

CSCI 585 Fall 2023 - MIDTERM RUBRICS

5 points each

Q1:

a (1 point). In what sense is a PK value in each row in a table, like a 'pointer'?

b (2 points). What would happen if there is a referential integrity violation, in a pair of connected tables?

c (2 points). A table could be created, without a PK - how might this happen? If we left it as-is (ie. without a PK), would the table be populatable with ****data****?

- a. *A primary key is one of the columns from the table which acts as a unique identifier which is used as the primary index for each row in the table. The column should contain the unique value for each of the rows such that no two rows have the same value of the primary key. Primary key also helps to relate with other tables in the schema. Pointer and primary key both are used to specify/indicate something. (Any record in case of DB and memory address in case of pointers). The PK, being unique, serves to functionally determine, ie. lookup, the rest of the (non prime) attributes.*
- b. *If there is referential integrity violation, in a pair of connected tables then it may lead to data loss, data inconsistency and anomalies. There are three primary causes of referential integrity constraint violation. They are as follows:*
- Insertion of new tuples in a referencing relation in a database*
 - Deletion of some existing tuples from a referenced relation in the database*
 - Update or changes in the values of the existing tuples in a referenced relation in the database*

Applying the above modifications to the relations in the database may violate the constraints of the database.

The referential integrity violation would lead to PK not being determinable.

- c. *Yes, a table can be created without using a Primary Key. This can be done using certain db engines functions which give unique indexes to certain columns. (For example `AUTO_INCREMENT` in phpmyadmin). Bulk copying of a table into a new one can also be a source of missing PK, eg. as shown in slide#34 in <https://bytes.usc.edu/cs585/f23-Da-taaa/lectures/SQL/slides.html>*
- Yes, the data can be populated in the table without the use of a primary key. However,*

*we should take care of data consistency, data integrity while adding the data into the table, ie. **avoid creating rows with duplicate or null PK.***

GPT:

a) A PK (Primary Key) value in each row of a table is like a "pointer" in the sense that it uniquely identifies that row within the table. Just like how a pointer in programming language refers to a specific memory location, a PK value points to a specific row of data in the database table. This uniqueness is essential because it allows you to retrieve, update, or delete specific rows without ambiguity.

b) If there is a referential integrity violation in a pair of connected tables, it means that the data in one table references data in another table, but the referenced data does not exist in the referenced table. This can happen in the following ways:

- Insertion Violation: When you try to insert a record into a table that contains a foreign key (FK) that references another table, and the referenced value does not exist in the referenced table.
- Update Violation: When you try to update a record in the referenced table, and this update would cause the referenced value to change in a way that breaks the integrity of the relationship.
- Deletion Violation: When you try to delete a record from the referenced table that is still being referenced by records in the referencing table.

When a referential integrity violation occurs, the database management system (DBMS) will typically prevent the operation that would cause the violation (e.g., it may reject an INSERT or UPDATE statement) or cascade the action to maintain integrity (e.g., delete related records or set foreign key values to NULL, depending on the configured rules). It enforces data consistency and ensures that the relationships between tables are maintained.

c) A table could be created without a PK in several scenarios:

Design Choice: In some cases, database designers may intentionally choose not to define a PK for a table. This might happen when the table is a join table for a many-to-many relationship or when the data is non-relational or not naturally suitable for a PK.

Historical Data: Tables that store historical data or logs may not have a traditional PK. Instead, they might use a combination of timestamp and other attributes for identifying records uniquely.

Temporary Tables: Temporary tables or staging tables used for data import/export or intermediate processing may not have PKs because they are short-lived and don't require indexing for efficient querying.

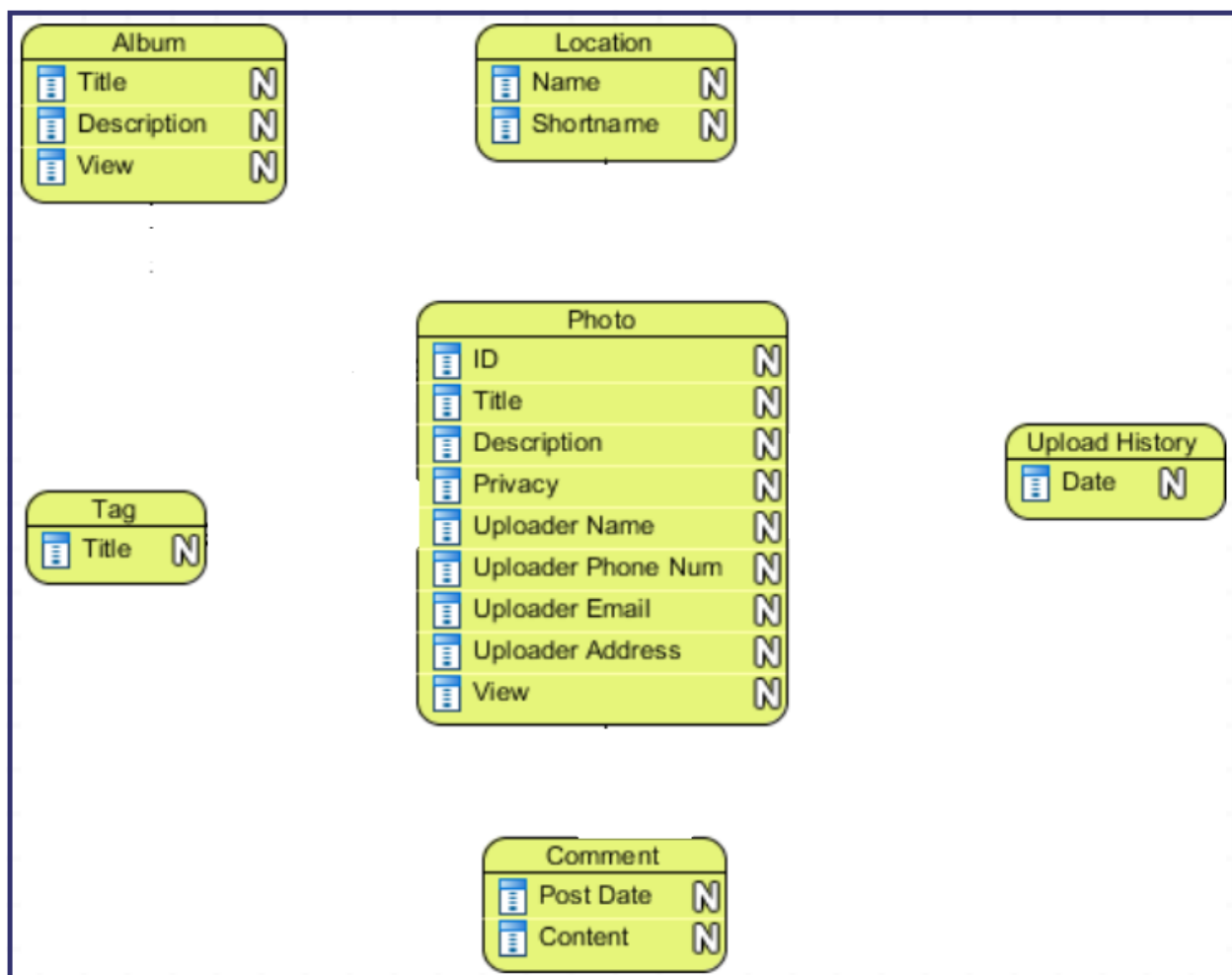
Data Warehouses: In some data warehousing scenarios, tables used for analytical purposes might not have PKs because they are optimized for read-heavy operations and may not require the same level of data integrity checks as transactional tables.

