SQL 4: Joining Tables, Updating and Deleting Data, ACID, and Transactions

Databases and Interfaces

Matthew Pike & Yuan Yao

University of Nottingham Ningbo China (UNNC)

Overview

This Lecture

- In this lecture we will cover:
  - Using `JOIN` to combine data from multiple tables.
  - Updating data in existing tables using `UPDATE`.
  - Using `DELETE` to remove data from tables.
  - Combining multiple SQL statements into a single transaction.
  - Motivation and use cases for transactions.
  - The ACID properties of transactions.

The Database Schema for this Lecture

```sql
CREATE TABLE Student(
    sID INTEGER PRIMARY KEY,
    firstName VARCHAR(20) NOT NULL,
    lastName VARCHAR(20) NOT NULL
);
```

```sql
CREATE TABLE Module(
    mCode CHAR(8) PRIMARY KEY,
    title VARCHAR(30) NOT NULL,
    credits INTEGER NOT NULL
);
```

```sql
CREATE TABLE Grade(
    sID INTEGER NOT NULL,
    mCode CHAR(8) NOT NULL,
    grade INTEGER NOT NULL,
    PRIMARY KEY (sID, mCode),
    FOREIGN KEY (sID)
        REFERENCES Student(sID),
    FOREIGN KEY (mCode)
        REFERENCES Module(mCode)
);
```

The Database Content for this Lecture

sID	firstName	lastName
1	John	Smith
2	Jane	Doe
3	Mary	Jones
4	Joe	Bloggs

Table 1: `Student` Table

mCode	title	credits
COMP1036	Fundamentals	20
COMP1048	Databases	10
COMP1038	Programming	20

Table 2: `Module` Table

sID	mCode	grade
1	COMP1036	35
1	COMP1048	50
2	COMP1048	65
2	COMP1038	70
3	COMP1036	35
3	COMP1038	65
6	COMP1038	55
6	COMP1099	68

Table 3: `Grade` Table

Joining Tables in SQL

The Need for `JOIN`

- Typically, we will split data into multiple tables to reduce redundancy and improve data integrity - something we will cover in more detail in the next lecture.
- However, this means that we need to be able to combine data from multiple tables when we query the database - this is where `JOIN` comes in.
- `JOIN` is an operation that allows us to combine rows from two or more tables, based on a related column between them. This is ultimately where `FOREIGN KEY` constraints come into play.
- This is a very powerful feature of SQL, and is one of the main reasons why SQL is so widely used - it allows us to store data in a way that is efficient and easy to maintain, while still allowing us to combine data from multiple tables when we need to.

Types of JOIN

- There are several different types of `JOIN`:
  - `CROSS JOIN`
  - `INNER JOIN`
  - `LEFT JOIN` and `RIGHT JOIN`
  - `NATURAL JOIN`
  - `FULL OUTER JOIN`
- It is important to understand the differences between these different types of `JOIN`, as they can produce very different results.

CROSS JOIN

- As with the `SELECT` from multiple tables, the `CROSS JOIN` returns the Cartesian product of the two tables.
  - This means that every row from the first table is combined with every row from the second table.
  - This also means that the resulting table will contain rows of data that are not related (nonsensical data).
- The syntax for a CROSS JOIN is:
  - `SELECT * FROM table1 CROSS JOIN table2;`
  - Which is equivalent to:
    - `SELECT * FROM table1, table2;`
- `CROSS JOIN` is rarely used in practice, as it can result in a very large number of rows.
  - We can constrain the number of rows returned by using a `WHERE` clause.

Example: CROSS JOIN

```sql
SELECT * FROM Student CROSS JOIN Module LIMIT 8;
```

sID	firstName	lastName	mCode	title	credits
1	John	Smith	COMP1036	Fundamentals	20
1	John	Smith	COMP1048	Databases	10
1	John	Smith	COMP1038	Programming	20
2	Jane	Doe	COMP1036	Fundamentals	20
2	Jane	Doe	COMP1048	Databases	10
2	Jane	Doe	COMP1038	Programming	20
3	Mary	Jones	COMP1036	Fundamentals	20
3	Mary	Jones	COMP1048	Databases	10

Table 4: The first 8 results of the `CROSS JOIN` of `Student` and `Module`

SELECT from Multiple Tables

> ⚠ **Do not use this approach!**
>
> In general, you should not use `SELECT` to combine data from multiple tables. Instead, you should use `JOIN` as this more readable and easier to understand.

