

## **Normalization for Relational Databases**

Normalization is the process of organizing data in a database. It includes creating tables and establishing relationships between those tables according to rules designed both to protect the data and to make the database more flexible by eliminating redundancy and inconsistent dependency.

### **Important points regarding normal forms in DBMS:**

1. First Normal Form (1NF): This is the most basic level of normalization. In 1NF, each table cell should contain only a single value, and each column should have a unique name. The first normal form helps to eliminate duplicate data and simplify queries.
2. Second Normal Form (2NF): 2NF eliminates redundant data by requiring that each non-key attribute be dependent on the primary key. This means that each column should be directly related to the primary key, and not to other columns.
3. Third Normal Form (3NF): 3NF builds on 2NF by requiring that all non-key attributes are independent of each other. This means that each column should be directly related to the primary key, and not to any other columns in the same table.
4. Boyce-Codd Normal Form (BCNF): BCNF is a stricter form of 3NF that ensures that each determinant in a table is a candidate key. In other words, BCNF ensures that each non-key attribute is dependent only on the candidate key.
5. Fourth Normal Form (4NF): 4NF is a further refinement of BCNF that ensures that a table does not contain any multi-valued dependencies.
6. Fifth Normal Form (5NF): 5NF is the highest level of normalization and involves decomposing a table into smaller tables to remove data redundancy and improve data integrity.

Normal forms help to reduce data redundancy, increase data consistency, and improve database performance. However, higher levels of normalization can lead to more complex database designs and queries. It is important to strike a balance between normalization and practicality when designing a database

## **Functional Dependencies and Normalization**

### **Functional Dependencies:**

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has  $n$  attributes  $A_1, A_2, \dots, A_n$ . Let us think of the whole database as being described by a single universal relation schema  $R = \{A_1, A_2, \dots, A_n\}$ .

A functional dependency, denoted by  $X \rightarrow Y$ , between two sets of attributes  $X$  and  $Y$  that are subsets of  $R$  specifies a constraint on the possible tuples that can form a relation state  $r$  of  $R$ . The constraint is that, for any two tuples  $t_1$  and  $t_2$  in  $r$  that have  $t_1[X] = t_2[X]$ , they must also have  $t_1[Y] = t_2[Y]$ .

$X \rightarrow Y$  means that  $X$  uniquely determines  $Y$  or  $Y$  is uniquely determined by  $X$ . This means that the values of the  $Y$  component of a tuple in  $r$  depend on, or are determined by, the values of the  $X$  component; alternatively, the values of the  $X$  component of a tuple uniquely (or functionally) determine the values of the  $Y$  component. We also say that there is a functional dependency from  $X$  to  $Y$ , or that  $Y$  is functionally dependent on  $X$ . The left-hand side of the Functional Dependency is sometimes called the determinant ( $X$ ) and the right-hand side is called dependent ( $Y$ ).

Functional Dependencies are divided into two types.

1. Full Functional Dependency
2. Partial Functional Dependency

### **Fully Functional Dependency :**

If  $X$  and  $Y$  are an attribute set of a relation,  $Y$  is fully functional dependent on  $X$ , if  $Y$  is functionally dependent on  $X$  but not on any proper subset of  $X$ .

### **Example –**

In the relation  $ABC \rightarrow D$ , attribute  $D$  is fully functionally dependent on  $ABC$  and not on any proper subset of  $ABC$ . That means that subsets of  $ABC$  like  $AB$ ,  $BC$ ,  $A$ ,  $B$ , etc cannot determine  $D$ .

Let us take another example –

### **Supply table**

supplier_id	item_id	price
1	1	540

2	1	545
1	2	200
2	2	201
1	1	540
2	2	201
3	1	542

From the table, we can clearly see that neither supplier\_id nor item\_id can uniquely determine the price but both supplier\_id and item\_id together can do so. So we can say that price is fully functionally dependent on { supplier\_id, item\_id }. This summarizes and gives our fully functional dependency –

{ supplier\_id , item\_id } -> price

### **Partial Functional Dependency :**

A functional dependency  $X \rightarrow Y$  is a partial dependency if Y is functionally dependent on X and Y can be determined by any proper subset of X.

For example, we have a relationship AC->B, A->D, and D->B.

Now if we compute the closure of  $\{A^+\} = ADB$

Here A is alone capable of determining B, which means B is partially dependent on AC.

Let us take another example –

### **Student table**

name	roll_no	course
Ravi	2	DBMS
Tim	3	OS
John	5	Java

Here, we can see that both the attributes name and roll\_no alone are able to uniquely identify a course. Hence we can say that the relationship is partially dependent.