Whether a table can be populated with data without a PK depends on the specific use case and design of the table. In many cases, it is possible to insert data into a table without a PK. However, having a PK is generally recommended for maintaining data integrity and facilitating

efficient data retrieval and manipulation. The choice of whether to have a PK or not should be based on the specific requirements of the application and the nature of the data being stored.

Q2:

Photo-uploading (eg. on Insta!) would involve the following entities and columns for storing ****data****:



The 'Photo' entity is connected to the other 5 - for each of those, indicate (mention) what kind of connectivity exists.

+1 for every correct answer

1. *Photo-Album (Many to many): One photo can be part of multiple albums and one album can have many photos*
2. *Photo-Location (Many to one): Many photos can be clicked at the same location, but each photo can be clicked at one and only one location*
3. *Photo-Upload history (One to one): Each upload is recorded in history, and each photo has only one history (assuming editing after posting is not allowed)*

OR

Photo-Upload history (One to many): A photo can have multiple entries in the history, for every time the photo is edited, but a specific entry will be associated with only one photo (if editing the photo is allowed)

4. *Photo-Comment(One to Many): Each photo can have multiple comments but each comment will belong to one and only one photo*
5. *Photo-Tag(Many to Many): Each photo can have multiple tags and each tag can belong to multiple photos*

GPT answers:

Answer 1:

- Connected to **Album** through a relationship where a photo can belong to one or more albums. This is usually a many-to-many relationship, as an album can contain multiple photos, and a photo can belong to multiple albums.
- Connected to **Location** to associate a photo with a specific location. This could be a one-to-one or many-to-one relationship, depending on whether each photo is taken at a unique location.
- Connected to **Tag** through a many-to-many relationship. A photo can have multiple tags, and each tag can be associated with multiple photos.
- Connected to **Upload History** to track when the photo was uploaded. This is likely a one-to-many relationship, as a photo can have multiple upload history records (if it's edited or re-uploaded).
- Connected to **Comment** through a one-to-many relationship. A photo can have multiple comments, but each comment is associated with only one photo.

Answer 2:

- Connected to Album through a Many-to-Many relationship. An individual photo can belong to multiple albums, and an album can contain multiple photos.
- Connected to Location through a Many-to-One relationship. Many photos can be taken at the same location, but each photo is associated with only one location.
- Connected to Tag through a Many-to-Many relationship. A photo can have multiple tags, and a tag can be associated with multiple photos.
- Connected to Upload history through a One-to-One relationship. Each photo upload is recorded in the upload history, and a photo has a single upload history entry.
- Connected to Comment through a One-to-Many relationship. A photo can have multiple comments, but each comment is associated with only one photo.

Q3:

Normalization is about streamlining **data**** storage.**

EMPLOYEE_ID	NAME	JOB_CODE	JOB	STATE_CODE	HOME_STATE
E001	Alice	J01	Chef	26	Michigan
E001	Alice	J02	Waiter	26	Michigan
E002	Bob	J02	Waiter	56	Wyoming
E002	Bob	J03	Bartender	56	Wyoming
E003	Alice	J01	Chef	56	Wyoming

Consider the table shown above.

a (1 point). What would you do, to make the table be in 1NF?

b (2 points). What would be the result, when you do 1NF -> 2NF?

c (2 points). How about when we do 2NF -> 3NF?

Solution

a) The table is already in 1NF as all values are atomic.

Primary Key = (EmployeeID, Job Code)

Grading

- Award 1 point if the student identifies that the table is already in 1NF.*
- -1 point : if the student fails to identify.*

b) Eliminating partial dependencies

<i>Employee ID</i>	<i>Name</i>	<i>State_Code</i>	<i>Home State</i>
<i>E001</i>	<i>Alice</i>	<i>26</i>	<i>Michigan</i>
<i>E002</i>	<i>Bob</i>	<i>56</i>	<i>Wyoming</i>
<i>E003</i>	<i>Alice</i>	<i>56</i>	<i>Wyoming</i>

<i>Job Code</i>	<i>Job</i>
<i>J01</i>	<i>Chef</i>
<i>J02</i>	<i>Waiter</i>
<i>J03</i>	<i>Bartender</i>

<i>Employee ID</i>	<i>Job Code</i>
<i>E001</i>	<i>J01</i>
<i>E001</i>	<i>J02</i>
<i>E002</i>	<i>J02</i>
<i>E002</i>	<i>J02</i>
<i>E003</i>	<i>J03</i>

Grading:

- *Award points if student eliminates all partial dependencies (reference above)*
- *Award points even when student eliminates transitive dependency along with partial dependencies (reference sol c below), as a 3NF is always a 2NF*
- *-1 if any partial dependency elimination is missed.*

c) Eliminating transitive dependency

<i>Employee ID</i>	<i>Name</i>	<i>State_Code</i>
<i>E001</i>	<i>Alice</i>	<i>26</i>
<i>E002</i>	<i>Bob</i>	<i>56</i>
<i>E003</i>	<i>Alice</i>	<i>56</i>

<i>Job Code</i>	<i>Job</i>
<i>J01</i>	<i>Chef</i>
<i>J02</i>	<i>Waiter</i>
<i>J03</i>	<i>Bartender</i>

<i>State_code</i>	<i>Home_State</i>
<i>26</i>	<i>Michigan</i>
<i>56</i>	<i>Wyoming</i>

<i>Employee ID</i>	<i>Job Code</i>
<i>E001</i>	<i>J01</i>
<i>E001</i>	<i>J02</i>
<i>E002</i>	<i>J02</i>
<i>E002</i>	<i>J02</i>
<i>E003</i>	<i>J03</i>

Grading

- *Award points when all partial and transitive dependencies are eliminated.*
- *-1 if any partial dependency elimination is missed.*
- *-1 if transitive dependency elimination is missed.*

GPT Answers:

Answer 1:

a. To make the table be in 1NF (First Normal Form):

- Each table cell should have a single value.
- Entries in a column are of the same kind.
- Each column must have a unique name.
- The order in which data is saved does not matter.

Given the table, it already appears to be in 1NF. There are no repeating groups or arrays, and each cell has a single value.

b. 1NF to 2NF (Second Normal Form):

- Remove subsets of data that apply to multiple rows of a table and place them in separate tables.
- Create relationships between these new tables and their predecessors through the use of foreign keys.

For our table, the `JOB_CODE`, `JOB`, and possibly `STATE_CODE` and `HOME_STATE` could be split into separate reference tables, because multiple employees can have the same job or come from the same state.

c. 2NF to 3NF (Third Normal Form):

- Remove columns that are not dependent upon the primary key.

Considering our table and its potential 2NF form, the `HOME_STATE` is dependent on the `STATE_CODE` and not on the `EMPLOYEE_ID`. Thus, in 3NF, the `HOME_STATE` column would be moved to a separate table where `STATE_CODE` is the primary key. This ensures that the `HOME_STATE` is functionally dependent only on `STATE_CODE`.

Answer 2:

a. To ensure the table is in 1NF:

- Every column should contain atomic (indivisible) values.
- Every column should contain values of the same data type.
- No two rows may have repeating values across columns.
- There should be a primary key that uniquely identifies each row.

Observing the provided table, it already adheres to the 1NF standards as each cell consists of a single value and there aren't any repeating groups.

b. Transitioning from 1NF to 2NF:

- 2NF aims to ensure that all non-key attributes are fully functionally dependent on the primary key. This means that there is no partial dependency of any column on the primary key.
- If we consider `EMPLOYEE_ID` and `JOB_CODE` as a composite primary key, then attributes like `NAME` and `HOME_STATE` are partially dependent only on `EMPLOYEE_ID`. Hence, we can move these to a separate table with `EMPLOYEE_ID` as the primary key.
- This results in two tables: one for employee details and another for job details.

c. Moving from 2NF to 3NF:

- 3NF mandates that every non-primary attribute should be functionally dependent only on the primary key.
- From the table in 2NF, we can observe that `HOME_STATE` is determined by `STATE_CODE`. This is a transitive dependency in the context of the primary key. Thus, to achieve 3NF, we create a separate table for states where `STATE_CODE` is the primary key.

Q4:

In the relational ****data**** model, we use tables.

a (1 point). What is your own example of intersecting two tables (e.g. using the INTERSECT SQL command)?

b (2 points). What is your own example of a quaternary bridge table?

c (2 points). What is your own example for GROUP BY along with HAVING?

Grading:

(a) Intersecting Two Tables -

- 0.5 point: Fails to provide a clear explanation around the example using tables, query, etc. Query is not required for this question.

***NOTE:** INTERSECT is not the only valid answer, there are other ways too, award marks for all such relevant ways.*

(b) Quaternary Bridge Table -

- 1 point: Fails to provide a clear explanation of the example using tables and FKs or does not mention FKs at all. The relationships and purpose of the quaternary bridge table are not well described. Query is not required for this question. We can accept an any answer that lists a set of 4 columns (a,b,c,d) where each column would be a PK in a different table, eg. (BookID,AuthorID,PublisherID,SubjectID).

(c) GROUP BY along with HAVING -

- 1 point: Does not provide a SQL query for the example using GROUP BY and HAVING, the syntax of the Query is incorrect or if Only one of two clauses are demonstrated well.

General Criteria for All Parts:

***0 point:** Award zero points for a particular section if there is no example provided or the example is entirely unrelated or incomprehensible.*

***0 point:** Check individually for each part. If any part of the answer resembles solutions provided by ChatGPT completely, the answer will be awarded zero points for that part. Originality is expected in examples.*

ChatGPT Answer - 1

(a) Intersecting Two Tables:

Let's say we have two tables in a relational database: "Employees" and "Projects." The "Employees" table contains information about employees, including their ID and name, while the "Projects" table contains information about various projects, including the project ID and name. We can use the INTERSECT SQL command to find employees who are working on specific projects. Here's an example query:

```
```sql
SELECT Employees.Name
FROM Employees
WHERE Employees.ID IN (
 SELECT EmployeeID
 FROM Employee_Projects
 WHERE Employee_Projects.ProjectID = (
 SELECT ProjectID
 FROM Projects
 WHERE Projects.Name = 'ProjectA'
)
)
)
INTERSECT
SELECT Employees.Name
FROM Employees
WHERE Employees.ID IN (
 SELECT EmployeeID
 FROM Employee_Projects
 WHERE Employee_Projects.ProjectID = (
 SELECT ProjectID
 FROM Projects
 WHERE Projects.Name = 'ProjectB'
)
)
);
```
```

In this example, we are finding employees who are working on both "ProjectA" and "ProjectB."

