Locking Rows for Update

In SQL, you can lock rows for update by using the "SELECT...FOR UPDATE" statement. This statement allows you to acquire a lock on the selected rows, which prevents other transactions from modifying them until the lock is released.

Here's an example of how to use "SELECT...FOR UPDATE" to lock rows for update:

```
BEGIN TRANSACTION;

SELECT * FROM my_table WHERE id = 1234 FOR UPDATE;

-- Perform some updates to the selected rows

COMMIT;
```

In this example, the "SELECT...FOR UPDATE" statement is used to select the rows with an id of 1234 and acquire a lock on them. Once the lock is acquired, you can perform some updates to the selected rows within the transaction. Finally, the "COMMIT" statement is used to release the lock and commit the changes to the database.

NOTE: It's important to note that locking rows for update can cause blocking and deadlocks in a multi-user environment. Therefore, it's important to use this feature judiciously and release the lock as soon as possible to minimize the impact on other transactions.

Creating Password and Security features

Here are some example queries that can be used to implement password and security features in SQL:

1.Creating a Hashed Password:

```
INSERT INTO users (username, password)

VALUES ('john', SHA256('password123'));
```

In this example, we are inserting a new user 'john' with a hashed password 'password123'.

2.Limiting Login Attempts:

```
CREATE TABLE login_attempts (

  user_id INT NOT NULL,

  timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

  FOREIGN KEY (user_id) REFERENCES users(user_id)

);


CREATE TRIGGER limit_login_attempts

BEFORE INSERT ON login_attempts

FOR EACH ROW

BEGIN
```

```
  DECLARE attempts INT;

  SET attempts = (SELECT COUNT(*) FROM login_attempts WHERE user_id = NEW.user_id AND timestamp > DATE_SUB(NOW(), INTERVAL 1 HOUR));

  IF attempts >= 5 THEN

    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Too many login attempts';

  END IF;

END;
```

In this example, we are creating a new table 'login_attempts' to store the login attempts of each user. We are also creating a trigger 'limit_login_attempts' that restricts the number of login attempts to 5 within a 1-hour period.

Queries on Working with Index

An index is a schema object. It is used by the server to speed up the retrieval of rows by using a pointer. It can reduce disk I/O(input/output) by using a rapid path access method to locate data quickly. An index helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements. Indexes can be created or dropped with no effect on the data. In this article, we will see how to create, delete, and uses the INDEX in the database.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and is then referred to one or more specific page numbers.

When should indexes be created:

- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause or a join condition.

When should indexes be avoided:

- The table is small
- The columns are not often used as a condition in the query
- The column is updated frequently

Here are some example queries on working with index in SQL:

Creating an Index:

```
CREATE INDEX idx_orders_customer_id ON orders (customer_id);
```

In this example, we are creating an index 'idx_orders_customer_id' on the 'customer_id' column of the 'orders' table to improve the performance of queries that filter by customer_id.

Dropping an Index:

```
DROP INDEX idx_orders_customer_id ON orders;
```

In this example, we are dropping the index 'idx_orders_customer_id' from the 'orders' table.

Listing Indexes:

SHOW INDEXES FROM orders;

In this example, we are listing all indexes defined on the 'orders' table.

Using an Index in a Query:

SELECT * FROM orders WHERE customer_id = 123;

In this example, we are using the 'idx_orders_customer_id' index to retrieve all orders for customer 123.

Analyzing an Index:

ANALYZE TABLE orders;

In this example, we are analyzing the 'orders' table to update statistics used by the query optimizer to choose the best execution plan.

Disabling an Index:

ALTER INDEX idx_orders_customer_id DISABLE;

In this example, we are disabling the 'idx_orders_customer_id' index to prevent it from being used by queries. This can be useful for testing or troubleshooting purposes.

Enabling an Index:

ALTER INDEX idx_orders_customer_id ENABLE;

In this example, we are enabling the 'idx_orders_customer_id' index after it was previously disabled.

Queries on Working with Sequence

SQL | SEQUENCES
Sequence is a set of integers 1, 2, 3, … that are generated and supported by some database systems to produce unique values on demand.

- A sequence is a user defined schema bound object that generates a sequence of numeric values.
- Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.
- The sequence of numeric values is generated in an a**scending or descending order** at defined intervals and can be configured to restart when exceeds max_value.

Different features of sequences:
1. Sequence is database object that generate produce integer values in sequential order.

2. It automatically generates primary key and unique key values

3. It may be ascending or descending order.

4. It can be used for multiple tables.

5. Sequence numbers are stored and generated independently of tables

6. It saves a time by reducing application code.

7. It is used to generate unique integers.

8. It is used to create an auto number field.

9. Useful when you need to create a unique number to act as a primary key

10. Oracle provides an object called as a Sequence that can generate numeric values. The value generated can have maximum of 38 digits

11. Provide intervals between numbers.

Here are some example queries on working with sequences in SQL:

1.Creating a Sequence:

CREATE SEQUENCE order_seq START WITH 1 INCREMENT BY 1 NO CYCLE;

In this example, we are creating a sequence 'order_seq' that starts with 1 and increments by 1 for each new value. The 'NO CYCLE' option means that the sequence will not wrap around when it reaches its maximum value.