- `SELECT` can be used with multiple tables, with table names separated by commas in the `FROM` clause.
  - `SELECT * FROM Student, Module;`
  - This is equivalent to a `CROSS JOIN` of the two tables.
- We can limit the columns returned by `SELECT` by specifying the table name before the column name.
  - `SELECT Student.sID, Module.mCode FROM Student, Module;`

Example: Emulating CROSS JOIN with SELECT

```sql
SELECT *
FROM Student, Module
LIMIT 8;
```

sID	firstName	lastName	mCode	title	credits
1	John	Smith	COMP1036	Fundamentals	20
1	John	Smith	COMP1048	Databases	10
1	John	Smith	COMP1038	Programming	20
2	Jane	Doe	COMP1036	Fundamentals	20
2	Jane	Doe	COMP1048	Databases	10
2	Jane	Doe	COMP1038	Programming	20
3	Mary	Jones	COMP1036	Fundamentals	20
3	Mary	Jones	COMP1048	Databases	10

Table 5: The first 8 results of the `SELECT` from `Student` and `Module`

Aside: Ambiguous Column Names

- When we use `SELECT` with multiple tables or `JOIN`, we may end up with ambiguous column names.
- For example, if we `SELECT` from `Student` and `Grade`, we will have two columns called `sID`.
- This may lead to errors, or unexpected results. For example the following query will fail:
    - `SELECT sID FROM Student, Grade;`
- Returns - `Parse error: ambiguous column name: sID`. To fix this, we need to specify which table the column comes from:
    - `SELECT Student.sID FROM Student, Grade;`

Example: SELECT from Multiple Tables

```sql
SELECT
Student.sID,
Module.mCode,
grade --Not ambiguous
FROM
    Student, Grade, Module
WHERE
    Student.sID = Grade.sID
    AND
    Module.mCode = Grade.mCode;
```

sID	mCode	grade
1	COMP1036	35
1	COMP1048	50
2	COMP1048	65
2	COMP1038	70
3	COMP1036	35
3	COMP1038	65

Table 6: The SELECT from Multiple Tables

INNER JOIN

`INNER JOIN`

- `INNER JOIN` is perhaps the most commonly used type of `JOIN`. It returns only rows where the join condition is met.
- The join condition is specified in the `ON` clause.
    - `SELECT * FROM table1 INNER JOIN table2 ON table1.column1 = table2.column2;`
- For example:
    - `SELECT * FROM Student INNER JOIN Grade ON Student.sID = Grade.sID;`
    - This will return only rows where the `sID` column in `Student` matches the `sID` column in `Grade`.

Example: INNER JOIN

```
SELECT
    Student.lastName,
    Grade.grade
FROM
    Student
INNER JOIN Grade ON
    Student.sID = Grade.sID;
```

lastName	grade
Smith	35
Smith	50
Doe	65
Doe	70
Jones	35
Jones	65

Table 7: The `INNER JOIN` of `Student` and `Grade`

INNER JOIN with Multiple Tables

We can use `INNER JOIN` with multiple tables - we just need to specify the join condition for each table.

```
SELECT
    Student.lastName,
    Grade.grade,
    Module.title
FROM
    Student
INNER JOIN Grade ON
    Student.sID = Grade.sID
INNER JOIN Module ON
    Grade.mCode = Module.mCode;
```

lastName	grade	title
Smith	35	Fundamentals
Smith	50	Databases
Doe	65	Databases
Doe	70	Programming
Jones	35	Fundamentals
Jones	65	Programming

Table 8: The `INNER JOIN` of `Student`, `Grade`, and `Module`

LEFT and RIGHT JOINs

`LEFT JOIN` and `RIGHT JOIN`

> ⚠ **Avoid using `RIGHT JOIN`**
>
> In general, you should avoid using `RIGHT JOIN` as it is less readable than `LEFT JOIN`. Instead, you should use `LEFT JOIN` and swap the order of the tables. Also, `RIGHT JOIN` is not supported in older versions of SQLite (including the version used in the labs).

- Often we will want to return all rows from one table, even if there is no match in the other table.
  - For example, we may want to return all students, even if they have not taken any modules.
- `LEFT` and `RIGHT` joins allow us to do this, using the following syntax:
  - `SELECT * FROM leftTable LEFT JOIN rightTable ON condition;`
  - `SELECT * FROM leftTable RIGHT JOIN rightTable ON condition;`
- In practice `LEFT JOIN` is more commonly used than `RIGHT JOIN`. You are advised to only use `LEFT JOIN`.