(b) Example of quaternary bridge table:

A quaternary bridge table is used to establish a many-to-many relationship between four entities. Let's consider an example involving four entities: "Students," "Courses," "Professors," and "Departments." We want to track which students are enrolled in which courses, taught by which professors, in which departments. We can create a quaternary bridge table called "Student_Course_Professor_Department" to represent this relationship. Here's a simplified schema for such a table:

```
```sql
CREATE TABLE Student_Course_Professor_Department (
 EnrollmentID INT PRIMARY KEY,
```

```
StudentID INT,
CourseID INT,
ProfessorID INT,
DepartmentID INT,
EnrollmentDate DATE,
Grade CHAR(2)
);
...
```

In this table, "EnrollmentID" is the primary key, and it establishes the relationship between students, courses, professors, and departments. Each row in this table represents a student's enrollment in a course taught by a specific professor in a particular department.

### (c) Example of GROUP BY with HAVING:

Let's assume we have a table named "Orders" that contains information about customer orders, including order IDs, customer IDs, and order totals. We want to find customers who have placed orders with a total order value greater than \$1,000. We can use the GROUP BY clause along with HAVING to achieve this. Here's an example query:

```
```sql  
SELECT Customers.CustomerName, SUM(Orders.OrderTotal) AS TotalOrderValue  
FROM Customers  
JOIN Orders ON Customers.CustomerID = Orders.CustomerID  
GROUP BY Customers.CustomerName  
HAVING SUM(Orders.OrderTotal) > 1000;  
...
```

In this example, we first join the "Customers" and "Orders" tables using the customer ID. Then, we use GROUP BY to group the results by customer name and calculate the total order value for each customer. Finally, the HAVING clause filters out customers whose total order value is not greater than \$1,000, giving us a list of customers who meet the specified condition.

ChatGPT Answer - 2

(a) Intersecting Two Tables:

Let's say we have two tables: `Employees` and `Departments`. We want to find employees who work in both the "Sales" and "Marketing" departments. We can use the `INTERSECT` SQL command to achieve this.

```
```sql  
SELECT EmployeeName
FROM Employees
WHERE Department = 'Sales'
```

## INTERSECT

```
SELECT EmployeeName
FROM Employees
WHERE Department = 'Marketing';
...
```

This query will return the names of employees who are in both the Sales and Marketing departments.

### (b) Example of Quaternary Bridge Table:

A quaternary bridge table is used to establish a many-to-many relationship between four entities. Let's consider an example involving four entities: Students, Courses, Professors, and Semesters. We want to track which students are enrolled in which courses for specific semesters, taught by particular professors. We can create a quaternary bridge table called `Enrollments` to represent this relationship.

#### **`Enrollments` Table:**

```
...
| EnrollmentID | StudentID | CourseID | ProfessorID | Semester |
|-----|-----|-----|-----|-----|
| 1 | 101 | 1 | 201 | Spring2023 |
| 2 | 102 | 2 | 202 | Fall2022 |
| 3 | 103 | 3 | 201 | Spring2023 |
...
```

In this table, each row represents a student's enrollment in a specific course, taught by a particular professor, during a particular semester. The `EnrollmentID` is a unique identifier for each enrollment.

### (c) Example of GROUP BY along with HAVING:

Let's consider a scenario where we have a table named `Orders` that contains information about customer orders, including the order total. We want to find customers who have placed orders with a total value greater than \$500. We can use the `GROUP BY` clause along with the `HAVING` clause to achieve this.

#### **`Orders` Table:**

```
...
| OrderID | CustomerID | OrderTotal |
|-----|-----|-----|
| 1 | 101 | 450 |
| 2 | 102 | 600 |
| 3 | 103 | 300 |
...
```

```
| 4 | 101 | 700 |
...
```

```
```sql  
SELECT CustomerID, SUM(OrderTotal) AS TotalSpent  
FROM Orders  
GROUP BY CustomerID  
HAVING SUM(OrderTotal) > 500;  
...
```

This query will group the orders by customer and calculate the total amount spent by each customer. Then, it will filter the results using the `HAVING` clause to only include customers whose total spending is greater than \$500. The result will show the customer IDs and their total spending above the specified threshold.

Q5:

a (2 points). How can deadlocks arise, in the 2PL scheme for **data** access during transactions? Explain, via an example.

-1 point: *Explanation is unrelated.*

-0.5 point: *No use of keywords from 2PL scheme.*

-0.5 point: *Example not related to 2PL scheme.*

Sample answer: (Many explanations can exist, consider all)

*Deadlocks can arise in the **growing phase** of the 2PL scheme when both transactions need to acquire a lock for the same variables. For example: $T1 = R[x] \rightarrow W[y]$ and $T2 = W[y] \rightarrow R[x]$: Here, $T1$ gets the read lock for x , and $T2$ gets the write lock for y and for the next operation, $T1$ and $T2$ are in a deadlock.*

ChatGPT answers:

- 1) Deadlocks can arise in the Two-Phase Locking (2PL) scheme for data access during transactions when multiple transactions acquire and hold locks on data items in a way that creates a circular dependency in the lock acquisition graph. This circular dependency prevents any of the involved transactions from making progress, resulting in a deadlock.
- 2) Deadlocks can manifest within the Two-Phase Locking (2PL) framework for data access during transactions due to the establishment of a circular dependency. In this context, a deadlock emerges when multiple transactions secure and maintain locks on data items, leading to a situation where they are mutually waiting for each other to relinquish the necessary locks. This impasse leads to a complete standstill, rendering all affected transactions unable to advance in their execution.

b (3 points). When we release locks in 2PL, what would necessitate us to hold on to all the locks and release them together? Explain.