## Normalization and Normal Forms in DBMS

If a dataset is maintained in the form of just a single table, it leads to Data Redundancy, which means a single value of data is stored multiple times. This leads to many issues like an increment in the database size, slower data retrieval, and data inconsistency. Thus, to overcome this, Normalization in DBMS is used in which a large table is reduced into smaller tables until each of the single tables contains one relation.

As a result of Normalization, the redundancy in a table is removed which improves the efficiency of the database system. Let's understand the Need for Normalization in DBMS using a simple example. Consider a table having Student ID, name, Branch Name, and Branch Code.

S_ID	S_NAME	Branch Name	Branch Code
1.	John	CS	101
2.	Brat	CS	101
3.	Siemen	Electrical	106

This is the single table storing all the data related to a college student. Storing data in this way leads to the following problems:

- If you want to “insert” a name and ID of a new student, you cannot do it until you don't have his branch name and branch code. This is called Insertion Anomaly.
- If you want to “delete” a student name, then the branch name and code will also be deleted. The deleted branch name and code cannot be recovered again. This is called Deletion Anomaly.
- If you want to “update” the branch name of John from CS to Civil, the update for Brat will also happen. Thus, multiple updations of single data occur. This is called Updation Anomaly.

A simple solution to the above problems is to reduce the dataset into two tables using Normalization in DBMS in the following way.

Table 1

S_ID	S_NAME	Branch Code
1.	John	101
2.	Brat	101
3.	Siemen	106

Table 2

Branch Code	Branch Code
101	CS
106	Electrical

As you can see, there is no need to repeatedly store the branch data. Also, new data can be inserted into the data separately without any anomaly. Therefore, Normalization in DBMS provides us with a way to remove the above anomalies using the concept of Normal Forms in DBMS. Before learning about the types of Normal Forms in DBMS, you should be aware of the concept of *Functional Dependency in DBMS*.

### Types of Normal Forms

There are the following Normal Forms in DBMS:

#### First Normal Form (1NF)

The First Normal Form of Normalization in DBMS states that an attribute of a Table cannot hold Multiple Values. We can also say that an attribute must be single-valued. In other words, the composite attribute or multivalued attribute is not allowed.

Thus, A Table is said to be in 1NF if:


- All columns contain atomic values.
- All values of a column(attribute) belong to the same domain.

- Column names are unique.

For example, if there are multiple branch names for a student, they must be converted to a single-valued attribute as shown below.

**Conversion of Table into First Normal Form (1NF)**

ID	Name	Branch Name
1.	John	CS, Civil
2.	Ben	Electronics
3.	Steve	Mechanical
4.		
5.		



ID	Name	Branch Name
1.	John	CS
2.	John	Civil
3.	Ben	Electronics
4.	Steve	Mechanical
5.		

**Second Normal Form (2NF)**

A Table ( Relation) in the Database exists in the Second Normal Form if:

- It exists in the 1NF.
- It does not have Partial Dependency

Let's understand the Second Normal Form of Normalization in DBMS with an example. Suppose there is a table 'Product' containing the Customer ID, Name, and Price of Products.

Customer_ID	Name	Price
1	Fan	1000
2	AC	1500
1	Phone	2000
4	Lamp	1000
4	Fan	1000
2	Earphones	2000

This table contains some redundancy as the price of the Fan is stored twice. This table is in 1NF as all values are atomic. Now, let's see for 2NF.

Here,

Candidate Keys are {Customer\_ID, Name}. The Prime Attributes ( attributes that are a part of the candidate key) will be Customer\_ID and Name and the Non-Prime Attribute will be Price. We see that Customer\_ID->Price is a Partial Functional Dependency because Price (non-prime attribute) depends on Customer\_ID(a subset of the candidate key).

Therefore, we conclude the table is not in 2NF. To convert the table to 2NF using Normalization in DBMS, we decompose it into two tables as given below:

### Conversion to Second Normal Form (2NF)

Table 1

Customer ID	Product Name
1	Fan
1	Phone
2	AC
2	Earphones
4	Lamp
4	Fan

Table 2

Product Name	Price
Fan	1000
Phone	2000
AC	1500
Earphones	2000
Lamp	1000

Both the above tables are in 1NF. Also, the Functional Dependencies are Customer ID->Product Name and Product Name->Price in which there is no Partial Dependency. Also, the price of the Fan is not stored multiple times. So, Data Redundancy is removed.

### Third Normal Form (3NF)

The Third Normal Form exists in a table if:

- It exists in the 2NF.
- There is no Transitive Dependency for non-prime attributes in the table.

