|  |
| --- |
| [Type the company name] |
| WEEK 6 - THEORY |
| [Type the document subtitle] |
|  |
| **Aakash s** |
| **[Pick the date]** |

|  |
| --- |
| [Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.] |

Data encryption

sqflite - advantages and limitations(sqlite commands)

ACID properties(isolation- examples)

Versioning in database

Data migration

**Data encryption**

**Securing data in a local database in a Flutter application involves implementing measures to protect sensitive information from unauthorized access.(unauthorized access refers to the situation where someone gains entry or uses a system or data without proper permission or authorization.)**

**Use Encryption:**

* **Encrypt sensitive data before storing it in the local database. Flutter provides packages like pointycastle or encrypt that you can use for encryption.**
* **For SQLite databases, you can use the sqflite package in combination with encryption plugins like sqflite\_encrypted to encrypt the entire database.**

**Use Secure Storage:**

* **For storing sensitive information such as API keys, passwords, and tokens, consider using Flutter's flutter\_secure\_storage package. This package stores data in a secure keystore/keychain.**

import 'package:flutter\_secure\_storage/flutter\_secure\_storage.dart';

void storeSensitiveData() {

final storage = new FlutterSecureStorage();

storage.write(key: 'api\_key', value: 'your\_api\_key');

}

Future<String?> readSensitiveData() async {

final storage = new FlutterSecureStorage();

return await storage.read(key: 'api\_key');

}

**Sqlite**

**Advantages**

* SQLite does not require a separate server process or system to operate (serverless).
* SQLite comes with zero-configuration, which means no setup or administration needed.
* A complete SQLite database is stored in a single cross-platform disk file.
* SQLite is very small and light weight, less than 400KiB fully configured or less than 250KiB with optional features omitted.
* SQLite is self-contained, which means no external dependencies.
* SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.
* SQLite supports most of the query language features found in SQL92 (SQL2) standard.
* SQLite is written in ANSI-C and provides simple and easy-to-use API.
* SQLite is available on UNIX (Linux, Mac OS-X, Android, iOS) and Windows (Win32, WinCE, WinRT)

**Limitations**

* Only LEFT OUTER JOIN is implemented(not right inner join or full outer join).
* The RENAME TABLE and ADD COLUMN variants of the ALTER TABLE command are supported. The DROP COLUMN, ALTER COLUMN, ADD CONSTRAINT are not supported
* FOR EACH ROW triggers are supported but not FOR EACH STATEMENT triggers.
* VIEWs in SQLite are read-only. You may not execute a DELETE, INSERT, or UPDATE statement on a view.
* The only access permissions that can be applied are the normal file access permissions of the underlying operating system.

**Sqlite commands**

The standard SQLite commands to interact with relational databases are similar to SQL. They are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into groups based on their operational nature .

DDL - Data Definition Language

|  |  |
| --- | --- |
| 1 | **CREATE**  Creates a new table, a view of a table, or other object in database. |
| 2 | **ALTER**  Modifies an existing database object, such as a table. |
| 3 | **DROP**  Deletes an entire table, a view of a table or other object in the database. |

## DML - Data Manipulation Language

|  |  |
| --- | --- |
| 1 | **INSERT**  Creates a record |
| 2 | **UPDATE**  Modifies records |
| 3 | **DELETE**  Deletes records |

## DQL - Data Query Language

**SELECT**

Retrieves certain records from one or more tables

**ACID properties**

**Atomicity**

**Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the changes made within the transaction are committed to the database, or none of them are. If any part of the transaction fails, the entire transaction is rolled back to its original state, maintaining the consistency of the database.**

**Consistency:**

Consistency ensures that a transaction brings the database from one valid state to another. If a transaction violates the integrity constraints of the database (e.g., primary key constraints, unique constraints), it is rolled back, and the database remains unchanged. Consistency guarantees that the database remains in a valid state before and after the execution of a transaction.

**Isolation:**

Isolation ensures that the concurrent execution of multiple transactions does not interfere with the correctness of each transaction. Each transaction appears to execute in isolation from other transactions, even if they are executing concurrently. Isolation prevents the effects of one transaction from being visible to other transactions until it is committed. Common isolation levels include Read Uncommitted, Read Committed, Repeatable Read, and Serializable.

**Isolation Example: Online Ticket Booking**

**Imagine you and your friend are both booking tickets online for a popular concert. The system needs to manage multiple users trying to reserve seats simultaneously.**

**Starting Point:**

**Initially, there are 100 available seats for the concert.**

**Transaction 1 (You):**

**You start the booking process to reserve two tickets. The system checks if there are enough available seats (let's say 98 seats remaining) and starts the reservation.**

**Transaction 2 (Your Friend):**

**At the same time, your friend also wants to book tickets. They start the process and check for available seats. The system sees 98 seats (due to your ongoing reservation) and starts their reservation with the assumption of 96 seats remaining.**

**Isolation at Work:**

**Isolation ensures that while your friend is going through the booking process, they don't see your reservation in progress. They're operating in their own isolated space, imagining there are still 98 seats available.**

**Completion of Transactions:**

**Assuming both transactions are successful, your reservation is confirmed, and your friend's reservation is also confirmed. The actual number of remaining seats is now 96.**

**In this scenario:**

**Consistency: The system ensures that the number of available seats never goes below zero, maintaining a consistent and valid state.**

**Isolation: Your friend's booking process is isolated from yours. They don't see your reservation affecting the available seat count until the transactions are completed.**

**Durability:**

Durability guarantees that once a transaction is committed, its effects are permanent and will survive any subsequent failures, such as power outages or crashes. The changes made by a committed transaction are stored in non-volatile storage (e.g., hard disk) and are not lost even if the system experiences a failure. Durability ensures the long-term reliability of the data.

**Versioning in database**

Database versioning refers to the practice of tracking and managing changes to your database schema (structure) and data over time

1. Controlled Change Management:

* You define distinct versions and upgrade paths, ensuring changes are applied in a structured manner.
* Upgrades trigger migration scripts based on version changes, transforming data from old to new formats.
* This prevents unexpected compatibility issues with older app versions when the schema changes.

2. Data Integrity and Reliability:

* Versioning reduces the risk of data loss or corruption during upgrades.
* You can implement roll-back mechanisms to revert to previous versions if new updates introduce problems.
* This helps safeguard your valuable data and maintain a reliable database for your app.

3. Improved Collaboration and Visibility:

* Versioning tracks change history, making it easier to understand the database's development and evolution.
* Different teams can collaborate effectively on database modifications with clear versioning guidelines.
* This increases transparency and fosters better communication within your development process.

4. Flexibility and Future-proofing:

* You can adapt your database to changing needs and user demands without impacting older versions.
* Versioning allows for incremental upgrades, introducing new features and functionalities gradually.
* This makes your app more flexible and adaptable to future requirements.

Different Versioning Approaches:

* Version numbers: Simple and common, assigning increasing numbers to different versions.
* Branching and merging: Similar to software development, allowing independent work on different schemas.
* Schema history: Tracks modifications directly on the schema with detailed change logs and roll-back points.

**Data migration**

**Data migration in a database refers to the process of safely transferring existing data from one format or structure to another. It's often necessary when you change the schema (structure) of your database, like adding new tables, columns, or modifying existing ones.**

Reasons for Migration:

* Schema changes: As your app or system evolves, the database schema may need to adapt to accommodate new features or functionalities. This triggers the need for data migration.
* Technology upgrades: Switching to a different database system or engine might require data migration to adapt to the new format.
* Consolidation: Merging databases from different sources might involve data migration to unify the format and structure.