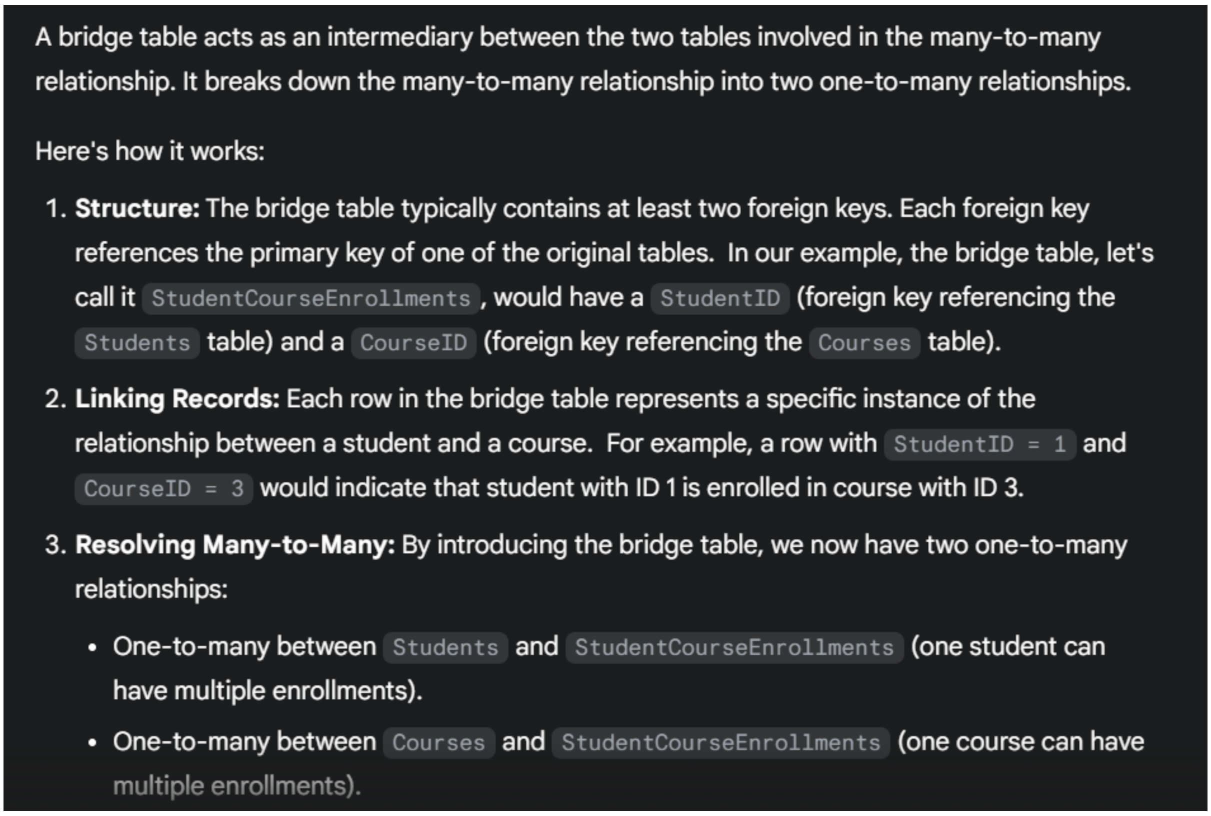
Student_ID (PK)	Name	Email_ID (Unique)	Phone_Number (Candidate)	Department_ID (FK)
101	John	john@email.com	9876543210	10
102	Alice	alice@email.com	9123456789	20
103	Bob	bob@email.com	9234567890	10

Explanation of Different Keys in this Table

Key Type	Definition	Example from Table
Primary Key (PK)	A unique identifier for each student. Cannot be NULL.	`Student_ID` (101, 102, 103) is the **Primary Key**.
Candidate Key	A set of attributes that can uniquely identify a row.	`Student_ID`, `Phone_Number`, and `Email_ID` (all are unique).
Super Key	A set of one or more attributes that uniquely identify a row (includes extra attributes).	`{Student_ID}`, `{Student_ID, Name}`, `{Phone_Number, Email_ID}`.
Foreign Key (FK)	A key that refers to the **Primary Key** of another table, creating a relationship.	`Department_ID` refers to the `Department_ID` in a **Department** table.



Alternate Key	A candidate key that is **not** chosen as the primary key.	Since `Student_ID` is chosen as **Primary Key**, the alternate key can be `Phone_Number` or `Email_ID`.
Composite Key	A key made up of two or more attributes to uniquely identify a record.	If we have an `Enrollments` table, a **composite key** could be `{Student_ID, Course_ID}` (one student enrolls in many courses).
Unique Key	Ensures values in a column remain unique but allows NULL values.	`Email_ID` is a unique key because no two students can have the same email.

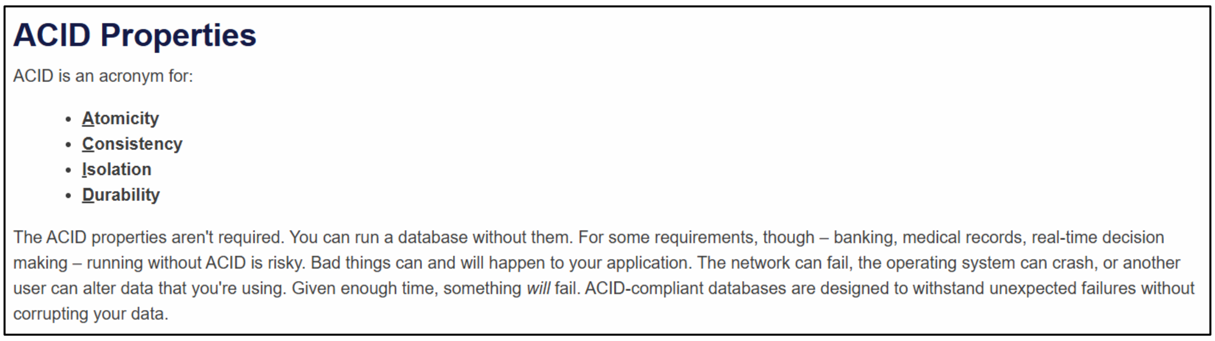


A bridge table acts as an intermediary between the two tables involved in the many-to-many relationship. It breaks down the many-to-many relationship into two one-to-many relationships.

Here's how it works:

1. **Structure:** The bridge table typically contains at least two foreign keys. Each foreign key references the primary key of one of the original tables. In our example, the bridge table, let's call it `StudentCourseEnrollments`, would have a `StudentID` (foreign key referencing the `Students` table) and a `CourseID` (foreign key referencing the `Courses` table).
2. **Linking Records:** Each row in the bridge table represents a specific instance of the relationship between a student and a course. For example, a row with `StudentID = 1` and `CourseID = 3` would indicate that student with ID 1 is enrolled in course with ID 3.
3. **Resolving Many-to-Many:** By introducing the bridge table, we now have two one-to-many relationships:
    - One-to-many between `Students` and `StudentCourseEnrollments` (one student can have multiple enrollments).
    - One-to-many between `Courses` and `StudentCourseEnrollments` (one course can have multiple enrollments).

ACID PROPERTIES:

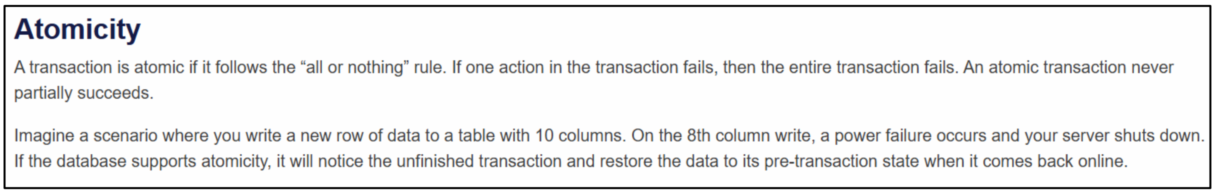


ACID Properties

ACID is an acronym for:

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

The ACID properties aren't required. You can run a database without them. For some requirements, though – banking, medical records, real-time decision making – running without ACID is risky. Bad things can and will happen to your application. The network can fail, the operating system can crash, or another user can alter data that you're using. Given enough time, something *will* fail. ACID-compliant databases are designed to withstand unexpected failures without corrupting your data.

