Block 03

Introduction to Relational Database Management Systems

VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here: https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-3

Database Management Systems

A *database* is a collection of data, typically stored electronically in a computer system.

A *database management system (DBMS)* is software package serving as an interface between the database and its end users or programs. DBMS allows users to

- Store
- Retrieve / query
- Update
- Manage
- Control access to

data in a database.

Databases are needed because they allow

- Storage of massive amounts of data
- Data access to multiple users
- Simultaneous change of data (concurrency)
- Efficient manipulation of large amounts of data
- Storage and access to the data in a safe, convenient, and reliable way
    - Safe: Data are consistent even if something bad happens (e.g. hardware or software failure)
    - Convenient: It is easy to manipulate or query a large amount of data (e.g. through a high level query language)
    - Reliable: Chance for the database to be down is very low.

Relational Databases

A *relational database* is a type of database, where

- All data is represented in terms of tuples and stored in tables with columns and rows
- The structure on how the data is stored is pre-defined and imposed by the programmer
- Structured data can be held.

A *relational database management system (RDBMS)* is a DBMS designed specifically for relational databases. *Structured Query Language (SQL)* is one of the most widely used database languages designed for managing data held in a RDBMS. We will formally introduce SQL later this week.

Nonrelational Databases

An alternative to relational database model is the nonrelational database model. Nonrelational databases, also know as NoSQL, store data in a different format to relational tables. Their key feature is that they allow unstructured and semi-structured data to be stored and manipulated. There are many different kinds of NoSQL databases, including:

- **Graph database**: A graph database stores data in terms of entities (or nodes) and the relationships (or edges) between entities. It is suitable for data with relations are well represented as a graph
- **Document-oriented database**: A document-oriented database stores semi-structured data (or document-oriented information) in the form of JSON-like documents. Document databases store all information for a given object in a single instance in the database, and every stored object can be different from every other
    - Below shows a JSON-like document storing information about two books. Note that data is stored in nested key-value pairs (e.g. `year` is the key and `1995` is the value for the first book) and the information available for each book is different (e.g. `edition` and `description` are only available for the first book).

```
[
  {
      "year" : 1995,
      "title" : "An introduction to database systems",
      "info" : {
          "authors" : "C. J. Date",
          "edition" : 6,
          "subject" : "database management",
          "description" : "A comprehensive treatment of database
technology. Features of this edition include: a proposal for
rapprochement between object-oriented and relational technologies;
expanded treatment of distributed databases; and chapters on
functional dependencies, views, domains and missing information.",
          "publisher" : "Reading, Mass. : Addison-Wesley Pub. Co."
      }
  },
  {
      "year" : 2016,
```

```
        "title" : "Introduction to computation and programming using
    Python: with application to understanding data",
        "info" : {
            "authors" : "Guttag, John V",
            "subject" : ["Computer programming", "Python (Computer
    program language)"],
            "publisher" : "The MIT Press"
        }
    }
]
```

RDBMS Terminology

Relation/Table

In a relational database, a relation is a table, with its rows representing a set of records (called tuples). . For example, a university database may have a relation (table) called `Student` to store the information for all students, a table `Course` for the course information and a table `Grade` for the grade of students in different courses:

`Student`:

student_id	name	year
201921323	Ava Smith	2
201832220	Ben Johnson	3
202003219	Charlie Jones	1

`Course`:

course_id	name	capacity
ST101	Programming for Ddata Science	60
ST115	Managing and Visualising Data	60
ST207	Databases	30

`Grade`:

course_id	student_id	final_mark
ST101	202003219	47
ST115	201921323	92
ST115	202003219	67
ST207	201933222	73

Attribute

In relational databases, the *attribute* is a column in the table (or relation). Each attribute has a type (or domain). For example, the `Student` table above has the attributes

- `student_id` (for student ID)
- `name`
- `year`

with the corresponding types being string, string and integer.

Tuple

A *tuple* is a set of attribute values. In a relational database, a tuple is a row in a table. For example, each row in the *Student* table is a tuple storing the information for one student.

Schema

The *schema* is the description on how the database and the database tables are constructed. For example, the `Student` table can have the schema:

```
Students(student_id: string, name: string, year: integer)
```

Key

A primary key is the attribute used to uniquely identify a tuple, or the set of attributes whose combined values are unique. For the examples above, the primary keys are:

- `Student`: `student_id`
- `Course`: `course_id`
- `Grade`: `student_id` and `course_id`

A foreign key is an attribute or a set of attributes in a relational database table that provides a link between data in two tables. For example, `student_id` is the foreign key to link between the tables `Grade` and `Student`.

Useful Links and Resources

- What is NoSQL? from MongoDB to learn more about NoSQL databases
- What is a database? from Oracle
- From Wikipedia
  - Databases
  - Relational database
  - Document-oriented database
  - SQL
  - Foreign key

Introduction to SQL and SQLite

Structured Query Language

Structured query language (SQL) is one of the most widely used database languages designed for managing data held in a relational database management system (RDBMS). It is used to:

- Create (databases/tables)
- Manipulate (insert, update and delete tables/tuples)
- Query

a relational database.