Now, let's first see Transitive Functional Dependency before going ahead with the Normalization in DBMS.

### *Transitive Functional Dependency*

Transitive Functional Dependency occurs when a non-prime attribute is dependent on another non-prime attribute. For example, in a relation, if Student ID determines Student's City then Student ID -> Student City is a Functional Dependency. And, if Student's State is determined by Student's City then Student City->Student State is also a Functional Dependency.



But, Student City is a Non-Prime Attribute that depends on Student ID (Prime Attribute). Therefore, Student State becomes indirectly dependent upon Student ID. We conclude that If Student ID  $\rightarrow$  Student City and Student City  $\rightarrow$  Student State exists, then Student ID  $\rightarrow$  Student State also exists.

Now, let's see how to convert a table into the Third Normal Form (3NF) of Normalization in DBMS. Suppose there is a table containing the data of students as follows:

STU_ID	STU_NAME	ZIPCODE	STU_STATE	STU_CITY
1.	Harry	201010	UP	Noida
2.	Stephan	02228	US	Boston
3.	<u>Lan</u>	60007	US	Chicago
4.	Katharine	06389	UK	Norwich
5.	John	462007	MP	Bhopal

In this table, the Candidate Key is Student ID which is a prime attribute, and the rest of the attributes are Non-Prime. Thus, there is no partial dependency. So, it is in 2NF.

But, STU\_ID  $\rightarrow$  ZIPCODE and ZIPCODE  $\rightarrow$  STU\_STATE are the two functional dependencies that exist. Since Zipcode is a non-prime attribute, STU\_STATE indirectly depends on STU\_ID. Therefore, STU\_ID  $\rightarrow$  STU\_STATE also exists which is a Transitive Functional Dependency.

To convert this into 3NF, we decompose the relation into two tables as below.

### Conversion to Third Normal Form (3NF)

Table 1

STU_ID	STU_NAME	ZIPCODE
1.	Harry	201010
2.	Stephan	02228
3.	Lan	60007
4.	Katharine	06389
5.	John	462007

Table 2

ZIPCODE	STU_STATE	STU_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Now, both the tables are free from Transitive Functional Dependency and thus exist in 3NF of Normalization in DBMS.

#### Boyce-Codd Normal Form (BCNF)

A table or a relation exists in the BCNF if:

- It exists in 3NF.
- For every Functional Dependency  $A \rightarrow B$  in the table, A is a Super Key.

In simple words, if there exists a Functional Dependency  $X \rightarrow Y$  in the table such that Y is a Prime Attribute and X is a non-prime attribute, then the table is not in BCNF.

Let's see how to identify whether a Table exists in BCNF or not using an example. Suppose there is a table named 'Customer Service' which contains the data about the Product, Customer, and Seller. We can observe here that despite Normalization up to 3NF, there is Data Redundancy in the table.

For the given table:

- Candidate Keys = {Product ID, Customer Name}
- Prime Attributes: Product ID and Customer Name
- Non-Prime Attributes: Seller Name

Product ID	Customer Name	Seller Name
1	x	ABC
1	y	RST
2	z	ABC
3	x	PQR
4	x	PQR
5	x	PQR

Data Redundancy



Therefore, the following conclusions can be drawn from the table:

1. Since all the values in the columns are atomic, the **Table is in 1NF**.
2. The dependency Customer Name -> Seller Name is not possible because One Customer may purchase from more than one seller. Thus, there is no Partial Dependency i.e. seller name(non-prime attribute) does not depend on Customer Name(a subset of Candidate Key). **Thus, the Table is in 2NF**.
3. Functional Dependencies present in the table are:

- Product ID, Customer Name->Seller Name
- Seller Name->Customer Name

For Transitive Dependency, a non-prime attribute should determine another non-prime attribute. But here, Seller Name is a non-prime attribute but Customer Name is a Prime Attribute. Therefore, there is no Transitive Dependency. Thus, **Table exists in 3NF**.

4. For a table to exist in BCNF, in Every Functional Dependency, LHS should be a Super Key. But in Seller Name->Customer Name, Seller Name is not a Key. Therefore, **the table does not exist in BCNF.**

Now, let's convert the table into BCNF Form of Normalization in DBMS. We break the table into two tables as follows:

**Conversion to BCNF**

Table 1

Product ID	Seller Name
1	ABC
1	RST
2	ABC
3	PQR

Table 2

Seller Name	Customer Name
ABC	x
ABC	z
RST	y
PQR	x

Therefore, the above two tables satisfy the conditions of BCNF. This is how the Normal Forms in DBMS remove the data redundancy in a Relational Database.

