Section 19: Databases

15 July 2021 19:23

Section 19: Databases

Database : the container for all the data that you store.
Database Dictionary : provides a comprehensive list of the structure and types of data in the database.

Table : a collection of related data help in the database.
Field : the basic unit of data in a table.
Column : is another name for field. This can be confusing, but relational databases existed before spreadsheets.
Row : a single set of data containing all the columns in the table. Rows are also called records.
File Flat database stores all data in a single table. This results in a lot of duplication.

Splitting the data is knows as **Normalization**. **Database normalization** is basically the process of removing redundant duplicated and irrelevant data from the tables and the more that this is done the higher the level of normalization.
If we look into a normalization we'll find that we can go up to level 6. 6 normal form but in most practical application it's rare to go beyond the 3rd level.

A **View** is a selection of rows and columns, possible from more than one table.

SQLite Demo Commands:-

.headers on // to see column names while printing the data

CREATE TABLE contacts (name text, phone integer, email text);
INSERT into contacts (name, phone, email) values('RJ', 34234234, 'rj@email.com');
SELECT * FROM contacts;
SELECT name, phone, email FROM contacts;
INSERT into contacts VALUES("R", 23434234234, "R@email.com");
INSERT into contacts (name, phone) values('K', 34234);

A **sqlite command** can end without semicolon and starts with . (dot)
A **sqlite statement** must end with semicolon

.backup testbackup // will backup current database
UPDATE contacts **SET** email="st@email.com"; // will update all email in database, need to very careful while using UPDATE command.
.restore testbackup // restoring backup database
UPDATE contacts SET email="st@email.com" WHERE name = "K";
SELECT * from contacts WHERE name="RJ";
SELECT name, email from contacts WHERE name="RJ";
DELETE FROM contacts WHERE phone="34234";

.tables : will list all the tables in database
.schema : print out the structure of your table
.dump : gives us the sequence statement for creating the table but all the inserts necessary to populate it with the data that's in it. So it wraps the whole thing and what's called a transaction. WE will get begin transactions and commits. Can be used to copy and paste the output from dumped into our code.
.exit or .quit : exit the sqlite shell

A **Key** in a table is in index which provides a way to really speed up searches and joins on a column.
Now when columns are indexed they can be searched much faster than if they are not.
Basically index columns are sorted so that they can be searched through much faster.

Relation Database : ordering of the rows is undefined. They are very similar to java maps or to set.
In fact relational database theory is heavily based on set theory.
By defining a key, we are making the data ordered on that column or group or columns and searches etc. work far more efficiently as a result of doing that.
There can be lots of keys on a table but there can only be one primary key. Primary key must be unique.

SQLite Autoincrement :-
https://www.sqlite.org/autoinc.html

.schema
CREATE TABLE songs (_id INTEGER PRIMARY KEY, track INTEGER, title TEXT NOT NULL, album INTEGER);
CREATE TABLE albums (_id INTEGER PRIMARY KEY, name TEXT NOT NULL, artist INTEGER);
CREATE TABLE artists (_id INTEGER PRIMARY KEY, name TEXT NOT NULL);

SELECT * FROM albums ORDER BY name; // lower case will show sorted after upper case
SELECT * FROM albums ORDER BY name COLLATE NOCASE; // can ignore case using NOCASE
SELECT * FROM albums ORDER BY name COLLATE NOCASE DESC; // sort in descending order (ASC for ascending)
SELECT * FROM albums ORDER BY artist, name COLLATE NOCASE; // sort first by album ID and then by album name
SELECT * FROM songs ORDER BY album, track;

Joining the Tables :-
SELECT songs.track, songs.title, albums.name FROM songs JOIN albums ON songs.album = albums._id;
SELECT songs.track, songs.title, albums.name FROM songs INNER JOIN albums ON songs.album = albums._id ORDER BY albums.name, songs.track;
SELECT albums.name, songs.track, songs.title FROM songs INNER JOIN albums ON songs.album = albums._id ORDER BY albums.name, songs.track;

Joining 3 Tables :
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
  ...> INNER JOIN albums ON songs.album = albums._id
  ...> INNER JOIN artists ON albums.artist = artists._id
  ...> ORDER BY artists.name, albums.name, songs.track;

ORDER also matters :-
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
WHERE albums.name = "Doolittle"
ORDER BY artists.name, albums.name, songs.track;

LIKE and Wild Card Searched and Character (%) :-
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
WHERE songs.title **LIKE** "%doctor%"
ORDER BY artists.name, albums.name, songs.track;

Creating a VIEW :-

```
CREATE VIEW IF NOT EXISTS artist_list AS
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
ORDER BY artists.name, albums.name, songs.track;
```

Cross check using .schema command. Can use those view as any other table.