SQLite

SQLite is the most widely used database engine. The other two popular database engines are MySQL and PostgreSQL. SQLite:

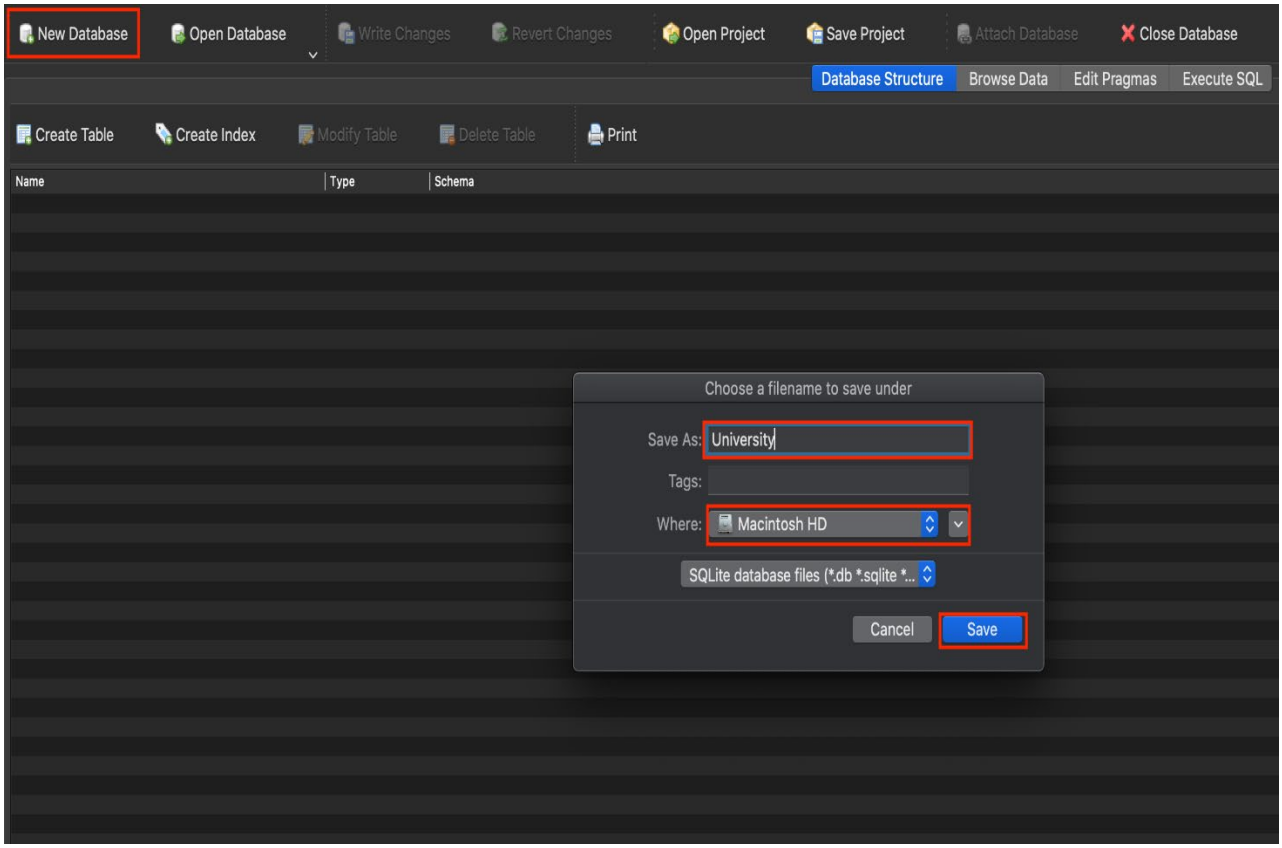
- Is *lite*
- Does not need configuration and uses little space
- Does not require a separate server process like other RDBMS systems do—SQLite reads and writes to a single file on your disk
- Is *self-contained* in the sense that it has very few dependencies
- Offres full-featured SQL.

Creating Databases

In this course we use **DB Browser for SQLite (DB4S)** to create, query and edit database files compatible with SQLite. You can download ot at the DB Browser for SQLite web pages.

In this document we consider a database `University`. We want to use it to store information about students, courses information and student grades.