**Some key points:**

**1<sup>st</sup> NF:**

- No multivalued.
- Cell value may be NULL

**2<sup>nd</sup> NF:**

- 1<sup>st</sup> NF
- Non partial dependency

**Example:**

**AB → D**

**$B \rightarrow C$**

**So, AB----- $\rightarrow$  Candidate Key**

**Here A,B  $\in$  Prime attribute**

**C,D  $\in$  Non prime attribute**

**Using prime attribute find non prime attribute**

**B $\rightarrow$ C Partial dependency**

**3<sup>rd</sup> NF:**

➤ **2<sup>nd</sup> NF**

➤ **No transitive dependency**

**Example:**

**Every dependency  $\alpha \rightarrow \beta$**

**(i) Either  $\alpha$  is Super Key**

**(ii) Or  $\beta$  is prime attribute**

**So, No**

**Partial Dependency: P (prime attribute) $\rightarrow$ NP (non prime attribute)**

**Transitive Dependency: NP (non prime attribute) $\rightarrow$  NP (non prime attribute)**

**If right hand side P (prime) then 3<sup>rd</sup> NF**

**If Lt hand side Super Key then also 3<sup>rd</sup> NF**

**BCNF:**

➤ **3<sup>rd</sup> NF**

➤  **$\alpha \rightarrow \beta$   $\alpha$  should be Super Key**

## Algorithms for Query Processing and Optimization

- A query expressed in a high-level query language such as SQL must be **scanned, parsed, and validate**.
- Scanner: identify the language tokens.
- Parser: check query syntax.
- Validate: check all attribute and relation names are valid.
- An internal representation (**query tree or query graph**) of the query is created after scanning, parsing, and validating.
- Then DBMS must devise **an execution strategy** for retrieving the result from the database files.
- How to choose a suitable (efficient) strategy for processing a query is known as query **optimization**.
- The term optimization is actually a misnomer because in some cases the chosen execution plan is not the optimal strategy – it is just a reasonably efficient one.
- There are two main techniques for implementing query optimization.
  - **Heuristic rules** for re-ordering the operations in a query.
  - **Systematically estimating** the cost of different execution strategies and choosing the lowest cost estimate.

### **Translating SQL Queries into Relation Algebra**

- An SQL query is first translated into an equivalent extended relation algebra expression (as a query tree) that is then optimized.
- Query block in SQL: the basic unit that can be translated into the algebraic operators and optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses.
- Consider the following SQL query.

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > ( SELECT MAX (SALARY)
                  FROM EMPLOYEE
                  WHERE DNO=5);
```

- We can decompose it into two blocks:

Inner block:

```
( SELECT MAX (SALARY)
FROM EMPLOYEE
WHERE DNO=5)
```

Outer block:

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > c
```

– Then translate to algebra expressions:

\* Inner block:  $\text{=MAX SALARY } (\sigma_{\text{DNO}=5} \text{ EMPLOY EE})$

\* Outer block:  $\pi_{\text{LNAME, F NAME}}(\sigma_{\text{SALARY} > c} \text{ EMPLOY EE})$

# Transaction Processing

In computer science, transaction processing is information processing <sup>[1]</sup> that is divided into individual, indivisible operations called *transactions*. Each transaction must succeed or fail as a complete unit; it can never be only partially complete.

For example, when you purchase a book from an online bookstore, you exchange money (in the form of credit) for a book. If your credit is good, a series of related operations ensures that you get the book and the bookstore gets your money. However, if a single operation in the series fails during the exchange, the entire exchange fails. You do not get the book and the bookstore does not get your money. The technology responsible for making the exchange balanced and predictable is called *transaction processing*. Transactions ensure that data-oriented resources are not permanently updated unless all operations within the transactional unit complete successfully. By combining a set of related operations into a unit that either completely succeeds or completely fails, one can simplify error recovery and make one's application more reliable.

## Methodology

---

The basic principles of all transaction-processing systems are the same. However, the terminology may vary from one transaction-processing system to another, and the terms used below are not necessarily universal.

### Rollback

Transaction-processing systems ensure database integrity by recording intermediate states of the database as it is modified, then using these records to restore the database to a known state if a transaction cannot be committed. For example, copies of information on the database *prior* to its modification by a transaction are set aside by the system before the transaction can make any modifications (this is sometimes called a *before image*). If any part of the transaction fails before it is committed, these copies are used to restore the database to the state it was in before the transaction began.