+1.5: *Achieve "ACID" of a transaction(or mention similar concepts like isolation and serializability). Specifically, ATOMICITY can be guaranteed by doing so.*

+1.5:*If a lock is released before all the locks are acquired, and two concurrent transactions $T1$ and $T2$ try to read and write the same value, it could lead to a lost update error. (give one or more valid examples to explain the reason. Like: lost update/reading uncommitted data/improper retrieval. -0.25 for not mentioning keyword or similar concept like " concurrent transaction").*

ChatGPT answer:

In the Two-Phase Locking (2PL) protocol, there are two distinct phases:

Growing Phase: During this phase, a transaction can acquire any number of locks but cannot release any. This phase continues until the transaction acquires its final lock.

Shrinking Phase: After a transaction has acquired its last lock and entered the shrinking phase, it can release locks but cannot acquire any new ones.

Your question pertains to the conditions under which a transaction would retain all its locks until the very end, releasing them all at once, rather than gradually releasing them during the Shrinking Phase.

The decision to hold onto locks until the very end and release them all at once can be motivated by several factors:

Simplicity in Implementation: If locks are released all at once at the end of the transaction, then there is no need for intricate logic to manage partial releases. This could make the locking mechanism's code simpler to write, understand, and maintain.

Avoiding Phantom Reads: By holding onto locks for the duration of the transaction, we can ensure that the state of the data being worked upon does not change. Releasing locks earlier could lead to "phantom reads," where new rows that match a previous condition can be inserted by another transaction.

Higher Commit Chances: If a transaction releases its locks gradually, there's a risk that another transaction could acquire a lock, update the data, and then force the first transaction to abort due to some inconsistency or conflict. By holding onto locks until all operations are complete, we might increase the chance that the transaction can commit successfully.

Short Transactions: For transactions that are expected to complete quickly, the time frame between acquiring the last lock and completing the transaction might be so short that there's little benefit in releasing locks early.

Ensuring Atomicity: One of the properties of transactions is atomicity, which means that a transaction's operations are either fully completed or fully undone. By holding onto locks, we can help ensure atomicity since other transactions won't be able to interfere until our transaction is done.

Q6:

a (2 points). With regards to ****data**** access, what was the 'CAP theorem' about?

b (3 points). What is CAP's modern equivalent? Explain, using a few sentences.

a) *+1 Point if mentioned Consistency, Availability, Partition Tolerance (-0.25 for each one missing).*

+1 Point for mentioning only two properties can be chosen at a given time with correct explanation.

b) *The modern equivalent of the CAP theorem is 'CA' [without the P] - in other words, a tradeoff between C and A. It's **NOT** about 'ACID' or 'BASE' at all.*

LLM Derived Answers:

a) The CAP theorem, also known as Brewer's theorem, is a concept in computer science and distributed systems that deals with the trade-offs between three key properties in a distributed data store: Consistency, Availability, and Partition Tolerance. The CAP theorem was formulated by computer scientist Eric Brewer in 2000.

Consistency: This property ensures that all nodes in a distributed system see the same data at the same time. In other words, if you write data to one node and immediately read it from another, you should always get the most recent write.

Availability: Availability means that every request to the system receives a response, without guaranteeing that it's the most recent data. In other words, the system remains operational and responsive, even if some of its nodes fail.

Partition Tolerance: Partition tolerance deals with the system's ability to continue functioning even when network partitions occur. Network partitions can happen when some communication links between nodes in a distributed system break, and nodes are unable to communicate with each other.

The CAP theorem asserts that in a distributed system, you can achieve at most two out of the three properties simultaneously, but not all three. This means that you have to make trade-offs depending on your system's requirements and the nature of the failure scenarios you expect to encounter. Here are the common trade-offs:

CA: If you prioritize Consistency and Availability, your system may not be able to tolerate network partitions. In the event of a network partition, you might have to sacrifice either Consistency or Availability.

CP: If you prioritize Consistency and Partition Tolerance, your system may not be as highly available, especially during network partitions. This is often the choice for systems that require strong data consistency and can tolerate some temporary unavailability.

AP: If you prioritize Availability and Partition Tolerance, your system may not guarantee strong Consistency. In this case, you might have eventual consistency, meaning that given enough time, all nodes will converge to the same state.

It's important to note that the CAP theorem is not a strict rule but rather a guideline for designing and understanding the behavior of distributed systems. In practice, various distributed databases and systems make different trade-offs depending on their specific use cases and requirements.

b) The CAP theorem still holds as a relevant concept for understanding the fundamental trade-offs in distributed systems. However, in more recent discussions, it has been suggested that the CAP theorem can be complemented by an "AP/EC" (Availability/Partition tolerance and Eventual Consistency) perspective, which provides a more nuanced view of the trade-offs. This acknowledges that in practical scenarios, it's often more about choosing the right trade-off between Availability, Partition Tolerance, and Consistency based on specific system and application requirements. This perspective recognizes that systems can offer varying degrees of consistency, availability, and partition tolerance, and it's essential to tailor the trade-offs to meet the specific needs of the application.

OR

The BASE theorem is an extension or alternative perspective to the CAP theorem and is often used in the context of NoSQL databases and distributed systems. BASE stands for:

1. **Basically Available**: This aligns with the "A" in CAP. It means that the system remains available for read and write operations even in the presence of faults or network partitions. It prioritizes high availability over strong consistency.
2. **Soft state**: Unlike the "C" (Consistency) in CAP, BASE allows for the system to exhibit soft state or eventual consistency. This means that the system may not provide strong consistency guarantees at all times, but consistency is achieved over time as the system converges.
3. **Eventual consistency**: This corresponds to the eventual consistency aspect of the CAP theorem. It means that, given enough time and absence of new updates, all replicas in the system will eventually converge to the same state.

So, in the context of the BASE theorem, the modern equivalent of the CAP theorem can be seen as a trade-off between "Basically Available" and "Eventual Consistency," with a willingness to accept soft state as a compromise when dealing with distributed systems. This approach recognizes that strong consistency might not always be practical or necessary for certain applications, and achieving high availability and eventual consistency can be more valuable.