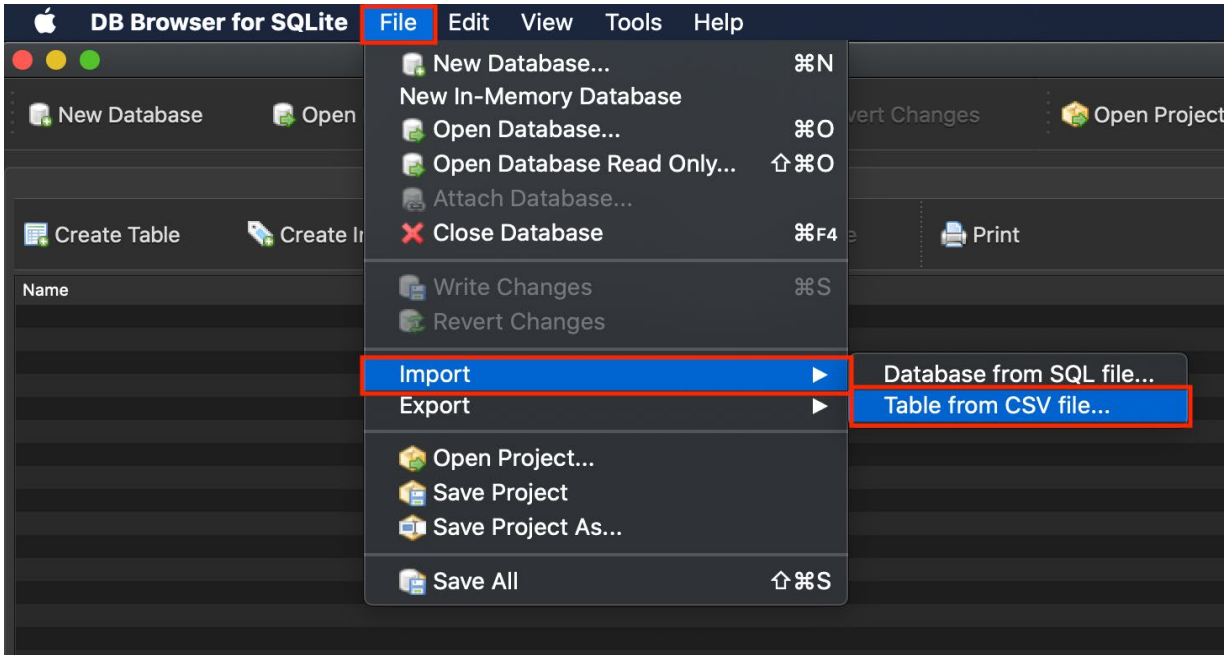
Click on `New Database` and use `University` as the name of the database. You can safely close the "Edit table definition" pop-up window by hitting `Cancel`.



