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

COMP1048: Databases and Interfaces (2024-2025)

Matthew Pike and Yuan Yao

Overview

This Lecture

- In this lecture, we will cover:
  - Using `JOIN` to combine data across multiple tables.
  - Modifying data in existing tables with `UPDATE`.
  - Removing data from tables with `DELETE`.
  - Combining multiple SQL statements into a single transaction.
  - Motivation and practical use cases for transactions.
  - The ACID properties governing 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`

- Data is often split across multiple tables to reduce redundancy and improve data integrity—a concept we will explore further in the next lecture.
- This structure means we need a way to combine data from multiple tables in our queries—this is where `JOIN` becomes essential.
- The `JOIN` operation allows us to combine rows from two or more tables based on a common column, typically defined by `FOREIGN KEY` constraints.
- `JOIN` is a powerful feature of SQL and is one of the primary reasons for SQL's widespread use. It enables efficient and maintainable data storage, while still allowing us to combine data from multiple tables when needed.

Types of JOIN

- There are several types of `JOIN`:
  - `CROSS JOIN`
  - `INNER JOIN`
  - `LEFT JOIN` and `RIGHT JOIN`
  - `NATURAL JOIN`
  - `FULL OUTER JOIN`
- Understanding the differences between these types of `JOIN` is crucial, as each can produce very different results.

CROSS JOIN

Proceed with Caution

If your solution is using `CROSS JOIN`, it is likely that there is a simpler approach!

- The `CROSS JOIN` returns the Cartesian product of two tables.
  - This means that every row from the first table is combined with every row from the second table.
  - As a result, the output may include rows of unrelated or nonsensical data.
- Syntax for a `CROSS JOIN`:
  - `SELECT * FROM table1 CROSS JOIN table2;`
  - Equivalent to:
    - `SELECT * FROM table1, table2;`
- `CROSS JOIN` is rarely used in practice, as it can produce an overwhelming number of rows.
  - To limit the number of rows returned, a `WHERE` clause can be applied.

Example: CROSS JOIN

The `CROSS JOIN` of `Student` and `Module`

The `CROSS JOIN` operation generates a Cartesian product, meaning it combines every row from `Student` with every row from `Module`. This can result in data that is not logically connected or meaningful (e.g., students enrolled in irrelevant modules).

```sql
SELECT * FROM Student CROSS JOIN Module LIMIT 5;
```

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

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

Using SELECT without JOIN can achieve the same Cartesian product effect as CROSS JOIN. However, be mindful of unintended large results without a WHERE clause.

```sql
SELECT *
FROM Student, Module
LIMIT 5;
```

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

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

Aside: Ambiguous Column Names

- When using `SELECT` with multiple tables or `JOIN`, we may encounter ambiguous column names.
- For example, if we `SELECT` from both `Student` and `Grade`, both tables contain a column named `sID`.
- This can lead to errors or unexpected results. For instance, the following query will fail:
    - `SELECT sID FROM Student, Grade;`
- This returns an error: `Parse error: ambiguous column name: sID`. To resolve this, specify the table for each ambiguous column:
    - `SELECT Student.sID FROM Student, Grade;`
- Specifying table names not only resolves ambiguity but also improves readability, making it easier to interpret and debug queries.

Example: SELECT from Multiple Tables

```sql
SELECT
    -- Ambiguous: sID is in Student and Grade
    Student.sID,
    -- Ambiguous: mCode is in Module and Grade
    Module.mCode,
    -- Not ambiguous: grade is only in Grade
    grade
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`

ℹ `INNER JOIN` is the default `JOIN`

`INNER JOIN` is the most commonly used type of `JOIN` and serves as the default in SQL. We can simply use `JOIN` instead of `INNER JOIN`, and the query will yield the same result.

- `INNER JOIN` is perhaps the most commonly used type of `JOIN`, returning 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 returns 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 by specifying the join condition for each table.

```sql
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, avoid using `RIGHT JOIN` as it is less readable than `LEFT JOIN`. Instead, use `LEFT JOIN` and swap the order of the tables. Additionally, `RIGHT JOIN` is not supported in older versions of SQLite (including the version used in these labs).

- Often, we want to return all rows from one table, even if there is no match in the other table. For example, we may want to display all students, even if they haven't enrolled in any modules.
- `LEFT` and `RIGHT` joins allow us to do this with 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`, and it is generally recommended to use `LEFT JOIN` exclusively.

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`, `Grade`, and `Module`. Note that students without grades or modules appear with `NULL` values.

Going from RIGHT JOIN to LEFT JOIN

```sql
-- RIGHT joins can be 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`

> **Joining Multiple Tables**
>
> When joining multiple tables, the joins occur in the order specified in the query. For example, in `FROM A JOIN B JOIN C`, `A` is first joined with `B`, and the resulting table is then joined with `C`.

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

Example: NATURAL JOIN

```sql
SELECT
    Student.sID,
    Student.lastName AS "Last",
    Grade.grade AS "Grade"
FROM
    Student
    -- Automatically joins on 'sID'
    -- as it is present in both tables
    NATURAL JOIN
    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
    -- 'Student' with 'Grade' on 'sID'
    NATURAL JOIN
    Grade
    -- Joins resulting table
    -- with 'Module' on 'mCode'
    NATURAL JOIN
    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`
>
> `FULL OUTER JOIN` is only supported in SQLite version 3.39.0 and above.