### Rollforward

It is also possible to keep a separate journal of all modifications to a database management system. (sometimes called *after images*). This is not required for rollback of failed transactions but it is useful for updating the database management system in the event of a database failure, so some transaction-processing systems provide it. If the database management system fails entirely, it must be restored from the most recent back-up. The back-up will not reflect transactions committed since the back-up was made. However, once the database management system is restored, the journal of after images can be applied to the database (*rollforward*) to bring the database management system up to date. Any transactions in progress at the time of the failure can then be rolled back. The result is a database in a consistent, known state that includes the results of all transactions committed up to the moment of failure.



## Deadlocks

In some cases, two transactions may, in the course of their processing, attempt to access the same portion of a database at the same time, in a way that prevents them from proceeding. For example, transaction A may access portion X of the database, and transaction B may access portion Y of the database. If at that point, transaction A then tries to access portion Y of the database while transaction B tries to access portion X, a *deadlock* occurs, and neither transaction can move forward. Transaction-processing systems are designed to detect these deadlocks when they occur. Typically both transactions will be cancelled and rolled back, and then they will be started again in a different order, automatically, so that the deadlock doesn't occur again. Or sometimes, just one of the deadlocked transactions will be cancelled, rolled back, and automatically restarted after a short delay.

Deadlocks can also occur among three or more transactions. The more transactions involved, the more difficult they are to detect, to the point that transaction processing systems find there is a practical limit to the deadlocks they can detect.

## Compensating transaction

In systems where commit and rollback mechanisms are not available or undesirable, a compensating transaction is often used to undo failed transactions and restore the system to a previous state.

## ACID criteria

---

Jim Gray defined properties of a reliable transaction system in the late 1970s under the acronym *ACID*—atomicity, consistency, isolation, and durability.<sup>[1]</sup>

### Atomicity

A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

### Consistency

A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

### Isolation

Even though transactions execute concurrently, it appears to each transaction T, that others executed either before T or after T, but not both.

### Durability

Once a transaction completes successfully (commits), its changes to the database survive failures and retain its changes.

## Concurrency Control Techniques

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories –

- Lock based protocols
- Time stamp based protocols

### Lock-based Protocols

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds –

- **Binary Locks** – A lock on a data item can be in two states; it is either locked or unlocked.
- **Shared/exclusive** – This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

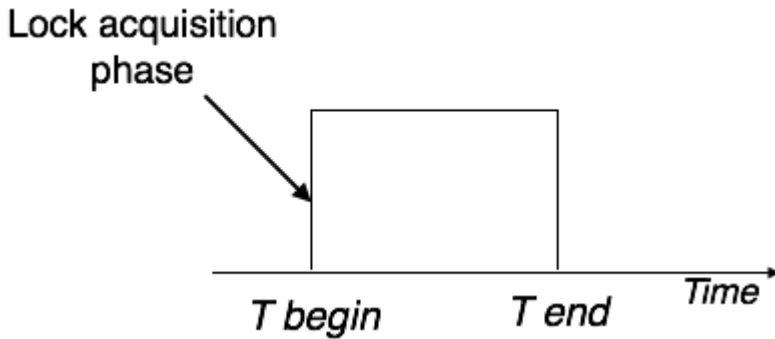
There are four types of lock protocols available –

### Simplistic Lock Protocol

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the 'write' operation.

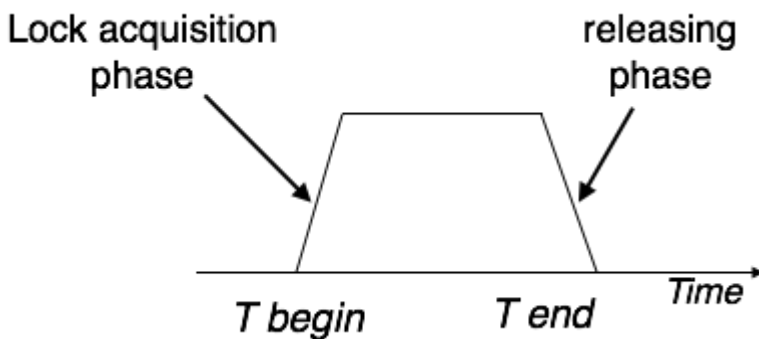
### Pre-claiming Lock Protocol

Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.