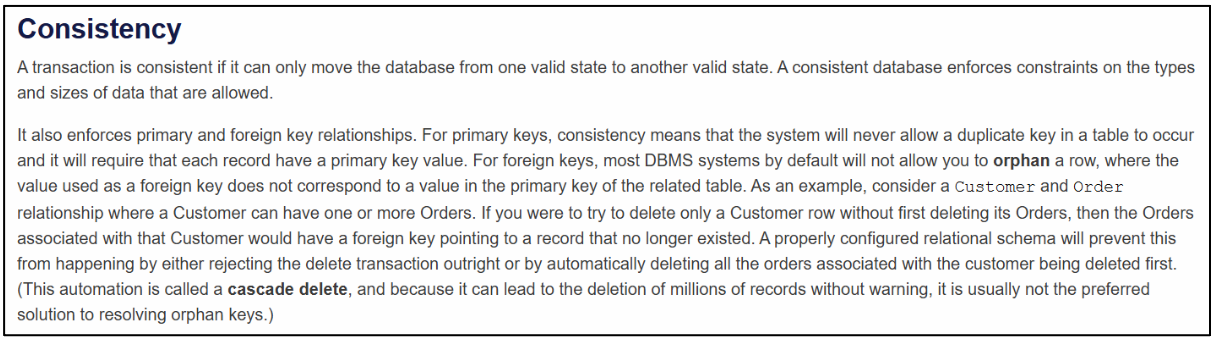


Atomicity

A transaction is atomic if it follows the "all or nothing" rule. If one action in the transaction fails, then the entire transaction fails. An atomic transaction never partially succeeds.

Imagine a scenario where you write a new row of data to a table with 10 columns. On the 8th column write, a power failure occurs and your server shuts down. If the database supports atomicity, it will notice the unfinished transaction and restore the data to its pre-transaction state when it comes back online.

Example: say we try to make a payment , either a payment is made or its is not made, there is no in between state.

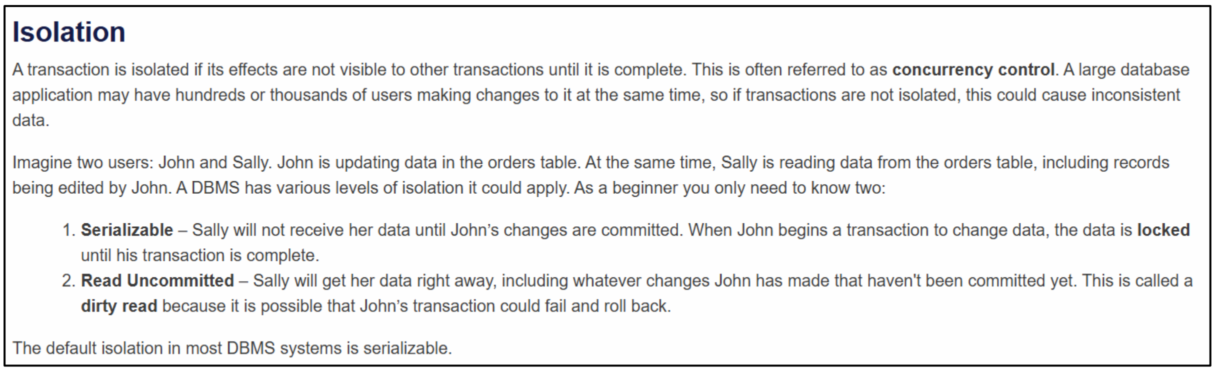


Consistency

A transaction is consistent if it can only move the database from one valid state to another valid state. A consistent database enforces constraints on the types and sizes of data that are allowed.

It also enforces primary and foreign key relationships. For primary keys, consistency means that the system will never allow a duplicate key in a table to occur and it will require that each record have a primary key value. For foreign keys, most DBMS systems by default will not allow you to **orphan** a row, where the value used as a foreign key does not correspond to a value in the primary key of the related table. As an example, consider a `Customer` and `Order` relationship where a Customer can have one or more Orders. If you were to try to delete only a Customer row without first deleting its Orders, then the Orders associated with that Customer would have a foreign key pointing to a record that no longer existed. A properly configured relational schema will prevent this from happening by either rejecting the delete transaction outright or by automatically deleting all the orders associated with the customer being deleted first. (This automation is called a **cascade delete**, and because it can lead to the deletion of millions of records without warning, it is usually not the preferred solution to resolving orphan keys.)

Example: once a person makes a payment , it should be valid or completely not valid. Say we made a payment and the money got deducted from our account. And if then the payment fails.
Then the money should return back to our account.



Isolation

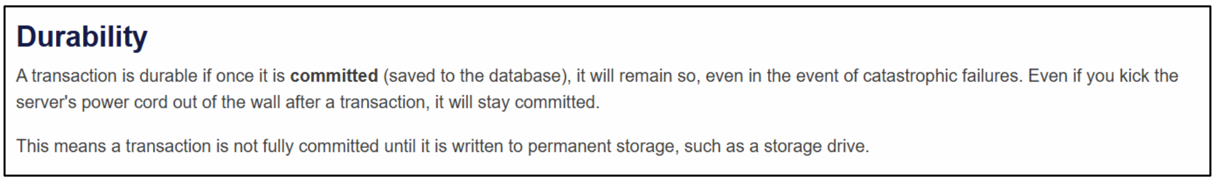
A transaction is isolated if its effects are not visible to other transactions until it is complete. This is often referred to as **concurrency control**. A large database application may have hundreds or thousands of users making changes to it at the same time, so if transactions are not isolated, this could cause inconsistent data.

Imagine two users: John and Sally. John is updating data in the orders table. At the same time, Sally is reading data from the orders table, including records being edited by John. A DBMS has various levels of isolation it could apply. As a beginner you only need to know two:

1. **Serializable** – Sally will not receive her data until John's changes are committed. When John begins a transaction to change data, the data is **locked** until his transaction is complete.
2. **Read Uncommitted** – Sally will get her data right away, including whatever changes John has made that haven't been committed yet. This is called a **dirty read** because it is possible that John's transaction could fail and roll back.

The default isolation in most DBMS systems is serializable.

Example: each persons account should be independent w.r.t. other person. Making a payment shouldn't have affect on other person's account.

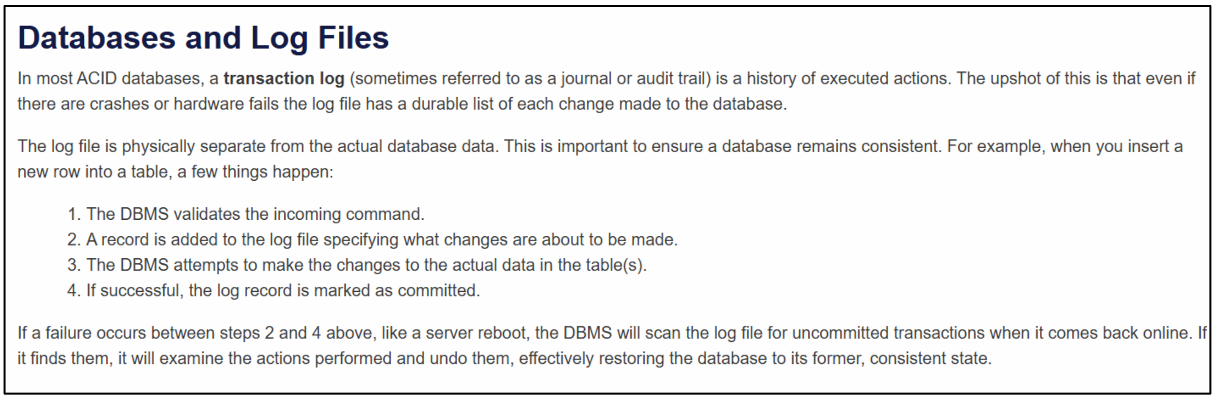


Durability

A transaction is durable if once it is **committed** (saved to the database), it will remain so, even in the event of catastrophic failures. Even if you kick the server's power cord out of the wall after a transaction, it will stay committed.

This means a transaction is not fully committed until it is written to permanent storage, such as a storage drive.