Deleting a VIEW using DROP command :-

```
DROP VIEW astist_list;    // we can also delete table using DROP, but deleting a view didn't acceft as we have creating it for our convenience.
```

We can also rename column name while creating a View (So the name don't clash):-

```
CREATE VIEW artist_list AS
SELECT artists.name AS artist, albums.name AS album, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
ORDER BY artists.name, albums.name, songs.track;
```

```
DELETE FROM songs WHERE track < 50;
SELECT * FROM songs WHERE track <> 71    // <> = !=
SELECT COUNT(*) FROM songs;
```

```
UPDATE artists SET name = "One Kitten" WHERE artists.name = "Mehitabel";
SELECT * FROM artists WHERE artists.name = "One Kitten";
SELECT count(title) From artist_list WHERE artist = "Aerosmith";
SELECT DISTINCT title From artist_list WHERE artist = "Aerosmith" ORDER BY title;    // for no duplicates
SELECT COUNT(DISTINCT title) From artist_list WHERE artist = "Aerosmith";    // counting distinct title
SELECT COUNT(DISTINCT album) From artist_list WHERE artist = "Aerosmith";
```

Download SQLite JDBC :- (Download jar file)
https://github.com/xerial/sqlite-jdbc/releases

2 Ways to create database from java :-

1) Using Driver Manager
2) Using Data source Object (For more advance)

Creating a Table using Driver Manager :-

```
Connection conn = DriverManager.getConnection("jdbc:sqlite:C:\\Users\\rajatkumar\\Documents\\IdeaProjects\\Java-Programming-Course\\TestDB\\testjava.db");

Statement statement = conn.createStatement();
statement.execute("CREATE TABLE contacts (name TEXT, phone INTEGER, email TEXT)");

statement.execute("CREATE TABLE IF NOT EXISTS contacts " +
                  " (name TEXT, phone INTEGER, email TEXT)");
```

Inserting the data in table :-

```
statement.execute("INSERT INTO contacts (name, phone, email) " +
                  "VALUES('Joe', 45632, 'joe@anywhere.com')");
```

Printing the data :-

```
statement.execute("SELECT * FROM contacts");
ResultSet results = statement.getResultSet();

//  ResultSet results = statement..execute("SELECT * FROM contacts");
while(results.next())
{
        System.out.println(results.getString("name") + " " +
        results.getInt("phone") + " " +
        results.getString("email"));
}
```

```
public static final String CONNECTION_STRING = "jdbc:sqlite:C:\\Users\\rajatkumar\\Documents
\\IdeaProjects\\Java-Programming-Course\\Music\\" + DB_NAME;
```

SQLite will create an empty database if it didn't find a data base file in mentioned path.

Column indices are one based (Indexing)

JDBC don't have .schema command support. But we can use **ResultSetMetaData** to get query about number of columns and names of it in a table. Column number starts from 1 not 0.
Interface ResultSetMetaData :-
https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html

Use songArtists.isEmpty()) instead of songArtists == null. Better Way.

Should use " = ?" **Prepared Statements** instead of regular to avoid **SQL injection Attacks**. It will substitute the value and will treat it as a single literal and won't interpreted as sql.

PreparedStatments :- It's good practice to use PreparedStatments because of the potential performance benefit, and because they protect the database against SQL injection attacks.

We do the following to use a PreparedStatement :

1) Declare a constant for the SQL statement that contains the placeholders.
2) Create a PreparedStatetement instance using
    a. Connection.prepareStatement(sqlStmtString)
3) When we're ready to perform the query (or the insert, update, delete), we call the appropriate setter methods to set the placeholders to the values we want to use in the statement
4) We run the statement using PreparedStatements.execute() or PreparedStatement.executeQuery()
5) We process the results the same way we do when using a regular old statement.

Transactions :- A transaction is a sequence of SQL statements that are treated as a single logical unit. If any of the statement fail, the results of any previous statements in the transaction can be rolled back, or just not saved. It's as if they never happened.

Database transactions must be **ACID** compliant.

1) **Atomicity** - If a series of SQL statements change the database, then either all the changes are committed, or none of them are.
2) **Consistency** - Before a transaction begins, the database is in a valid state. When it completes, the database is still in a valid state.
3) **Isolation** - Until the changes committed by a transaction are completed, they won't be visible to other connections. Transactions can't depend on each other.
4) Durability -Once the changes performed by a transaction are committed to the database, they're permanent. If an application then crashes or the database server goes down (in the case of a client/server database like MYSQL), the changes made by the transition are still there

when the application runs again, or the database comes back up.

Essentially Transactions ensure the integrity of the data within a Database.

SQLite uses transactions by default, and auto-commits by default.
But we change it using :-

```
conn.setAutoCommit(false);
If(condition)
{
        conn.commit();
}
catch(Exception e) {….}
finally {
         try {
            System.out.println("Resetting default commit behavior");
            conn.setAutoCommit(true);
                }
                catch(Exception e) { …..}
}
```

Can also create a GUI Program using JavaFX and connect SQLite database using JDBC.