Example: LEFT JOIN

```sql
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Grade.grade AS "Grade"
FROM
    Student LEFT JOIN Grade
    ON
    Student.sID = Grade.sID;
```

sID	Last	Grade
1	Smith	35
1	Smith	50
2	Doe	70
2	Doe	65
3	Jones	35
3	Jones	65
4	Bloggs	NA

Table 9: The `LEFT JOIN` of `Student` and `Grade`. We see that student 4 is missing from the `Grade` table, but is still returned.

Example: LEFT JOIN on Multiple Tables

```sql
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Module.mCode AS "Module",
    Grade.grade AS "Grade"
FROM
    Student LEFT JOIN Grade
    ON
    Student.sID = Grade.sID
    LEFT JOIN Module
    ON
    Grade.mCode = Module.mCode;
```

sID	Last	Module	Grade
1	Smith	COMP1036	35
1	Smith	COMP1048	50
2	Doe	COMP1038	70
2	Doe	COMP1048	65
3	Jones	COMP1036	35
3	Jones	COMP1038	65
4	Bloggs	NA	NA

Table 10: The `LEFT JOIN` of `Student` and `Grade`. Note the final row.

Going from RIGHT JOIN to LEFT JOIN

```sql
-- RIGHT Joins can be trivially
-- converted to LEFT joins by
-- swapping the order of the tables.
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Grade.grade AS "Grade"
FROM
    Grade RIGHT JOIN Student
    ON
    Student.sID = Grade.sID;
```

```sql
-- Equivalent to:
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Grade.grade AS "Grade"
FROM
    Student LEFT JOIN Grade
    ON
    Student.sID = Grade.sID;
```

NATURAL JOIN

Using `NATURAL JOIN`

- `NATURAL JOIN` is a special type of `JOIN` that does not require a join condition.
- Instead, it automatically joins the two tables on all columns that have the same name in both tables.
    - For example, if both tables have a column called `sID`, the `NATURAL JOIN` will automatically join the tables on the `sID` column (equivalent to `ON table1.sID = table2.sID`).

Example: NATURAL JOIN

```
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Grade.grade AS "Grade"
FROM
    Student
    NATURAL JOIN -- sID
    Grade;
```

sID	Last	Grade
1	Smith	35
1	Smith	50
2	Doe	65
2	Doe	70
3	Jones	35
3	Jones	65

Table 11: The `NATURAL JOIN` of `Student` and `Grade`

Example: NATURAL JOIN on Multiple Tables

```sql
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Module.mCode AS "Module",
    Grade.grade AS "Grade"
FROM
    Student
    NATURAL JOIN -- sID
    Grade
    NATURAL JOIN -- mCode
    Module;
```

sID	Last	Module	Grade
1	Smith	COMP1036	35
1	Smith	COMP1048	50
2	Doe	COMP1048	65
2	Doe	COMP1038	70
3	Jones	COMP1036	35
3	Jones	COMP1038	65

Table 12: The `NATURAL JOIN` of `Student`, `Grade`, and `Module`

FULL OUTER JOIN

Getting All Rows with `FULL OUTER JOIN`

> **!** Support for `FULL OUTER JOIN`
>
> Support for `FULL OUTER JOIN` is only available in SQLite version 3.39.0 and above.

- The `FULL OUTER JOIN` returns all rows from both tables, where the join condition is met.
- Any rows from the left table that do not have a match in the right table are returned with `NULL` values.
- Any rows from the right table that do not have a match in the left table are returned with `NULL` values.
- The syntax for a `FULL OUTER JOIN` is:
  - `SELECT * FROM table1 FULL OUTER JOIN table2 ON condition;`

Example: FULL OUTER JOIN

```
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Module.mCode AS "Module",
    Grade.grade AS "Grade"
FROM
    Student FULL OUTER JOIN Grade
    ON
    Student.sID = Grade.sID
    FULL OUTER JOIN Module
    ON
    Grade.mCode = Module.mCode;
```

sID	Last	Module	Grade
1	Smith	COMP1036	35
1	Smith	COMP1048	50
2	Doe	COMP1038	70
2	Doe	COMP1048	65
3	Jones	COMP1036	35
3	Jones	COMP1038	65
4	Bloggs	NA	NA
NA	NA	COMP1038	55
NA	NA	NA	68