Example: when we make payment and our device crashes or turn's off, then it would rollback to the original space and won't cause any loss. And if its committed then transaction would pass.



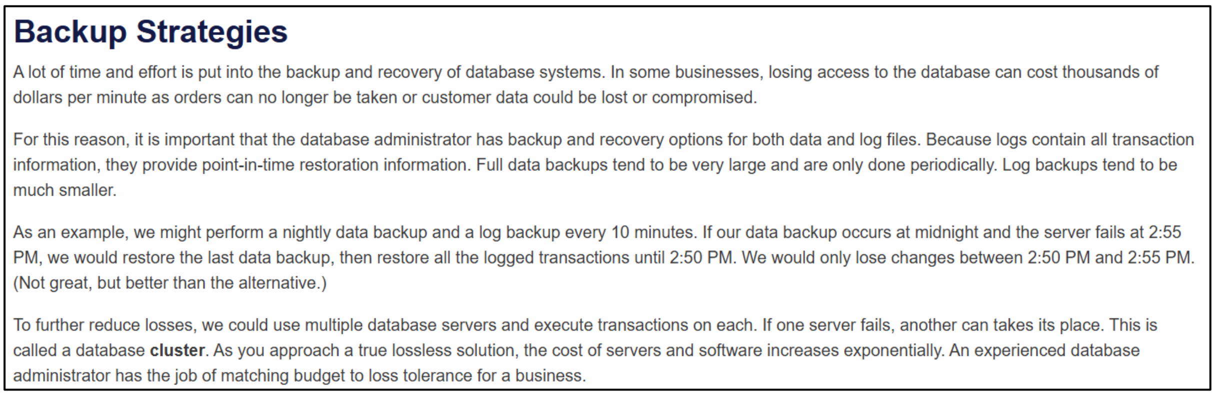
Databases and Log Files

In most ACID databases, a **transaction log** (sometimes referred to as a journal or audit trail) is a history of executed actions. The upshot of this is that even if there are crashes or hardware fails the log file has a durable list of each change made to the database.

The log file is physically separate from the actual database data. This is important to ensure a database remains consistent. For example, when you insert a new row into a table, a few things happen:

1. The DBMS validates the incoming command.
2. A record is added to the log file specifying what changes are about to be made.
3. The DBMS attempts to make the changes to the actual data in the table(s).
4. If successful, the log record is marked as committed.

If a failure occurs between steps 2 and 4 above, like a server reboot, the DBMS will scan the log file for uncommitted transactions when it comes back online. If it finds them, it will examine the actions performed and undo them, effectively restoring the database to its former, consistent state.



Backup Strategies

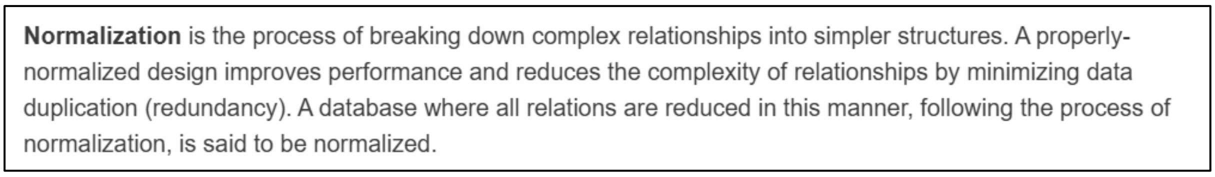
A lot of time and effort is put into the backup and recovery of database systems. In some businesses, losing access to the database can cost thousands of dollars per minute as orders can no longer be taken or customer data could be lost or compromised.

For this reason, it is important that the database administrator has backup and recovery options for both data and log files. Because logs contain all transaction information, they provide point-in-time restoration information. Full data backups tend to be very large and are only done periodically. Log backups tend to be much smaller.

As an example, we might perform a nightly data backup and a log backup every 10 minutes. If our data backup occurs at midnight and the server fails at 2:55 PM, we would restore the last data backup, then restore all the logged transactions until 2:50 PM. We would only lose changes between 2:50 PM and 2:55 PM. (Not great, but better than the alternative.)

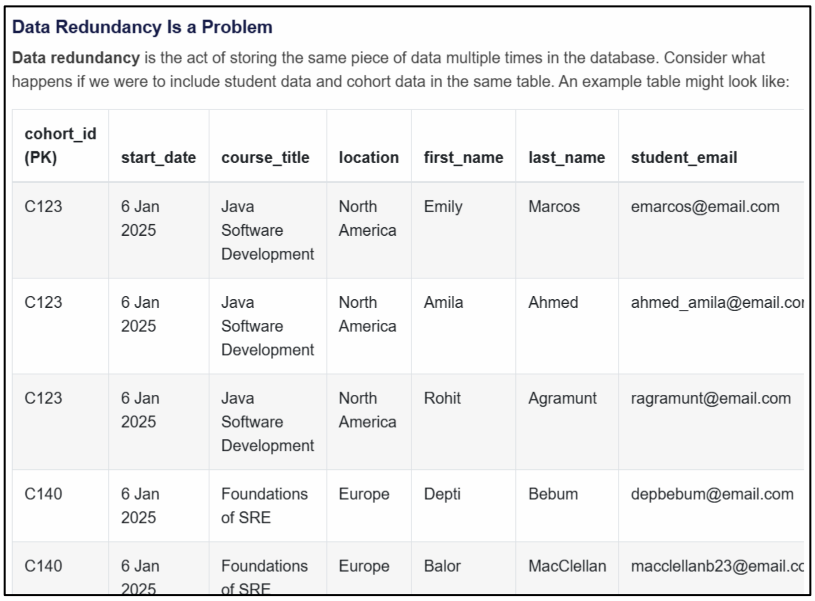
To further reduce losses, we could use multiple database servers and execute transactions on each. If one server fails, another can takes its place. This is called a database **cluster**. As you approach a true lossless solution, the cost of servers and software increases exponentially. An experienced database administrator has the job of matching budget to loss tolerance for a business.

What is Normalization?



Normalization is the process of breaking down complex relationships into simpler structures. A properly-normalized design improves performance and reduces the complexity of relationships by minimizing data duplication (redundancy). A database where all relations are reduced in this manner, following the process of normalization, is said to be normalized.

In normalisation we simply break a bigger table into smaller tables, which helps in reducing data redundancy and insertion, updation, deletion anomalies.

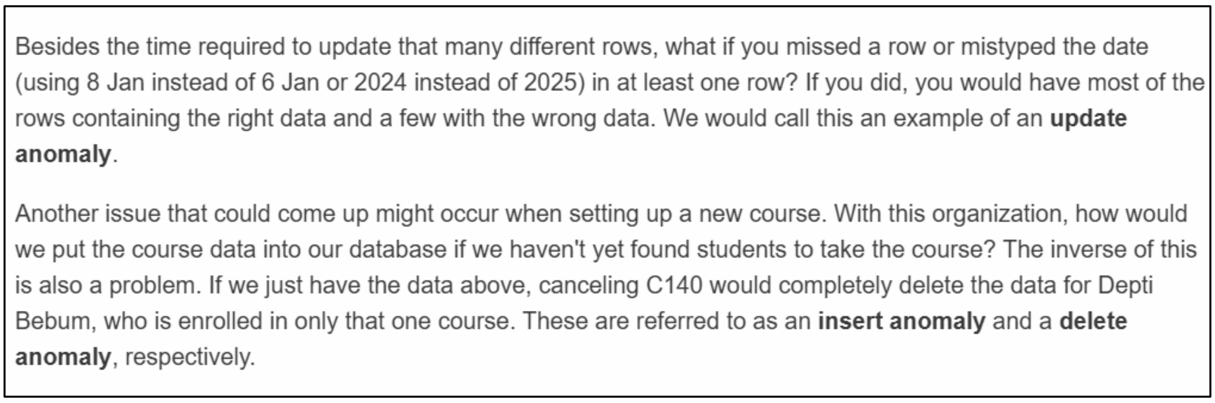


Data Redundancy Is a Problem

Data redundancy is the act of storing the same piece of data multiple times in the database. Consider what happens if we were to include student data and cohort data in the same table. An example table might look like:

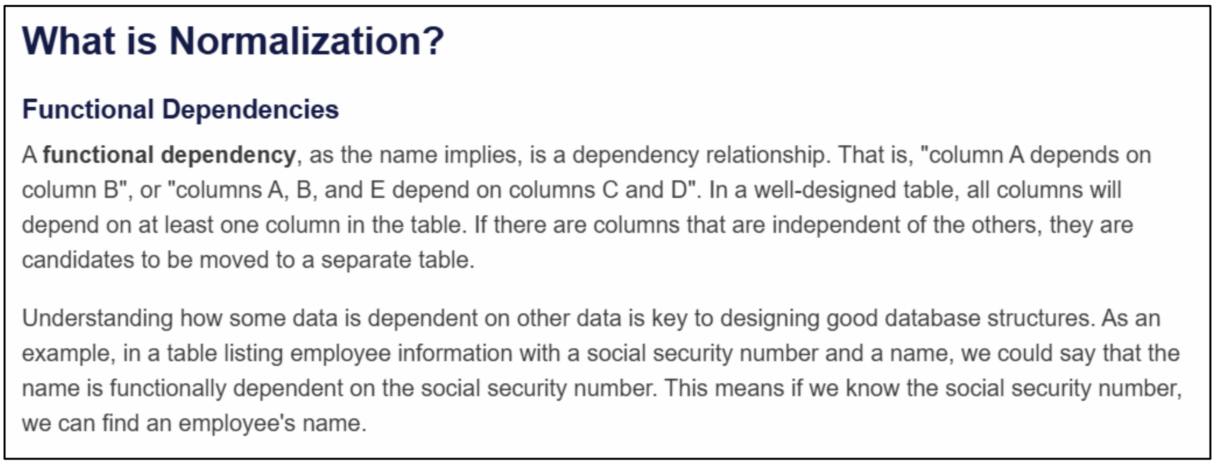
cohort_id (PK)	start_date	course_title	location	first_name	last_name	student_email
C123	6 Jan 2025	Java Software Development	North America	Emily	Marcos	emarcos@email.com
C123	6 Jan 2025	Java Software Development	North America	Amila	Ahmed	ahmed_amila@email.co
C123	6 Jan 2025	Java Software Development	North America	Rohit	Agramunt	ragramunt@email.com
C140	6 Jan 2025	Foundations of SRE	Europe	Depti	Bebum	depbebum@email.com
C140	6 Jan 2025	Foundations of SRE	Europe	Balor	MacClellan	macclellanb23@email.cc

This may lead to update and insert and deletion anomalies:



Besides the time required to update that many different rows, what if you missed a row or mistyped the date (using 8 Jan instead of 6 Jan or 2024 instead of 2025) in at least one row? If you did, you would have most of the rows containing the right data and a few with the wrong data. We would call this an example of an **update anomaly**.

Another issue that could come up might occur when setting up a new course. With this organization, how would we put the course data into our database if we haven't yet found students to take the course? The inverse of this is also a problem. If we just have the data above, canceling C140 would completely delete the data for Depti Bebum, who is enrolled in only that one course. These are referred to as an **insert anomaly** and a **delete anomaly**, respectively.



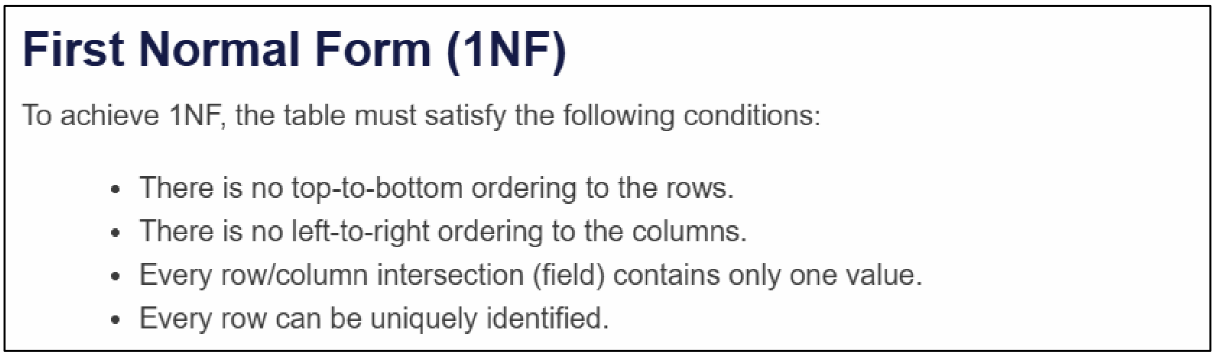
What is Normalization?

Functional Dependencies

A **functional dependency**, as the name implies, is a dependency relationship. That is, "column A depends on column B", or "columns A, B, and E depend on columns C and D". In a well-designed table, all columns will depend on at least one column in the table. If there are columns that are independent of the others, they are candidates to be moved to a separate table.

Understanding how some data is dependent on other data is key to designing good database structures. As an example, in a table listing employee information with a social security number and a name, we could say that the name is functionally dependent on the social security number. This means if we know the social security number, we can find an employee's name.

If B is functionally dependent on A, (or A functionally determines B A=>B) then A is known as Determinant and B is known as dependent. Like social security number defines all the employees uniquely, but converse is not true as name can repeat. (at least there should be one primary key).

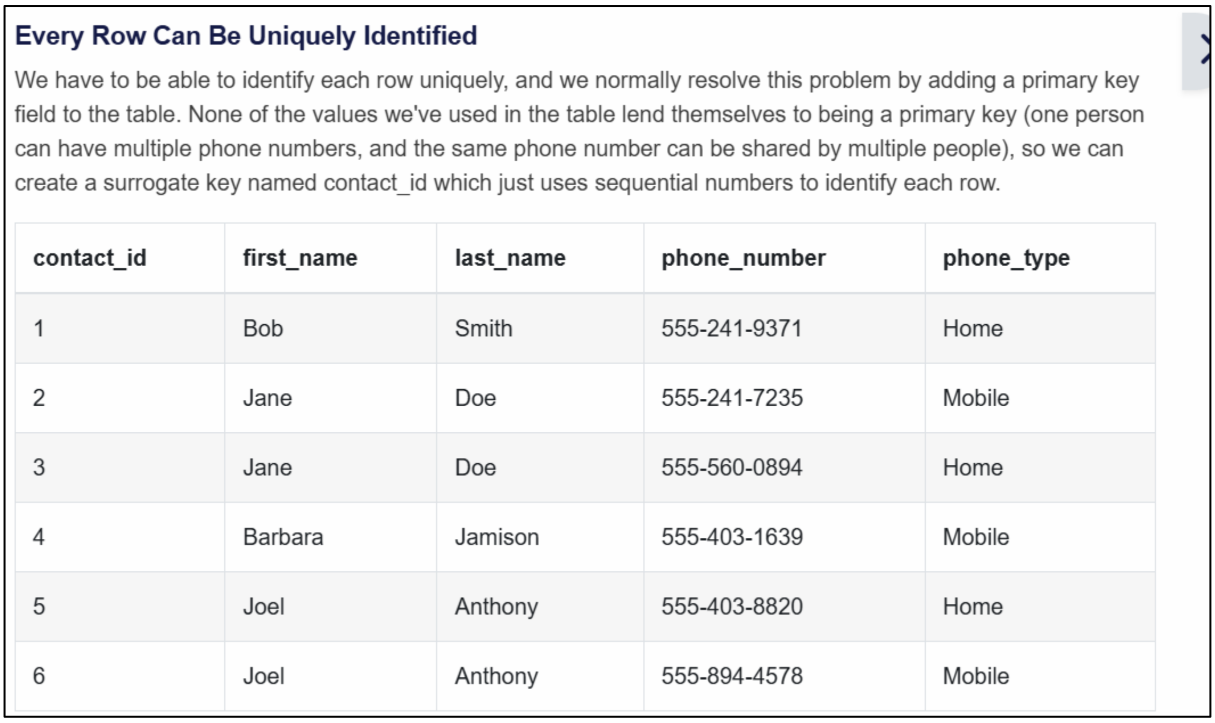


First Normal Form (1NF)

To achieve 1NF, the table must satisfy the following conditions:

- There is no top-to-bottom ordering to the rows.
- There is no left-to-right ordering to the columns.
- Every row/column intersection (field) contains only one value.
- Every row can be uniquely identified.