### Two-Phase Locking 2PL

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

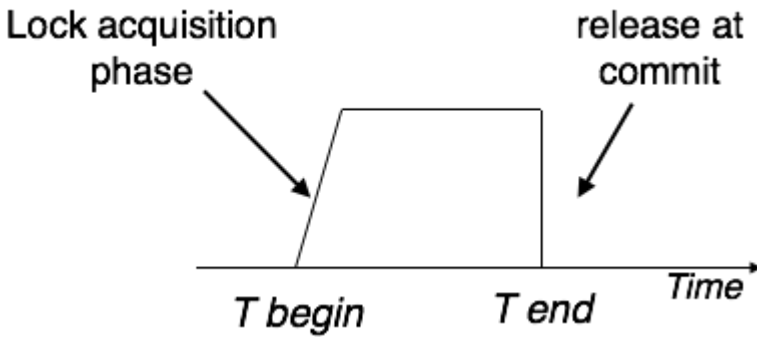


Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

### Strict Two-Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



Strict-2PL does not have cascading abort as 2PL does.

### Timestamp-based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last 'read and write' operation was performed on the data item.

### Timestamp Ordering Protocol

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction  $T_i$  is denoted as  $TS(T_i)$ .
- Read time-stamp of data-item  $X$  is denoted by  $R\text{-timestamp}(X)$ .
- Write time-stamp of data-item  $X$  is denoted by  $W\text{-timestamp}(X)$ .

Timestamp ordering protocol works as follows –

- **If a transaction  $T_i$  issues a read( $X$ ) operation –**
  - If  $TS(T_i) < W\text{-timestamp}(X)$ 
    - Operation rejected.
  - If  $TS(T_i) \geq W\text{-timestamp}(X)$ 
    - Operation executed.

- All data-item timestamps updated.
- **If a transaction  $T_i$  issues a write( $X$ ) operation –**
  - If  $TS(T_i) < R\text{-timestamp}(X)$ 
    - Operation rejected.
  - If  $TS(T_i) < W\text{-timestamp}(X)$ 
    - Operation rejected and  $T_i$  rolled back.
  - Otherwise, operation executed.

#### Thomas' Write Rule

This rule states if  $TS(T_i) < W\text{-timestamp}(X)$ , then the operation is rejected and  $T_i$  is rolled back.

Time-stamp ordering rules can be modified to make the schedule view serializable.

Instead of making  $T_i$  rolled back, the 'write' operation itself is ignored.

## Database Recovery Techniques

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss. There are mainly two types of recovery techniques used in DBMS:

**Rollback/Undo Recovery Technique:** The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.

**Commit/Redo Recovery Technique:** The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.

In addition to these two techniques, there is also a third technique called checkpoint recovery. Checkpoint recovery is a technique used to reduce the recovery time by periodically saving the state of the database in a checkpoint file. In the event of a failure, the system can use the checkpoint file to restore the database to the most recent consistent state before the failure occurred, rather than going through the entire log to recover the database.

Overall, recovery techniques are essential to ensure data consistency and availability in DBMS, and each technique has its own advantages and limitations that must be considered in the design of a recovery system.

**Database systems**, like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transaction are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database. There are both automatic and non-automatic ways for both, backing up of data and recovery from any failure situations. The techniques used to recover the lost data due to system crashes, transaction errors, viruses, catastrophic failure, incorrect commands execution, etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred update and immediate update or backing up data can be used. Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur during the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.

- The log is kept on disk **start\_transaction(T)**: This log entry records that transaction T starts the execution.
- **read\_item(T, X)**: This log entry records that transaction T reads the value of database item X.
- **write\_item(T, X, old\_value, new\_value)**: This log entry records that transaction T changes the value of the database item X from old\_value to new\_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.
- **commit(T)**: This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- **abort(T)**: This records that transaction T has been aborted.
- **checkpoint**: Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in a consistent state, and all the transactions were committed.

A transaction T reaches its **commit** point when all its operations that access the database have been executed successfully i.e. the transaction has reached the point at which it will not **abort** (terminate without completing). Once committed, the transaction is permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, item is searched back in the log for all transactions T that have written a **start\_transaction(T)** entry into the log but have not written a **commit(T)** entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

- **Undoing** – If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry **write\_item(T, x, old\_value, new\_value)** and set the value of item x in the database to old-value. There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates.
- **Deferred update** – This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm**
- **Immediate update** – In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm**.

- **Caching/Buffering** – In this one or more disk pages that include data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. A collection of in-memory buffers called the DBMS cache is kept under the control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A dirty bit is associated with each buffer, which is 0 if the buffer is not modified else 1 if modified.
- **Shadow paging** – It provides atomicity and durability. A directory with n entries is constructed, where the ith entry points to the ith database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to the original are updated to refer new replacement page.
- **Backward Recovery** – The term “Rollback ” and “UNDO” can also refer to backward recovery. When a backup of the data is not available and previous modifications need to be undone, this technique can be helpful. With the backward recovery method, unused modifications are removed and the database is returned to its prior condition. All adjustments made during the previous traction are reversed during the backward recovery. In another word, it reprocesses valid transactions and undoes the erroneous database updates.
- **Forward Recovery** – “Roll forward “and “REDO” refers to forwarding recovery. When a database needs to be updated with all changes verified, this forward recovery technique is helpful.  
Some failed transactions in this database are applied to the database to roll those modifications forward. In another word, the database is restored using preserved data and valid transactions counted by their past saves.