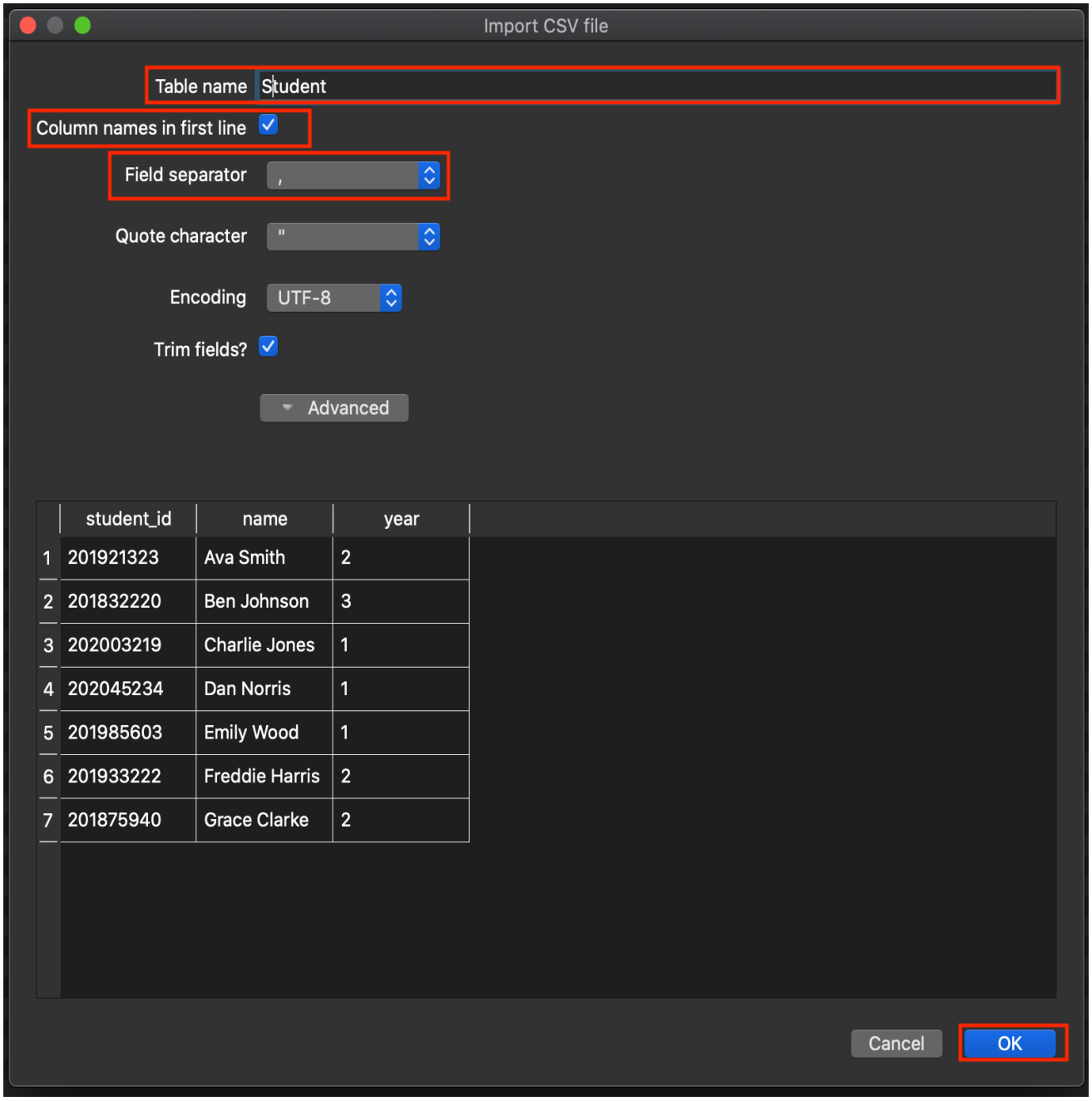
Screenshot on how to create a database

Add Tables to the Database From CSV Files

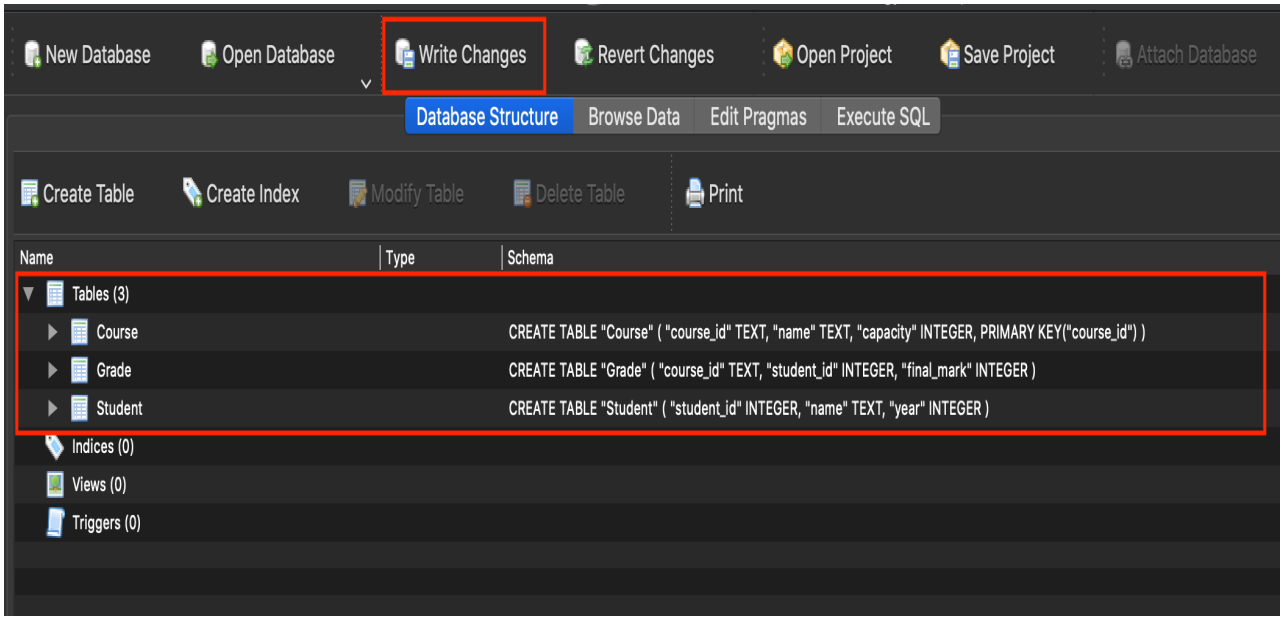
1. Click `File -> Import`.
2. Select the one of the CSV files `student.csv`, `grade.csv` and `course.csv`.
3. Check:
    - `Table name`: in this tutorial we capitalize the first letter in table names.
    - `Field separator`: set as `,`.
    - tick `Column names in first row`.
    - Click OK if everything is fine.
4. Repeat the process for the other two CSV files.
5. Click `Write Changes`