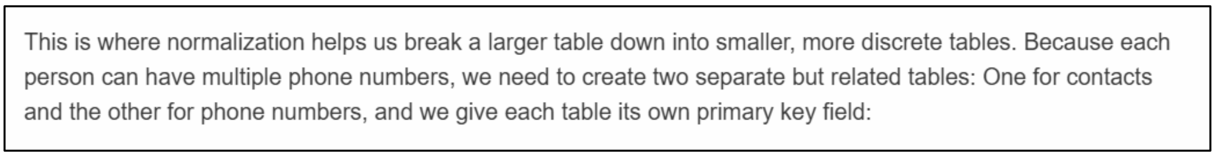
There should be atomic values, and each column should contain atomic value and no lists etc.
And each row should also have uniquely identified values.
Table should have primary key and no multi values.



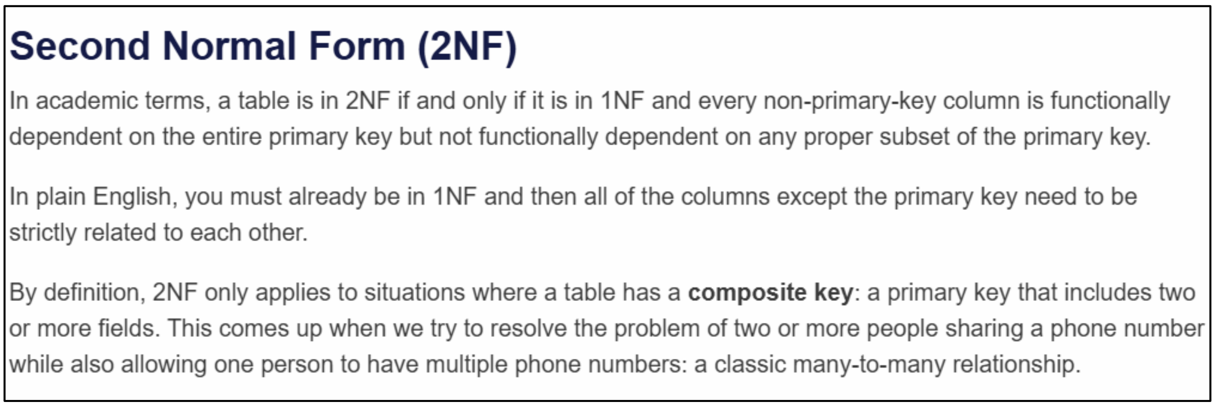
Every Row Can Be Uniquely Identified

We have to be able to identify each row uniquely, and we normally resolve this problem by adding a primary key field to the table. None of the values we've used in the table lend themselves to being a primary key (one person can have multiple phone numbers, and the same phone number can be shared by multiple people), so we can create a surrogate key named contact_id which just uses sequential numbers to identify each row.

contact_id	first_name	last_name	phone_number	phone_type
1	Bob	Smith	555-241-9371	Home
2	Jane	Doe	555-241-7235	Mobile
3	Jane	Doe	555-560-0894	Home
4	Barbara	Jamison	555-403-1639	Mobile
5	Joel	Anthony	555-403-8820	Home
6	Joel	Anthony	555-894-4578	Mobile



This is where normalization helps us break a larger table down into smaller, more discrete tables. Because each person can have multiple phone numbers, we need to create two separate but related tables: One for contacts and the other for phone numbers, and we give each table its own primary key field:

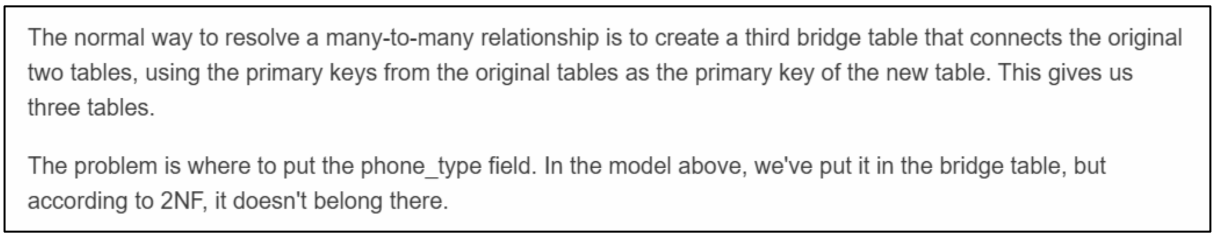


Second Normal Form (2NF)

In academic terms, a table is in 2NF if and only if it is in 1NF and every non-primary-key column is functionally dependent on the entire primary key but not functionally dependent on any proper subset of the primary key.

In plain English, you must already be in 1NF and then all of the columns except the primary key need to be strictly related to each other.

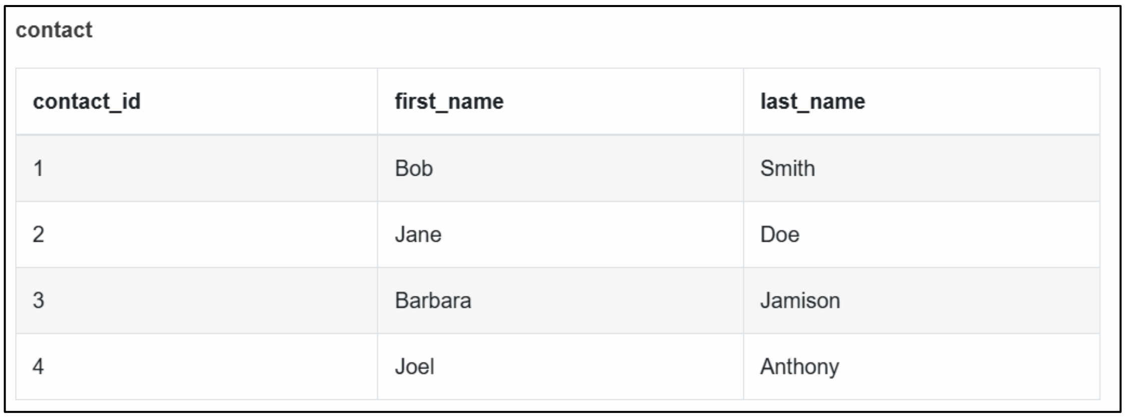
By definition, 2NF only applies to situations where a table has a **composite key**: a primary key that includes two or more fields. This comes up when we try to resolve the problem of two or more people sharing a phone number while also allowing one person to have multiple phone numbers: a classic many-to-many relationship.



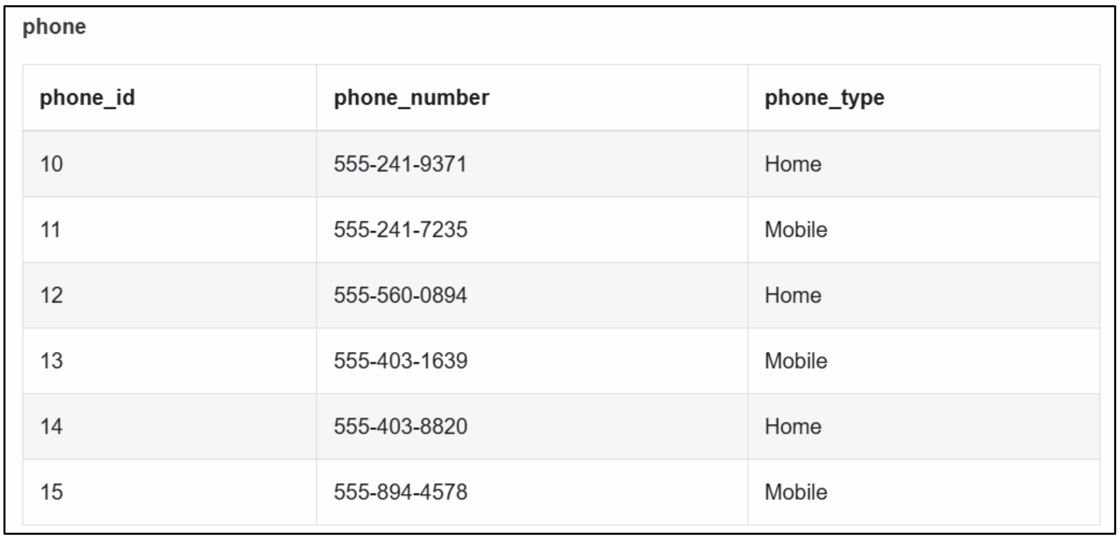
The normal way to resolve a many-to-many relationship is to create a third bridge table that connects the original two tables, using the primary keys from the original tables as the primary key of the new table. This gives us three tables.