Some of the backup techniques are as follows:

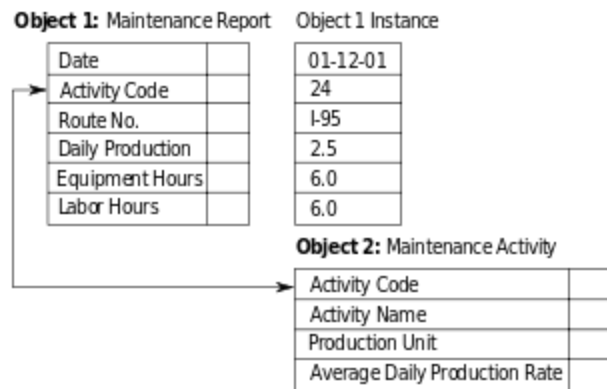
- **Full database backup** – In this full database including data and database, Meta information needed to restore the whole database, including full-text catalogs are backed up in a predefined time series.
- **Differential backup** – It stores only the data changes that have occurred since the last full database backup. When some data has changed many times since last full database backup, a differential backup stores the most recent version of the changed data. For this first, we need to restore a full database backup.
- **Transaction log backup** – In this, all events that have occurred in the database, like a record of every single statement executed is backed up. It is the backup of transaction log entries and contains all transactions that had happened to the database. Through this, the database can be recovered to a specific point in time. It is even possible to perform a backup from a transaction log if the data files are destroyed and not even a single committed transaction is lost.



## Object and Object-Relational Databases

An **object–relational database (ORD)**, or **object–relational database management system (ORDBMS)**, is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with pure relational systems, it supports extension of the data model with custom data types and methods.

### Object-Oriented Model



Example of an object-oriented database model<sup>[1]</sup>

An object–relational database can be said to provide a middle ground between relational databases and object-oriented databases. In object–relational databases, the approach is essentially that of relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSes in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying.

In an object–relational database, one might see something like this, with user-defined data-types and expressions such as `BirthDay()`:

```
CREATE TABLE Customers (  
  Id      Cust_Id  NOT NULL PRIMARY KEY,  
  Name    PersonName NOT NULL,  
  DOB     DATE     NOT NULL  
);  
SELECT Formal( C.Id )  
FROM Customers C  
WHERE BirthDay ( C.DOB ) = TODAY;
```

The object–relational model can offer another advantage in that the database can make use of the relationships between data to easily collect related records. In an address book application, an additional table would be added to the ones above to hold zero or more addresses for each customer.

Using a traditional RDBMS, collecting information for both the user and their address requires a "join":

```
SELECT InitCap(C.Surname) || ', ' || InitCap(C.FirstName), A.city  
FROM Customers C join Addresses A ON A.Cust_Id=C.Id -- the join  
WHERE A.city="New York"
```

The same query in an object–relational database appears more simply:

```
SELECT Formal( C.Name )  
FROM Customers C  
WHERE C.address.city="New York" -- the linkage is 'understood' by the ORDB
```

## **Database Security and Authorization**

Database security refers to the range of tools, controls, and measures designed to establish and preserve database confidentiality, integrity, and availability. This article will focus primarily on confidentiality since it's the element that's compromised in most data breaches.

Database security must address and protect the following:

- The data in the database
- The database management system (DBMS)
- Any associated applications
- The physical database server and/or the virtual database server and the underlying hardware
- The computing and/or network infrastructure used to access the database

Common threats and challenges

Many software misconfigurations, vulnerabilities, or patterns of carelessness or misuse can result in breaches. The following are among the most common types or causes of database security attacks and their causes.

Insider threats

An insider threat is a security threat from any one of three sources with privileged access to the database:

- A malicious insider who intends to do harm
- A negligent insider who makes errors that make the database vulnerable to attack
- An infiltrator—an outsider who somehow obtains credentials via a scheme such as phishing or by gaining access to the credential database itself

Insider threats are among the most common causes of database security breaches and are often the result of allowing too many employees to hold privileged user access credentials.

Human error

Accidents, weak passwords, password sharing, and other unwise or uninformed user behaviors continue to be the cause of nearly half (49%) of all reported data breaches.