Table 13: The `FULL OUTER JOIN` of `Student`, `Grade`, and `Module`

Updating Existing Data with SQL

UPDATE Statement

- Data stored in a database is rarely static - it is often updated, deleted, and new data is added.
- The `UPDATE` statement allows us to modify existing records in a table, without having to delete and re-insert the record.
- The `UPDATE` statement can be used to update a single record, or multiple records. The syntax is:
  - `UPDATE table_name SET column1 = value1, column2 = value2, ... [WHERE condition];`
  - The `WHERE` clause is optional - if it is omitted, all records in the table will be updated.
  - Within the `SET` clause, you can specify multiple columns and values.
- The `UPDATE` statement can reference column values from other columns in the same row.
  - For example, `UPDATE table SET column1 = column1 + 1;` will increment the value of `column1` by 1.

Example: UPDATE Statement

```sql
UPDATE Student
SET
    firstName = "Johnathan",
    lastName = "Creek"
WHERE sID = 1;
```

```sql
SELECT * FROM Student;
```

sID	firstName	lastName
1	Johnathan	Creek
2	Jane	Doe
3	Mary	Jones
4	Joe	Bloggs

Table 14: The Student table after UPDATE

Example: UPDATE Statement on Multiple Rows

```sql
UPDATE
    Grade
SET
    grade = grade + 10;
```

```sql
SELECT * FROM Grade LIMIT 5;
```

sID	mCode	grade
1	COMP1036	45
1	COMP1048	60
2	COMP1048	75
2	COMP1038	80
3	COMP1036	45

Table 15: The Grade table after **UPDATE**

Example: UPDATE Statement referencing other Columns

```sql
UPDATE
    Grade
SET
    grade = sID + grade;
```

```sql
SELECT * FROM Grade LIMIT 5;
```

sID	mCode	grade
1	COMP1036	46
1	COMP1048	61
2	COMP1048	77
2	COMP1038	82
3	COMP1036	48

Table 16: The Grade table after **UPDATE**

Deleting Data with SQL

Removing Data with `DELETE`

- As data becomes out of date, or is no longer needed, it is often removed from the database.
- The `DELETE` statement allows us to remove records from a table.
- Similar to the `UPDATE` statement, the `DELETE` statement can be used to delete a single record, or multiple records.
- The syntax for the `DELETE` statement is:
    - `DELETE FROM table_name [WHERE condition];`
    - The `WHERE` clause is optional - if it is omitted, all records in the table will be deleted.
- The `DELETE` statement returns the number of rows that were deleted.

Example: DELETE Statement

```
DELETE FROM Grade WHERE sID = 3;
```

```
SELECT * FROM Grade;
```

sID	mCode	grade
1	COMP1036	46
1	COMP1048	61
2	COMP1048	77
2	COMP1038	82
6	COMP1038	71
6	COMP1099	84

Table 17: The Grade table after DELETE

Referential Integrity

Referential Integrity and SQLite

Remember, by default, SQLite does not enforce referential integrity. To enable it, we need to use: `PRAGMA foreign_keys = ON;`.

- When we modify (update or delete) data in a table which is referenced by a `FOREIGN KEY` constraint, we need to consider the impact on the other tables.
  - This is known as referential integrity.
- For example, if we delete a student from the `Student` table, we need to consider what happens to the `Grade` table.
  - Do we delete the student's grades?
  - Do we set the student's grades to `NULL`?
  - Do we prevent the student from being deleted?

Example (1/4): DELETEing a Student

- If not specified (as is the case in our `CREATE` statement for `Grade`) SQLite will set the `ON DELETE` action to `NO ACTION`.
  - This means that the `DELETE` will fail if there are any rows in the `Grade` table that reference the student being deleted. This is the safest option, as it prevents accidental deletion of data.

```
PRAGMA foreign_keys = ON;
```

```
-- This will fail:
DELETE FROM
    Student
WHERE sID = 1;
```

```
SELECT * FROM Student;
```