The problem is where to put the phone_type field. In the model above, we've put it in the bridge table, but according to 2NF, it doesn't belong there.

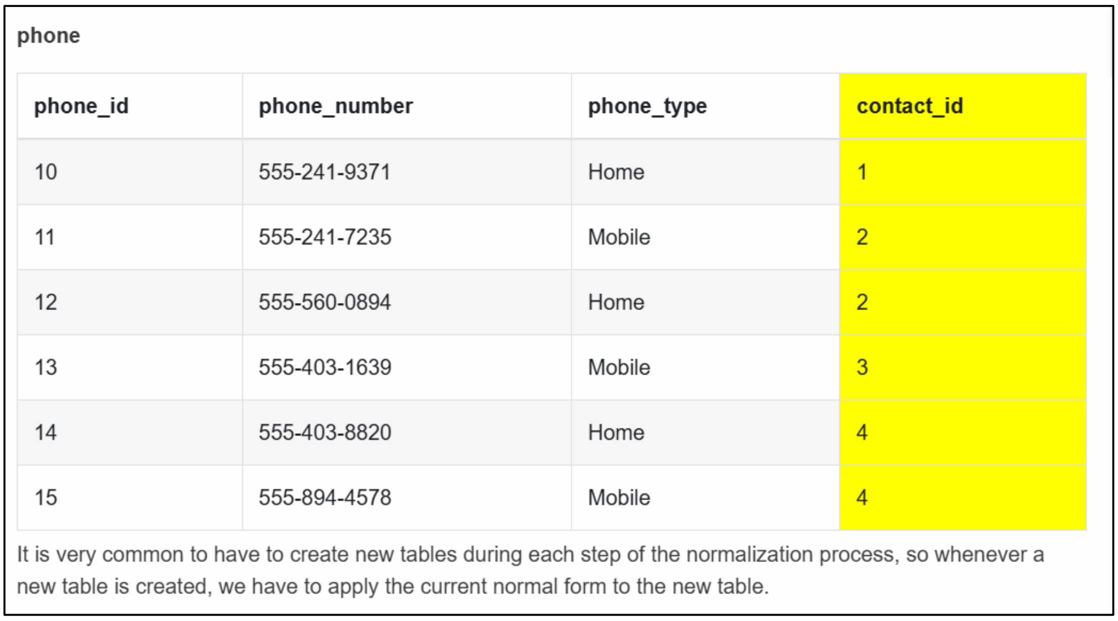
After 1st normal form, the table was divided into two parts one for contact and then for phone id, and then we added the contact id along with our phone id table.


contact

contact_id	first_name	last_name
1	Bob	Smith
2	Jane	Doe
3	Barbara	Jamison
4	Joel	Anthony


phone

phone_id	phone_number	phone_type
10	555-241-9371	Home
11	555-241-7235	Mobile
12	555-560-0894	Home
13	555-403-1639	Mobile
14	555-403-8820	Home
15	555-894-4578	Mobile

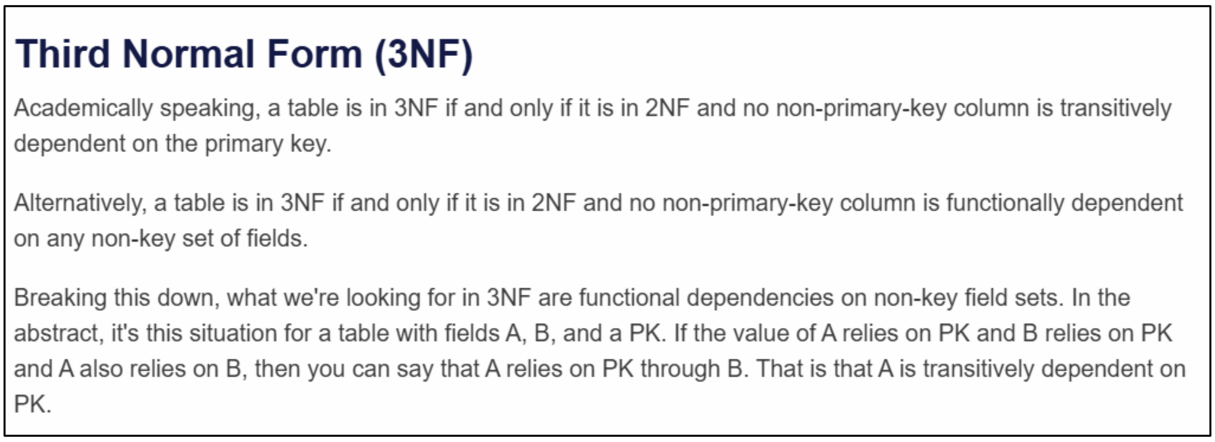

phone

phone_id	phone_number	phone_type	contact_id
10	555-241-9371	Home	1
11	555-241-7235	Mobile	2
12	555-560-0894	Home	2
13	555-403-1639	Mobile	3
14	555-403-8820	Home	4
15	555-894-4578	Mobile	4

It is very common to have to create new tables during each step of the normalization process, so whenever a new table is created, we have to apply the current normal form to the new table.

Then in 2nd normal form, We would create a bridge table which would contain both the contact as well as phone_id as composite key (group of two or more columns together to form a primary key). So as to link both the tables together.

Then in 3rd normal form:



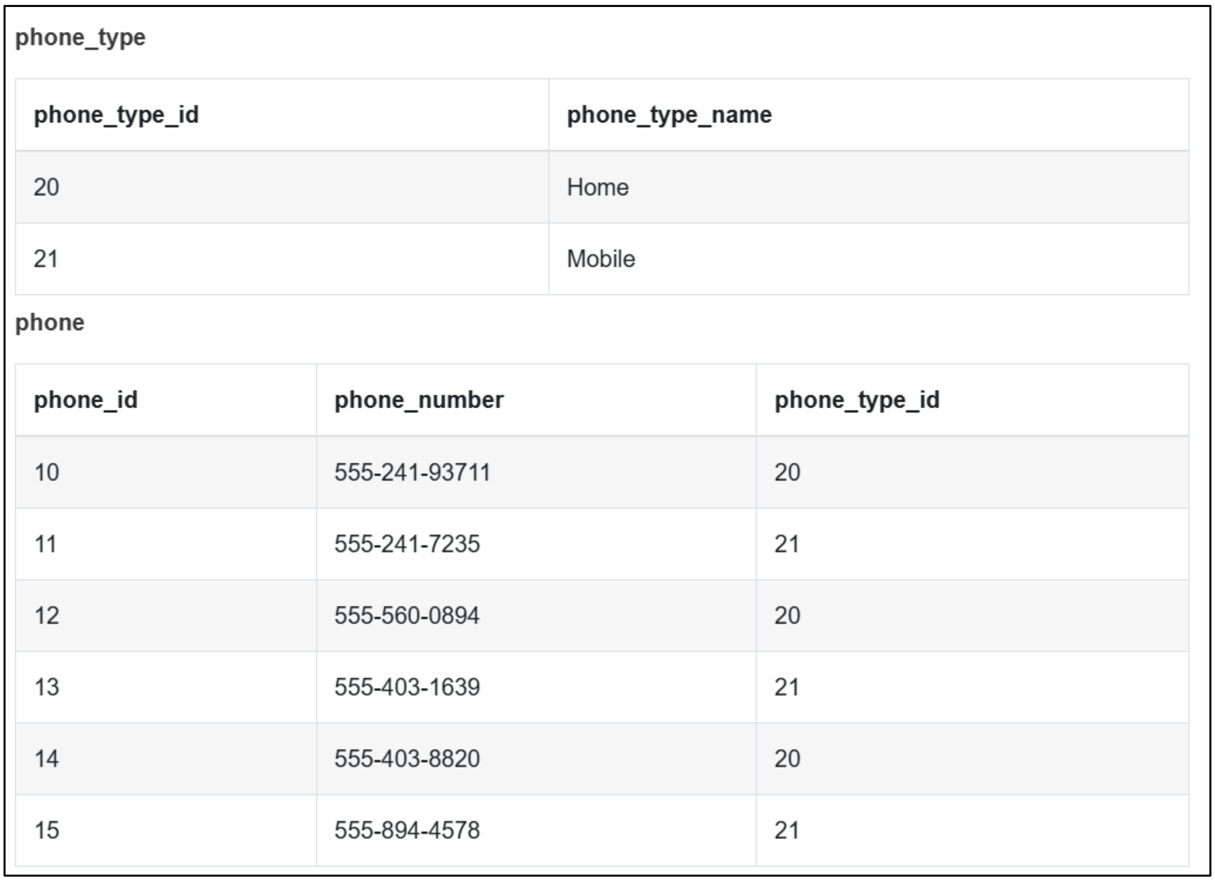
Third Normal Form (3NF)

Academically speaking, a table is in 3NF if and only if it is in 2NF and no non-primary-key column is transitively dependent on the primary key.