## Exploitation of database software vulnerabilities

Hackers make their living by finding and targeting vulnerabilities in all kinds of software, including database management software. All major commercial database software vendors and open source database management platforms issue regular security patches to address these vulnerabilities, but failure to apply these patches in a timely fashion can increase your exposure.

### SQL/NoSQL injection attacks

A database-specific threat, these involve the insertion of arbitrary SQL or non-SQL attack strings into database queries served by web applications or HTTP headers. Organizations that don't follow secure web application coding practices and perform regular vulnerability testing are open to these attacks.

### Buffer overflow exploitations

Buffer overflow occurs when a process attempts to write more data to a fixed-length block of memory than it is allowed to hold. Attackers may use the excess data, stored in adjacent memory addresses, as a foundation from which to launch attacks.

### Malware

Malware is software written specifically to exploit vulnerabilities or otherwise cause damage to the database. Malware may arrive via any endpoint device connecting to the database's network.

### Attacks on backups

Organizations that fail to protect backup data with the same stringent controls used to protect the database itself can be vulnerable to attacks on backups.

These threats are exacerbated by the following:

- **Growing data volumes:** Data capture, storage, and processing continues to grow exponentially across nearly all organizations. Any data security tools or practices need to be highly scalable to meet near and distant future needs.
- **Infrastructure sprawl:** Network environments are becoming increasingly complex, particularly as businesses move workloads to multicloud or hybrid cloud architectures, making the choice, deployment, and management of security solutions ever more challenging.
- **Increasingly stringent regulatory requirements:** The worldwide regulatory compliance landscape continues to grow in complexity, making adhering to all

mandates more difficult.

- **Cybersecurity skills shortage:** Experts predict there may be as many as 8 million unfilled cybersecurity positions by 2022.

### Denial of service (DoS/DDoS) attacks

In a denial of service (DoS) attack, the attacker deluges the target server—in this case the database server—with so many requests that the server can no longer fulfill legitimate requests from actual users, and, in many cases, the server becomes unstable or crashes.

In a distributed denial of service attack (DDoS), the deluge comes from multiple servers, making it more difficult to stop the attack.

When evaluating database security in your environment to decide on your team's top priorities, consider each of the following areas:

- **Physical security:** Whether your database server is on-premise or in a cloud data center, it must be located within a secure, climate-controlled environment. (If your database server is in a cloud data center, your cloud provider will take care of this for you.)
- **Administrative and network access controls:** The practical minimum number of users should have access to the database, and their permissions should be restricted to the minimum levels necessary for them to do their jobs. Likewise, network access should be limited to the minimum level of permissions necessary.
- **End user account/device security:** Always be aware of who is accessing the database and when and how the data is being used. Data monitoring solutions can alert you if data activities are unusual or appear risky. All user devices connecting to the network housing the database should be physically secure (in the hands of the right user only) and subject to security controls at all times.
- **Encryption:** ALL data—including data in the database, and credential data—should be protected with best-in-class encryption while at rest and in transit. All encryption keys should be handled in accordance with best-practice guidelines.
- **Database software security:** Always use the latest version of your database management software, and apply all patches as soon as they are issued.
- **Application/web server security:** Any application or web server that interacts with the database can be a channel for attack and should be subject to ongoing security testing and best practice management.

- **Backup security:** All backups, copies, or images of the database must be subject to the same (or equally stringent) security controls as the database itself.
- **Auditing:** Record all logins to the database server and operating system, and log all operations performed on sensitive data as well. Database security standard audits should be performed regularly.

Today, a wide array of vendors offer data protection tools and platforms.

### Authorization

Authorization is a privilege provided by the Database Administrator. Users of the database can only view the contents they are authorized to view. The rest of the database is out of bounds to them.

The different permissions for authorizations available are:

- **Primary Permission** - This is granted to users publicly and directly.
- **Secondary Permission** - This is granted to groups and automatically awarded to a user if he is a member of the group.
- **Public Permission** - This is publicly granted to all the users.
- **Context sensitive permission** - This is related to sensitive content and only granted to a select user.

The categories of authorization that can be given to users are:

- **System Administrator** - This is the highest administrative authorization for a user. Users with this authorization can also execute some database administrator commands such as restore or upgrade a database.
- **System Control** - This is the highest control authorization for a user. This allows maintenance operations on the database but not direct access to data.
- **System Maintenance** - This is the lower level of system control authority. It also allows users to maintain the database but within a database manager instance.
- **System Monitor** - Using this authority, the user can monitor the database and take snapshots of it.