- `FULL OUTER JOIN` returns all rows from both tables, including those where the join condition is not met.
- Rows from the left table with no match in the right table are returned with `NULL` values for the right table's columns.
- Rows from the right table with no match in the left table are returned with `NULL` values for the left table's columns.
- 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's often updated, deleted, or appended with new data.
- The `UPDATE` statement allows us to modify existing records in a table without needing to delete and reinsert the record.
- `UPDATE` can be used to update a single record or multiple records, using the following syntax:
    - `UPDATE table_name SET column1 = value1, column2 = value2, ... [WHERE condition];`
    - The `WHERE` clause is optional; if omitted, all records in the table will be updated.
    - Multiple columns and values can be specified within the `SET` clause.
- The `UPDATE` statement can also reference other column values within the same row.
    - For example, `UPDATE table SET column1 = column1 + 1;` increments `column1` by 1 for each row.

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 executing the `UPDATE` statement

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 outdated or unnecessary, 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 omitted, all records in the table will be deleted.
- The `DELETE` statement typically returns the number of rows that were deleted.

Example: DELETE Statement

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

```sql
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 deleting records where `sID = 3`

Referential Integrity

Considering Referential Integrity

Referential Integrity and SQLite

By default, SQLite does not enforce referential integrity. To enable it, use: `PRAGMA foreign_keys = ON;`.

- When modifying (updating or deleting) data in a table referenced by a `FOREIGN KEY` constraint, we must consider the impact on related tables—this is known as referential integrity.
- For example, if we delete a student from the `Student` table, we need to decide what happens in the `Grade` table:
  - Should we delete the student's grades?
    - SQL: `ON DELETE CASCADE`.
  - Should we set the student's grades to `NULL`?
    - SQL: `ON DELETE SET NULL`.
  - Should we prevent the student from being deleted altogether?
    - SQL: `ON DELETE NO ACTION` (the default in SQLite).

Example (1/4): DELETEing a Student

- If not specified (as in our `CREATE` statement for `Grade`), SQLite sets the `ON DELETE` action to `NO ACTION` by default.
  - 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 related data.

```sql
PRAGMA foreign_keys = ON;
```

```sql
-- This will fail, as there are rows
-- in the `Grade` table that reference
-- student with the sID of 1.
DELETE FROM
    Student
WHERE sID = 1;
```

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

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

Table 18: The `Student` table after attempted `DELETE` (constraint prevented deletion)

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

```sql
PRAGMA foreign_keys = ON;
```

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

```sql
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 treated as a single unit.
    - Either all of the statements are executed, or none of them are.
- Transactions ensure the database remains in a consistent state upon completion.
    - For example, if a transaction updates two tables but one update fails, the database reverts to its state before the transaction began.

ACID Properties

- SQLite is a transactional database, guaranteeing that all transactions are ACID compliant:
  - **A**tomic - When a transaction is committed, all changes are saved to the database.
  - **C**onsistent - Ensures the database is always in a valid state.
  - **I**solated - A pending transaction does not affect other transactions.
  - **D**urable - Once committed, a transaction remains so, even in the event of a system failure.
- SQLite maintains ACID compliance for all transactions, even if interrupted by power failure or system crash.

Transaction Syntax

Automatic Transaction Wrapping in SQLite

In SQLite, each standalone `INSERT`, `UPDATE`, or `DELETE` is automatically wrapped in an implicit transaction, which is committed once the statement finishes. For more control, especially with multiple statements, we can use `BEGIN TRANSACTION;` and `COMMIT;` explicitly to manage transactions as a single unit.

```
BEGIN TRANSACTION;
-- SQL statements
COMMIT;
```

- `BEGIN TRANSACTION` starts the transaction.
- `COMMIT` commits the transaction, saving all changes to the database.
- If any SQL statement in the transaction fails, `ROLLBACK` can be used to undo all changes made within the transaction:
    - `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 has been committed

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

Learning Resources

Online Tutorials

These are clickable links to the online tutorials:

- Join Operators - https://www.sqlitetutorial.net/sqlite-join/
- `Update` - https://www.sqlitetutorial.net/sqlite-update/
- `Delete` - https://www.sqlitetutorial.net/sqlite-delete/
- Transactions - https://www.sqlitetutorial.net/sqlite-transaction/
- A Visual Explanation of SQL Joins - https://blog.codinghorror.com/a-visual-explanation-of-sql-joins/

Textbooks and Documentation

- Chapter 5 and 22 of the Databases textbook.
- SQLite Transactions - https://www.sqlite.org/lang_transaction.html
- SQLite Joins - https://www.sqlite.org/syntax/join-operator.html

Reference Materials

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwarz, P. (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.