Alternatively, a table is in 3NF if and only if it is in 2NF and no non-primary-key column is functionally dependent on any non-key set of fields.

Breaking this down, what we're looking for in 3NF are functional dependencies on non-key field sets. In the abstract, it's this situation for a table with fields A, B, and a PK. If the value of A relies on PK and B relies on PK and A also relies on B, then you can say that A relies on PK through B. That is that A is transitively dependent on PK.

Here we create phone type id as it is not interdependent to phone number and replace it within the diagram:

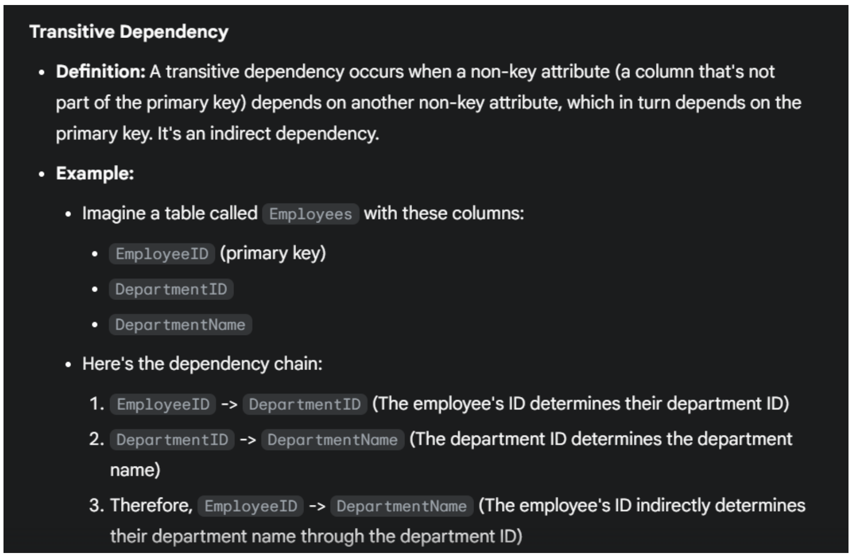


phone_type

phone_type_id	phone_type_name
20	Home
21	Mobile

phone

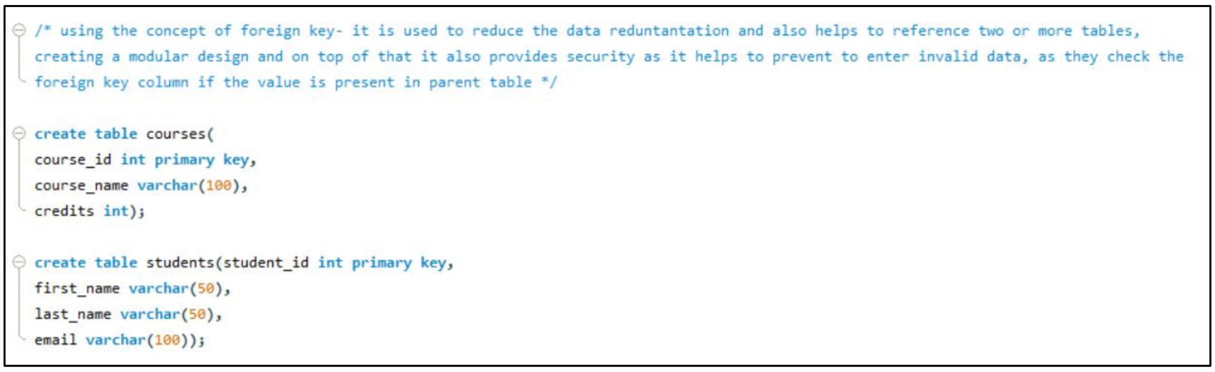
phone_id	phone_number	phone_type_id
10	555-241-93711	20
11	555-241-7235	21
12	555-560-0894	20
13	555-403-1639	21
14	555-403-8820	20
15	555-894-4578	21



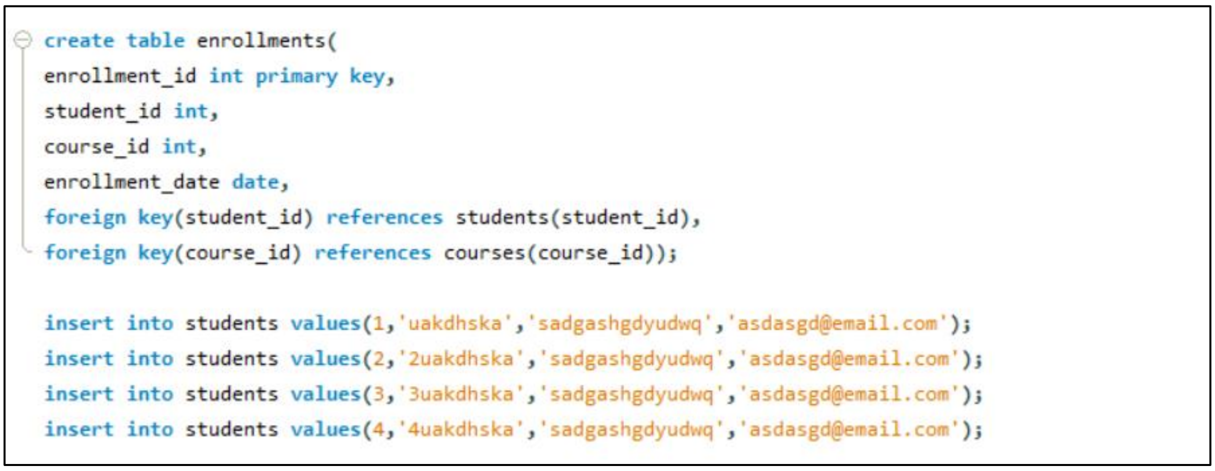
Transitive Dependency

- **Definition:** A transitive dependency occurs when a non-key attribute (a column that's not part of the primary key) depends on another non-key attribute, which in turn depends on the primary key. It's an indirect dependency.
- **Example:**
  - Imagine a table called `Employees` with these columns:
    - `EmployeeID` (primary key)
    - `DepartmentID`
    - `DepartmentName`
  - Here's the dependency chain:
    1. `EmployeeID` -> `DepartmentID` (The employee's ID determines their department ID)
    2. `DepartmentID` -> `DepartmentName` (The department ID determines the department name)
    3. Therefore, `EmployeeID` -> `DepartmentName` (The employee's ID indirectly determines their department name through the department ID)