OR

PACELC Theorem: The PACELC theorem, proposed by Daniel Abadi, extends the CAP theorem to provide a more nuanced view of distributed system trade-offs. PACELC stands for "Partition tolerance, Availability, Consistency, Else (or Latency, i.e., how the system behaves under normal conditions)." It acknowledges that in real-world distributed systems, there's a spectrum of possible behaviors beyond just the binary choices of CAP.

OR

CRDTs (Conflict-free Replicated Data Types): CRDTs are data structures and algorithms designed for distributed systems, emphasizing mergeability and conflict resolution. They provide a way to achieve high availability and partition tolerance while ensuring eventual consistency.

(PACLEC/CRDT not mentioned in class, definitely LLM derived)

Q7:

a (2 points). Trace the evolution of server-side technologies, for providing connectivity to DBs. You need to mention+discuss at least four different ones.

b (3 points). How do modern apps handle ****data**** connectivity? Mention the acronym we discussed in class, and explain its components.

Solution:

(a) Student can include any kind of server-side technologies used historically, all of them should not be the latest ones. CGI, Asp.net, Ruby on Rails, Node.js are some of the examples.

*(b) In apps, requests would go to the web server. Now the data can be generated in many ways. The data can be in the form of a static database or the result of some calculations. One other way would be by function call. For example, REST API is a function call, and you can use other modern API architecture styles, such as GraphQL, on top of that. But modern apps would distribute function calls and use a newer architecture called **MCC Microservice Container Cloud**); different components of this architecture are as follows:*

- **Microservice:** One main server would get the request from the app and then break it into some smaller functions(microservices). Then it would send each function to one machine(server) to get the result.*
- **Containers:** Each microservice can be available through multiple containers. The main server can balance the load of all requests between multiple containers. This would help with scaling up or down the app based on the number of requests.*
- **Cloud:** We need a big system such as GCP or Azure to run this whole system. So we use clouds to manage different parts of the architecture.*

Grading:

(a) +1 point for mentioning the name of technology.

+1 point for describing about the mentioned technologies.

(b)

+ 0.5 point Mentioning correct acronym - MCC (Microservice Container Cloud)

+ 2.5 points Explanation of each component (1 point explanation of Microservice - 1 point explanation of Containers - 0.5 point explanation of Cloud)

ChatGPT Answers:

(a)

Answer 1:

The evolution of server-side technologies for providing connectivity to databases has seen significant advancements over the years. Here's a brief overview of four different stages in this evolution:

Common Gateway Interface (CGI):

- Era: Late 1990s
- Description: CGI was one of the earliest methods for connecting web servers to databases. It allowed web servers to execute external programs (typically written in Perl, C, or other languages) to generate dynamic content and interact with databases.
- Pros: CGI was a significant step forward in creating dynamic web applications. It allowed for the first server-side scripting.
- Cons: CGI had performance issues as it spawned a new process for each request, making it inefficient for handling a large number of concurrent connections.

PHP (Personal Home Page, later PHP: Hypertext Preprocessor):

- Era: Late 1990s to early 2000s
- Description: PHP was one of the first scripting languages specifically designed for web development. It provided native support for interacting with databases through extensions like MySQL, PostgreSQL, and others.
- Pros: PHP was easy to learn and widely adopted for building dynamic web applications. It introduced a more efficient way to connect to databases.
- Cons: PHP had security vulnerabilities, and its code often lacked proper separation of concerns, leading to complex and unmaintainable code.

Java EE (Java Platform, Enterprise Edition):

- Era: Early 2000s to mid-2010s
- Description: Java EE was a comprehensive platform for building enterprise-level applications. It included technologies like Java Servlets and JavaServer Pages (JSP), which provided powerful database connectivity options through JDBC (Java Database Connectivity).
- Pros: Java EE offered robust, scalable, and secure solutions for database connectivity. JDBC allowed for standardized interactions with various databases.
- Cons: Java EE was complex and required a steep learning curve. It was heavyweight for smaller applications.

Node.js and the Rise of JavaScript:

- Era: Mid-2010s to present
- Description: Node.js, built on the V8 JavaScript engine, introduced server-side JavaScript development. It rapidly gained popularity for its event-driven, non-blocking I/O model, making it suitable for real-time applications and database connectivity.
- Pros: Node.js simplified development by allowing the use of a single language (JavaScript) for both client and server-side. It leveraged non-blocking I/O for high concurrency and efficiency.

- Cons: Node.js had limited core database support, which required the use of third-party libraries. The asynchronous programming model can be challenging for developers new to JavaScript.

These are just a few key milestones in the evolution of server-side technologies for connecting to databases. Since my knowledge is up to date as of September 2021, there may have been further developments in the field, with new technologies and paradigms emerging. Keep in mind that the choice of technology depends on factors like project requirements, scalability, performance, and developer expertise.

Answer 2:

Same as last one with this new item:

Ruby on Rails and ActiveRecord:

- Era: Mid-2000s - Present
- Description: Ruby on Rails is a web application framework that includes the ActiveRecord ORM (Object-Relational Mapping) component. ActiveRecord simplifies database connectivity by mapping database tables to Ruby objects and providing an expressive API for working with databases. It emphasizes convention over configuration, making development faster and more straightforward.
- Advantages: Rapid development, clean and readable code, and built-in support for common database operations.
- Challenges: Primarily suited for smaller to medium-sized applications, may require additional configuration for legacy or complex database schemas.

(b)

- Answer 1:

Modern apps handle data connectivity using a variety of techniques and technologies, depending on the platform, the type of app, and the specific use case. Here are some common approaches and technologies used in handling data connectivity in modern apps:

1. RESTful APIs: Many modern apps communicate with server-side components via Representational State Transfer (REST) APIs. These APIs use HTTP methods (e.g., GET, POST, PUT, DELETE) to send and receive data in a standardized format, typically JSON or XML. Mobile and web apps can make HTTP requests to these APIs to fetch or send data.
2. GraphQL: GraphQL is an alternative to REST that allows clients to request exactly the data they need, reducing over-fetching and under-fetching of data. It provides a more flexible and efficient way to query and manipulate data, especially in complex apps.
3. WebSocket and WebRTC: For real-time communication, apps use WebSocket for bidirectional, full-duplex communication with a server. WebSocket is often employed in chat applications and other scenarios where low-latency data exchange is crucial.