Screenshot on how to add tables to the database from an existing CSV file



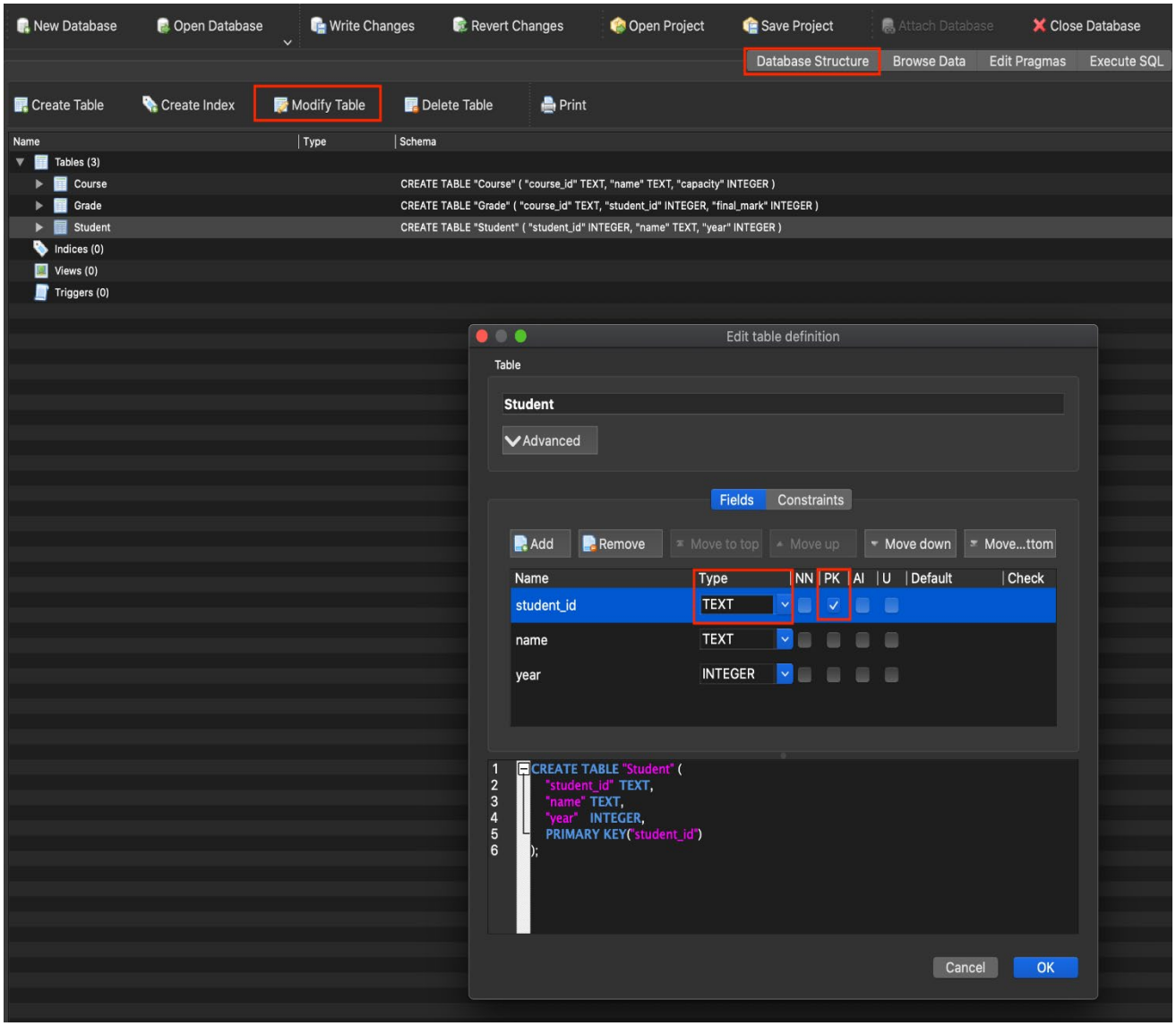
Screenshot on how to add tables to the database from an existing CSV file