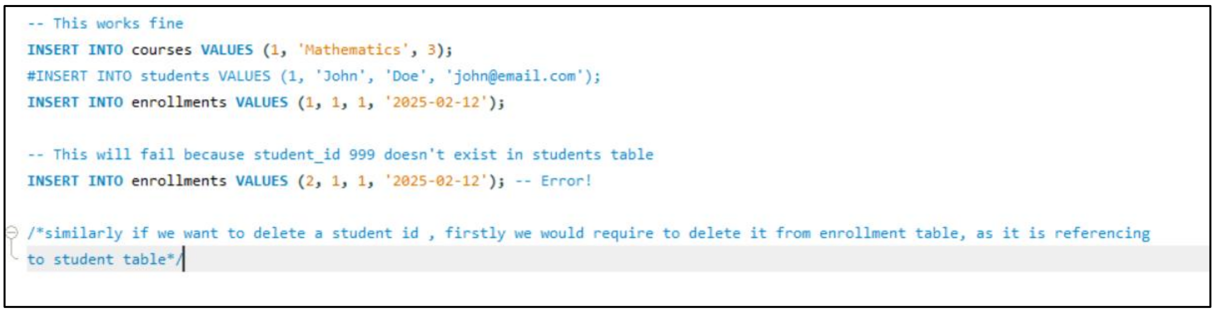
Using the concept of foreign key, to prevent reduntantation and on the same hand preventing invalid data entry:



```sql
/* using the concept of foreign key- it is used to reduce the data reduntantation and also helps to reference two or more tables,
creating a modular design and on top of that it also provides security as it helps to prevent to enter invalid data, as they check the
foreign key column if the value is present in parent table */

create table courses(
course_id int primary key,
course_name varchar(100),
credits int);

create table students(student_id int primary key,
first_name varchar(50),
last_name varchar(50),
email varchar(100));
```



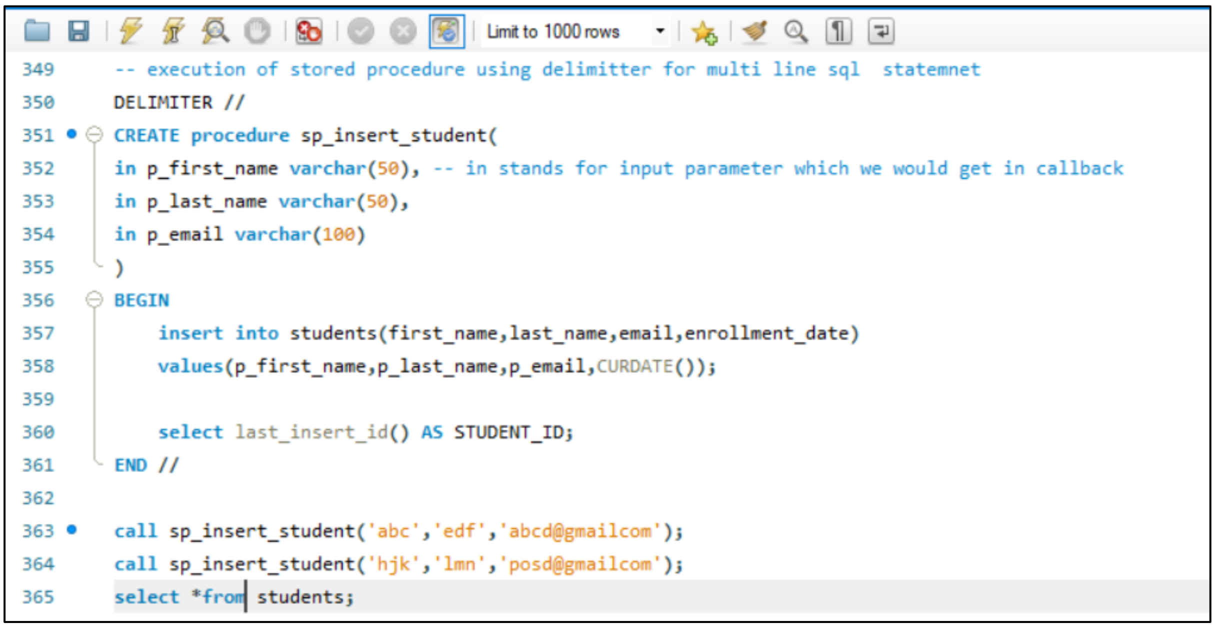
```sql
create table enrollments(
enrollment_id int primary key,
student_id int,
course_id int,
enrollment_date date,
foreign key(student_id) references students(student_id),
foreign key(course_id) references courses(course_id));

insert into students values(1,'uakdhska','sadgashgdyudwq','asdasgd@email.com');
insert into students values(2,'2uakdhska','sadgashgdyudwq','asdasgd@email.com');
insert into students values(3,'3uakdhska','sadgashgdyudwq','asdasgd@email.com');
insert into students values(4,'4uakdhska','sadgashgdyudwq','asdasgd@email.com');
```



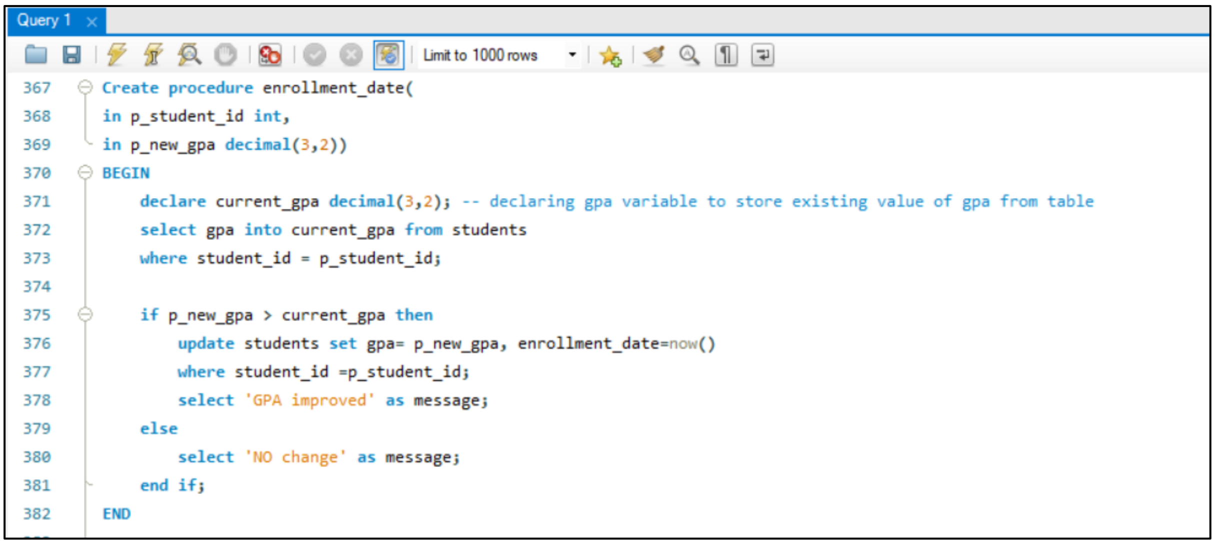
```sql
-- This works fine
INSERT INTO courses VALUES (1, 'Mathematics', 3);
#INSERT INTO students VALUES (1, 'John', 'Doe', 'john@email.com');
INSERT INTO enrollments VALUES (1, 1, 1, '2025-02-12');

-- This will fail because student_id 999 doesn't exist in students table
INSERT INTO enrollments VALUES (2, 1, 1, '2025-02-12'); -- Error!

/*similarly if we want to delete a student id , firstly we would require to delete it from enrollment table, as it is referencing
to student table*/
```

Now using the concept of stored procedure:

- Stored procedures are functions that can be called later, and it provides modularity and helps to avoid writing large pieces of code again and again.
- Now for creating a procedure, we can simply go in the schemas
- And then in stored_procedures and then right click on it
- to choose the option of creating a procedure
- and then we can simply create procedure.
- And use the procedure name with the values to perform callback.



```
-- execution of stored procedure using delimitter for multi line sql  statemnet
DELIMITER //
CREATE procedure sp_insert_student(
in p_first_name varchar(50), -- in stands for input parameter which we would get in callback
in p_last_name varchar(50),
in p_email varchar(100)
)
BEGIN
    insert into students(first_name,last_name,email,enrollment_date)
    values(p_first_name,p_last_name,p_email,CURDATE());

    select last_insert_id() AS STUDENT_ID;
END //

call sp_insert_student('abc','edf','abcd@gmailcom');
call sp_insert_student('hjk','lmn','posd@gmailcom');
select *from students;
```



```
Create procedure enrollment_date(
in p_student_id int,
in p_new_gpa decimal(3,2))
BEGIN
    declare current_gpa decimal(3,2); -- declaring gpa variable to store existing value of gpa from table
    select gpa into current_gpa from students
    where student_id = p_student_id;

    if p_new_gpa > current_gpa then
        update students set gpa= p_new_gpa, enrollment_date=now()
        where student_id =p_student_id;
        select 'GPA improved' as message;
    else
        select 'NO change' as message;
    end if;
END
```