WebRTC is used for peer-to-peer real-time communication in applications like video conferencing.

4. Offline Data Sync: Many apps need to work offline or with intermittent connectivity. They use techniques like local data caching and synchronization to store data on the device and update it when a connection is available. Technologies like Service Workers, IndexedDB, and client-side databases are used for this purpose.

5. Content Delivery Networks (CDNs): To optimize content delivery and reduce latency, apps often leverage CDNs to cache and serve static assets like images, JavaScript, and CSS files from geographically distributed servers.

6. Mobile App SDKs: Mobile app developers can use software development kits (SDKs) provided by platform-specific services, such as Firebase for Android and iOS or AWS Mobile for Amazon Web Services. These SDKs offer built-in data storage, authentication, and offline capabilities.

7. Data Compression: Apps may use data compression techniques to reduce the amount of data transferred over the network. This is especially important in bandwidth-constrained environments or when dealing with large media files.

8. Reactive Programming: Some modern apps use reactive programming libraries like RxJS (for JavaScript) or RxSwift (for Swift) to handle asynchronous operations, making it easier to work with real-time data streams and events.

9. Background Sync and Push Notifications: Apps can take advantage of background sync and push notifications to keep users updated even when the app is not actively running. Background sync allows an app to periodically sync data with a server, while push notifications deliver updates to the device.

10. OAuth and OpenID Connect: Apps often use OAuth and OpenID Connect for secure user authentication and authorization. These protocols provide a standard way to authenticate users and obtain authorization to access their data.

11. Security and Encryption: To protect data in transit and at rest, apps implement encryption and security measures, such as TLS/SSL for data in transit and data encryption algorithms for data at rest.

12. Monitoring and Analytics: Apps use monitoring and analytics tools to track network performance, user behavior, and error reporting. Services like Google Analytics and New Relic are commonly used for this purpose.

Modern app development often combines several of these techniques and technologies to provide a seamless, efficient, and secure data connectivity experience for users. The choice of approach depends on the app's requirements, architecture, and the platforms it targets.

- Answer 2:

1. APIs (Application Programming Interfaces): Most apps rely on APIs to connect to remote data sources. These are sets of rules and protocols that allow one piece of software to interact with another. RESTful APIs and GraphQL are popular choices for building web-based APIs.

2. HTTP/HTTPS: Apps communicate with remote servers over the HTTP (Hypertext Transfer Protocol) or its secure version, HTTPS. This is the foundation for most web-based data connectivity. It allows apps to send and receive data via requests and responses.

3. Web Sockets: For real-time communication, many modern apps use Web Sockets. This technology allows bidirectional communication between the app and the server, making it suitable for chat applications, online gaming, and other real-time scenarios.

4. Mobile Data and Wi-Fi: Mobile apps can use the device's cellular data connection or Wi-Fi to access data. They often check for the presence of an internet connection and adapt their behavior accordingly, such as syncing data in the background when a connection is available.

5. Local Caching: To improve performance and reduce the need for constant data transfers, apps often cache data locally. This means storing a copy of data on the device so that the app can access it quickly, even without an internet connection.

6. Offline Capabilities: Some apps are designed to work offline or in a limited connectivity environment. They can store data locally, perform operations, and then sync with the server once a connection is re-established.

7. Push Notifications: Apps can use push notifications to inform users about new data or updates. These notifications are typically sent from a remote server to the app, alerting the user even when the app is not actively running.

8. Background Sync: Many apps synchronize data in the background, even when the app is not in use. This ensures that the data is up to date when the user opens the app.

9. Load Balancing and Content Delivery Networks (CDNs): To ensure high availability and reduce latency, apps often distribute their services across multiple servers and use CDNs to serve static assets, such as images and videos, from servers geographically closer to the user.

10. Authentication and Authorization: Apps use various authentication mechanisms, such as OAuth or API keys, to ensure that only authorized users can access certain data. This is essential for security and privacy.

11. Data Compression: To minimize data usage and improve performance, apps may compress data before transmitting it over the network.

12. Error Handling and Retry Strategies: Apps should be able to handle network failures gracefully, implementing retry mechanisms and error handling to deal with intermittent connectivity issues.

13. Monitoring and Analytics: Many apps use monitoring tools and analytics to track network performance, identify issues, and gather insights into user behavior and preferences.

Q8:

When it comes to **data**** processing in general, 'that's what it's all about'!**

What is it all about? Explain, using SQL syntax (it is ok if your syntax isn't perfect!).

(3 points) Create tables -> Enter data into them -> Do queries(any operation- update, getting information.)

(2 points) Sample SQL Commands:

Create table:

```
CREATE TABLE table_name (  
    column1 datatype PRIMARY KEY (THIS IS OPTIONAL),  
    column2 datatype,  
    column3 datatype,  
    ....  
);
```

Enter Data:

```
INSERT INTO table_name  
VALUES (value1, value2, value3, ...);
```

Do Queries:

Select statements(any)

Sample queries(but not limited to)

```
SELECT * FROM table1  
WHERE column2 > value;
```

Or

```
SELECT column1, column2 FROM table1 WHERE column2='value';
```

Deduct Marks:

- (0 points) If mentioned unrelated explanation
- (-3 points) If failed to explain the above 3 things(tables ← data ← queries); can give partial depending on explanation
- (-2 points) if syntax examples are not present; can give partial for SELECT keywords etc. only if relevant

LLM Answers:

GPT-4	GPT-3.5	Bard
<p>Data processing, at its core, is about transforming raw data into meaningful information. SQL (Structured Query Language) is one of the primary tools used in relational databases to process data. When you think about data processing using SQL, here's what it's all about:</p> <p>Storing Data SQL allows you to create tables to store structured data.</p> <pre>CREATE TABLE users (id INT PRIMARY KEY, name VARCHAR(255), email VARCHAR(255));</pre> <p>Retrieving Data You can query the database to fetch data based on certain criteria.</p> <pre>SELECT name, email FROM users WHERE id = 5;</pre> <p>Updating Data SQL provides mechanisms to modify existing data.</p> <pre>UPDATE users SET email = 'new.email@example.com' WHERE id = 5;</pre> <p>Deleting Data If you need to remove records, SQL can handle that too.</p> <pre>DELETE FROM users WHERE id = 5;</pre> <p>Aggregating Data SQL provides functions to summarize and analyze data.</p> <pre>SELECT AVG(salary) as AverageSalary FROM employees;</pre>	<p>Data processing in the context of databases and SQL is indeed fundamental. It's all about retrieving, manipulating, and managing data to extract valuable insights or perform necessary operations. Here's an explanation using SQL-like syntax to illustrate key concepts:</p> <p>Data Retrieval (SELECT): This is one of the fundamental operations. You use the SELECT statement to retrieve specific data from a database. For example:</p> <pre>SELECT FirstName, LastName FROM Customers WHERE Country = 'USA';</pre> <p>This SQL statement retrieves the first and last names of customers from the 'Customers' table who are located in the USA.</p> <p>Data Manipulation (INSERT, UPDATE, DELETE): Data processing involves adding, modifying, or removing data from the database.</p> <p>INSERT adds new data:</p> <pre>INSERT INTO Orders (CustomerID, OrderDate) VALUES (123, '2023-10-25');</pre> <p>UPDATE modifies existing data:</p> <pre>UPDATE Products SET Price = 19.99 WHERE ProductID = 456;</pre> <p>DELETE removes data:</p> <pre>DELETE FROM Employees WHERE EmployeeID = 789;</pre>	<p>Data processing in general is all about transforming raw data into a format that can be used to answer questions, make decisions, and gain insights. This involves a number of steps, including:</p> <p>Data collection: Gathering data from various sources, such as sensors, surveys, and transactions.</p> <p>Data cleaning: Identifying and correcting errors and inconsistencies in the data.</p> <p>Data preparation: Formatting and transforming the data into a format that is compatible with the analysis tools being used.</p> <p>Data analysis: Applying statistical and machine learning techniques to the data to extract insights.</p> <p>Data visualization: Communicating the insights gained from the analysis in a visually appealing and easy-to-understand way.</p> <p>SQL (Structured Query Language) is a powerful programming language that is used to manage and query data in relational databases. SQL can be used to perform all of the steps involved in data processing, from data cleaning and preparation to data analysis and visualization.</p> <p>Here is a simple example of how SQL can be used to process data:</p> <pre>-- Select all customers who have purchased more than \$100 worth of products in the past</pre>

Joining Data

If data is distributed across different tables, SQL provides JOIN operations to fetch combined data.

```
SELECT users.name,
orders.order_date FROM
users JOIN orders ON
users.id =
orders.user_id;
```

Ensuring Data Integrity

SQL allows for setting constraints to ensure data remains consistent and valid.

```
ALTER TABLE users ADD
CONSTRAINT chk_email
CHECK (email LIKE
' %@ %. %');
```

Optimization

Using indexing, you can speed up data retrieval operations.

```
CREATE INDEX
idx_users_email ON
users(email);
```

Security

SQL also provides mechanisms to control access to the data.

```
GRANT SELECT ON users
TO some_user;
```

Transactions

Ensures that a series of SQL statements are executed completely or not at all, preserving data integrity.

```
BEGIN; UPDATE accounts
SET balance = balance -
100 WHERE id = 1;
UPDATE accounts SET
balance = balance + 100
WHERE id = 2; COMMIT;
```

Data Filtering (WHERE): You can filter data using the WHERE clause to narrow down results based on specified conditions.

Data Sorting (ORDER BY): The ORDER BY clause helps arrange data in a specific order, such as ascending or descending.

```
SELECT ProductName,
Price FROM Products
ORDER BY Price DESC;
```

This SQL statement retrieves product names and prices from the 'Products' table, sorted by price in descending order.

Data Aggregation (GROUP BY, HAVING): SQL allows you to perform calculations on data, often used with grouping.

```
SELECT Category,
AVG(Price) AS AvgPrice
FROM Products GROUP BY
Category HAVING
AVG(Price) > 50;
```

This SQL statement calculates the average price of products in each category and only displays results where the average price is greater than 50.

Data Joins (INNER JOIN, LEFT JOIN, etc.): Data is often spread across multiple tables, and SQL provides ways to combine data from these tables.

```
SELECT Orders.OrderID,
Customers.FirstName,
Customers.LastName FROM
Orders INNER JOIN
Customers ON
Orders.CustomerID =
Customers.CustomerID;
```

```
year.
```

```
SELECT customer_id,
SUM(product_price) AS
total_spend FROM orders
WHERE order_date >=
CURRENT_DATE - INTERVAL
1 YEAR GROUP BY
customer_id HAVING
total_spend > 100;
```

```
-- Calculate the
average customer spend
for each product
category.
```

```
SELECT
product_category,
AVG(product_price) AS
average_spend FROM
orders GROUP BY
product_category;
```

```
-- Create a bar chart
showing the top 10
products by sales.
SELECT product_name,
SUM(product_quantity)
AS total_sales FROM
orders GROUP BY
product_name ORDER BY
total_sales DESC LIMIT
10;
```

This is just a small example of what SQL can do. For more complex data processing tasks, SQL can be used in conjunction with other programming languages and tools.

Overall, data processing is all about transforming raw data into a format that can be used to gain insights. SQL is a powerful language that can be used to perform all of the steps involved in data processing.

	<p>This SQL statement combines data from the 'Orders' and 'Customers' tables using an inner join based on the 'CustomerID' column.</p> <p>In essence, data processing in SQL is about efficiently and effectively working with data to meet various business needs, whether it's analyzing trends, making informed decisions, or generating reports. The specific SQL syntax may vary depending on the database management system (e.g., MySQL, PostgreSQL, SQL Server) being used, but these core principles remain consistent.</p>	
--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--