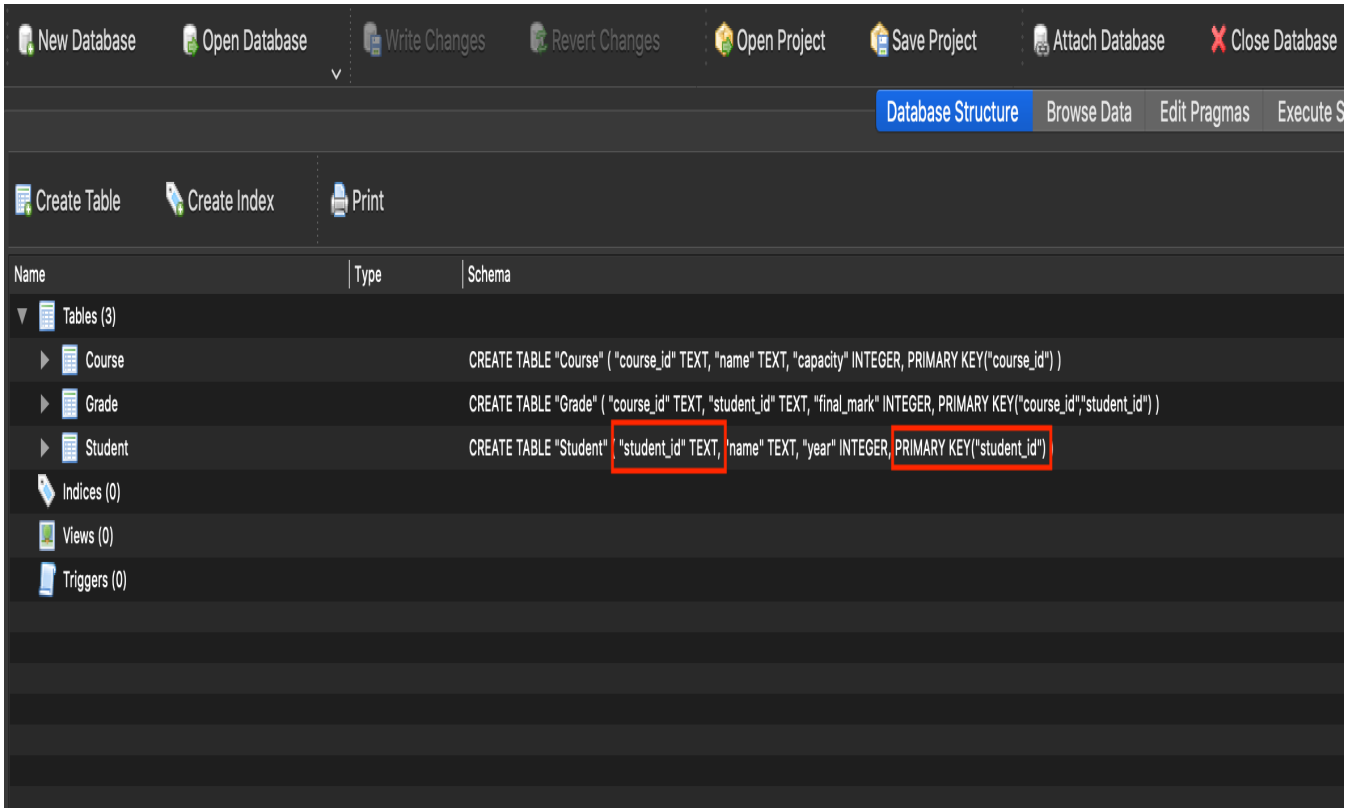
Screenshot on how to add tables to the database from an existing CSV file

Update the Schema of Tables

1. Click `Database Structure`.
2. Select `Student` and click `Modify Table`.
3. Set `student_id` with Type `TEXT` and select `PK` (primary key).
4. Click `ok`.
5. Similarly for `Course` table, set `course_id` as primary key. For `Grade` table, set `student_id` with Type `TEXT` and BOTH `course_id` and `student_id` as the primary key.
6. Click `Write Changes`.



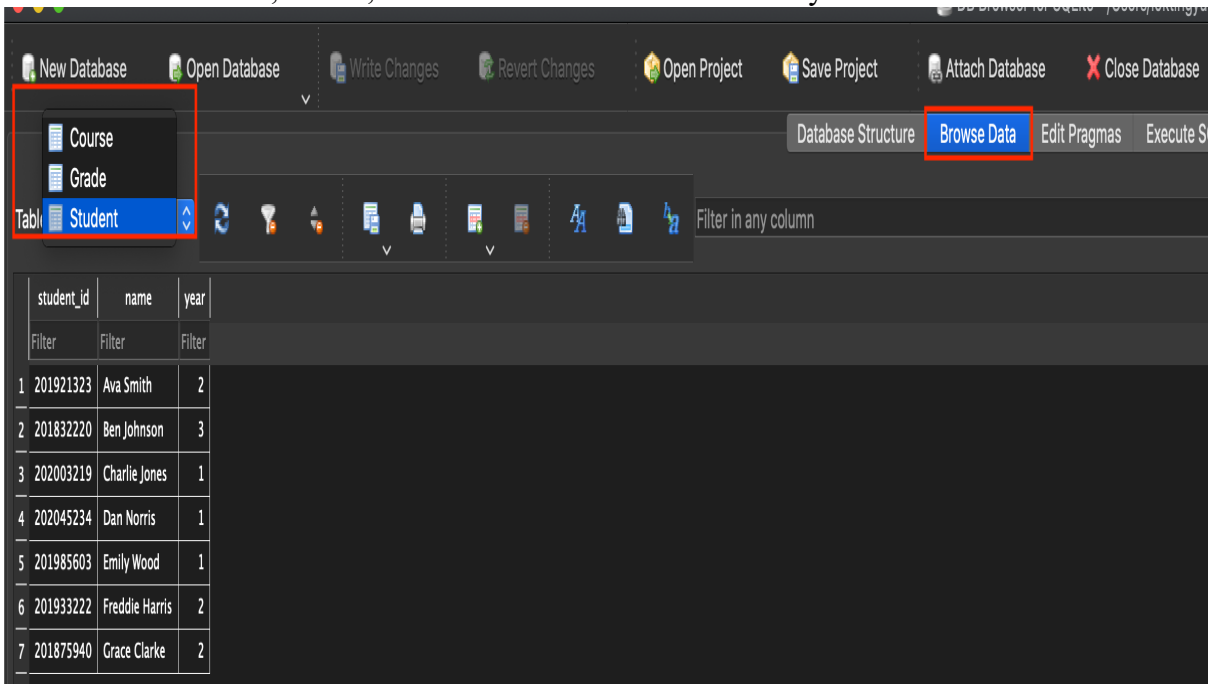
Screenshot on how to update the schema of the tables