2.Getting the Next Sequence Value:

SELECT NEXTVAL('order_seq');

In this example, we are retrieving the next value from the 'order_seq' sequence.

3.Using a Sequence in a Table:

CREATE TABLE orders (order_id INTEGER PRIMARY KEY DEFAULT NEXTVAL('order_seq'), customer_id INTEGER, order_date DATE);

In this example, we are creating a table 'orders' with an 'order_id' column that uses the 'order_seq' sequence as its default value. This ensures that each new order will be assigned a unique identifier.

4.Altering a Sequence:

ALTER SEQUENCE order_seq INCREMENT BY 2;

In this example, we are changing the increment of the 'order_seq' sequence to 2 instead of 1.

5.Resetting a Sequence:

ALTER SEQUENCE order_seq RESTART WITH 1;

In this example, we are resetting the 'order_seq' sequence to start with 1 again.

6.Listing Sequences:

SELECT sequence_name, last_value, is_cycled FROM information_schema.sequences;

In this example, we are listing all sequences in the database and their current values and cycling status.

Queries on Working with Synonym

SQL | SYNONYM
A **SYNONYM** provides another name for database object, referred to as original object, that may exist on a local or another server. A synonym belongs to schema, name of synonym should be unique. A synonym cannot be original object for an additional synonym and synonym cannot refer to user-defined function. Synonyms are database dependent and cannot be accessed by other databases.

Here are some example queries on working with synonyms in SQL:

Creating a Synonym:

```
CREATE SYNONYM orders FOR dbo.orders;
```

In this example, we are creating a synonym 'orders' for the 'dbo.orders' table. This allows users to refer to the table using a shorter and more convenient name.

Using a Synonym in a Query:

```
SELECT * FROM orders WHERE customer_id = 123;
```

In this example, we are using the 'orders' synonym to retrieve all orders for customer 123.

Altering a Synonym:

```
ALTER SYNONYM orders TO dbo.all_orders;
```

In this example, we are changing the name of the 'orders' synonym to 'dbo.all_orders'.

Dropping a Synonym:

```
DROP SYNONYM orders;
```

In this example, we are dropping the 'orders' synonym from the database.

Listing Synonyms:

```
SELECT name, base_object_name FROM sys.synonyms;
```

In this example, we are listing all synonyms in the database and their associated base objects.

Using a Synonym for a Remote Object:

```
CREATE SYNONYM sales FOR salesdb.dbo.sales@linked_server;
```

In this example, we are creating a synonym 'sales' for a remote table 'salesdb.dbo.sales' accessed through a linked server. This allows us to refer to the remote table using a local name.

Queries on Working with Controlling Access

Controlling User Access

There are two commands in SQL that allow database access control involving the assignment of privileges and the revocation of privileges. The following are the two commands used to distribute both system and object privileges in a relational database:

- GRANT
- REVOKE

The GRANT Command

The GRANT command is used to grant both system-level and object-level privileges to an existing database user account.

The REVOKE Command

The REVOKE command removes privileges that have been granted to database users.
The REVOKE command has two options: RESTRICT and CASCADE. When the RESTRICT option is used, REVOKE succeeds only if the privileges specified explicitly in the REVOKE statement leave no other users with abandoned privileges. The CASCADE option revokes any privileges that would otherwise be left with other users. In other words, if the owner of an object granted USER1 privileges with GRANT OPTION, USER1 granted USER2 privileges with GRANT OPTION, and then the owner revokes USER1's privileges, CASCADE also removes the privileges from USER2.

Here are some example queries on controlling access in SQL:

Creating a User:

```
CREATE USER user1 WITH PASSWORD 'password1';
```

In this example, we are creating a user 'user1' with a password 'password1'. This allows the user to log in to the database and perform authorized actions.

Granting Permissions:

```
GRANT SELECT, INSERT, UPDATE ON dbo.orders TO user1;
```

In this example, we are granting the 'user1' user permission to select, insert, and update records in the 'dbo.orders' table. This allows the user to perform these actions on the specified table.

Revoking Permissions:

```
REVOKE SELECT, INSERT, UPDATE ON dbo.orders FROM user1;
```

In this example, we are revoking the 'user1' user's permission to select, insert, and update records in the 'dbo.orders' table. This denies the user the ability to perform these actions on the specified table.

Listing Permissions:

```
SELECT * FROM sys.database_permissions WHERE grantee_principal_id = USER_ID('user1');
```

In this example, we are listing all permissions granted to the 'user1' user in the current database. This allows us to view the user's authorized actions.

Creating a Role:

```
CREATE ROLE sales_team;
```

In this example, we are creating a role 'sales_team'. This allows us to group users together for easier management of permissions.

Assigning a Role:

```
ALTER ROLE sales_team ADD MEMBER user1;
```

In this example, we are assigning the 'user1' user to the 'sales_team' role. This allows the user to inherit the permissions granted to the role.