sID	firstName	lastName
1	Johnathan	Creek
2	Jane	Doe
3	Mary	Jones
4	Joe	Bloggs

Table 18: The Student table after `DELETE`

Example (2/4): Add CASCADE to ON DELETE Action in Grade Table

```
PRAGMA foreign_keys = OFF;
```

```
DROP TABLE Grade;
```

```
CREATE TABLE Grade(
    sID INTEGER NOT NULL,
    mCode CHAR(8) NOT NULL,
    grade INTEGER NOT NULL,
    PRIMARY KEY (sID, mCode),
    FOREIGN KEY (sID) REFERENCES Student(sID)
        ON DELETE CASCADE,
    FOREIGN KEY (mCode) REFERENCES Module(mCode)
        ON DELETE CASCADE
);
```

```
INSERT INTO
    Grade
VALUES
    (1, 'COMP1036', 35),
    (1, 'COMP1048', 50),
    (2, 'COMP1048', 65),
    (2, 'COMP1038', 70),
    (3, 'COMP1036', 35),
    (3, 'COMP1038', 65),
    (6, 'COMP1038', 55),
    (6, 'COMP1099', 68);
```

Example (3/4): (Finally) We can DELETE a Student

```
PRAGMA foreign_keys = ON;
```

```
DELETE FROM
    Student
WHERE sID = 1;
```

```
SELECT * FROM Student;
```

sID	firstName	lastName
2	Jane	Doe
3	Mary	Jones
4	Joe	Bloggs

Table 19: The Student table after DELETE

Example (4/4): Caution! CASCADE will also DELETE the Grade!

```
SELECT * FROM Grade;
```

sID	mCode	grade
2	COMP1048	65
2	COMP1038	70
3	COMP1036	35
3	COMP1038	65
6	COMP1038	55
6	COMP1099	68

Table 20: The Grade table after DELETE from Student (!!). Practice caution when using CASCADE.

Transactions

Transactions

- A transaction is a sequence of SQL statements that are treated as a single unit.
    - Either all of the statements are executed, or none of them are.
- Transactions are used to ensure that the database is in a consistent state after the transaction is completed.
    - For example, if a transaction updates two tables, and one of the updates fails, the database should be left in the same state as before the transaction was started.

ACID Properties

- SQLite is a transactional database, which guarantees that all transactions are ACID compliant.
- This means, that SQLite guarantees that all transactions are:
    - **A**tomic - When a transaction is committed, all of the changes are saved to the database.
    - **C**onsistent - Guarantees that the database is always in a valid state.
    - **I**solated - A pending transaction will not affect other transactions.
    - **D**urable - Once a transaction has been committed, it will remain so, even in the event of a system failure.
- SQLite guarantees that all transactions are ACID compliant even if the transaction is interrupted by a power failure or system crash.

Transaction Syntax

- The syntax for a transaction is:
    - `BEGIN TRANSACTION;`
    - `-- SQL statements`
    - `COMMIT;`
- The `BEGIN TRANSACTION` statement starts a transaction.
- The `COMMIT` statement commits the transaction, which means that the changes are saved to the database.
- If any of the SQL statements in the transaction fail, the `ROLLBACK` statement can be used to undo the changes.
    - `ROLLBACK;`

Example: A Successful Transaction

```sql
BEGIN TRANSACTION;
```

```sql
INSERT INTO
    Student
VALUES
    (5, 'Jane', 'Smith');
```

```sql
-- Commit the changes to the database:
COMMIT;
```

```sql
SELECT * FROM Student;
```

sID	firstName	lastName
2	Jane	Doe
3	Mary	Jones
4	Joe	Bloggs
5	Jane	Smith

Table 21: The Student table after the transaction

Example: Rolling Back a Transaction

```sql
BEGIN TRANSACTION;
```

```sql
INSERT INTO
    Student
VALUES
    (6, 'Adam', 'Smith');
```

```sql
-- Rollback the changes to the database:
ROLLBACK;
```

```sql
SELECT * FROM Student;
```

sID	firstName	lastName
2	Jane	Doe
3	Mary	Jones
4	Joe	Bloggs
5	Jane	Smith

Table 22: The Student table after the transaction. Note that the changes have been rolled back, and the new student has not been added.

References

Online Tutorials

These are clickable links to the online tutorials:

- Join Operators
- `Update`
- `Delete`
- Transactions
- A Visual Explanation of SQL Joins

Textbooks and Documentation

- Chapter 5 and 22 of the Databases textbook.
- SQLite Transactions
- SQLite Joins

Mohan, Chandrasekaran, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.” *ACM Transactions on Database Systems (TODS)* 17 (1): 94–162.