Screenshot of the updated schemas

Browse Data

Click `Browse Data` to browse the data. When clicking on the list from `Table`, you should see three tables `Student`, `Grade`, and `Course`. Select the table that you want to browse.



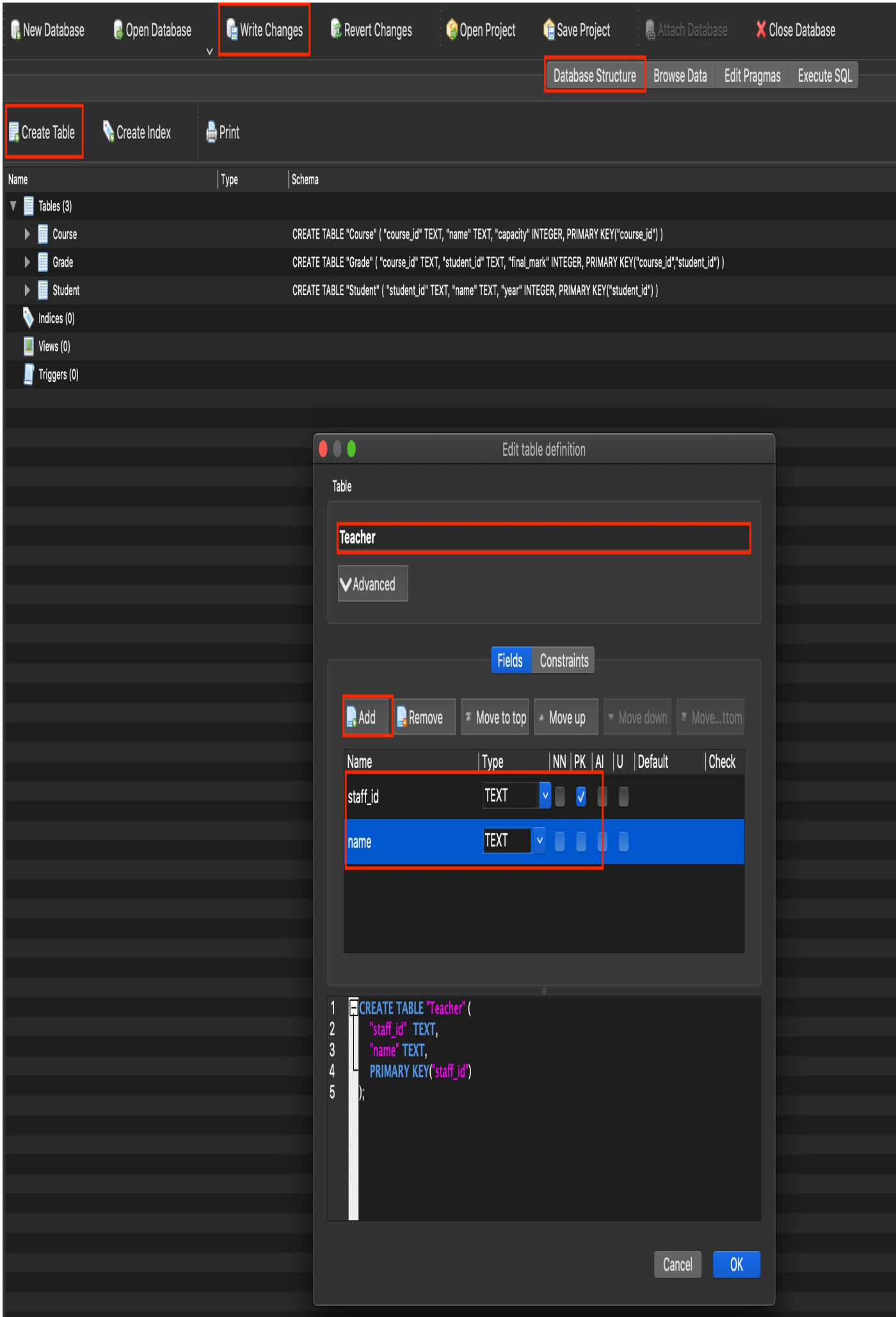
Screenshot on how to browse data

Manipulating Databases

Add a New Table

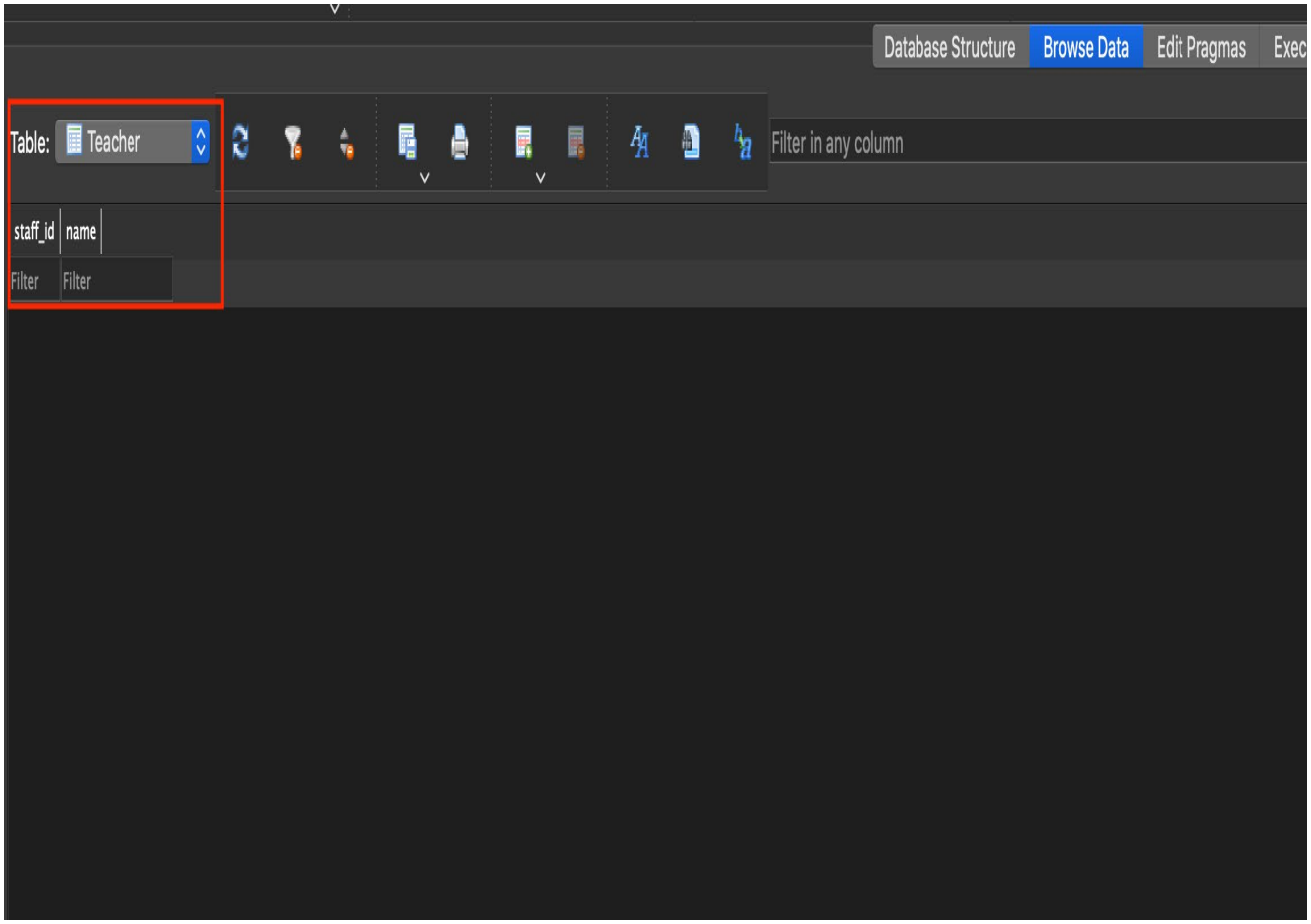
Add a new table `Teacher`:

1. Click `Database Structure`.
2. Click `Create Table`.
3. Write `Teacher` in the first blank for the name of the table.
4. Under the `Field`, click `Add` to create a new attribute. Type `staff_id` as Name and `TEXT` as Type. Select `PK`.
5. Under the `Field`, click `Add` to create a new attribute. Type `name` as Name and `TEXT` as Type.
6. Click `ok`.
7. Click `Write Changes`.


New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Close Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Create Table
Create Index
Print
Name
Type
Schema
Tables (3)
Course
CREATE TABLE "Course" ("course_id" TEXT, "name" TEXT, "capacity" INTEGER, PRIMARY KEY("course_id"))
Grade
CREATE TABLE "Grade" ("course_id" TEXT, "student_id" TEXT, "final_mark" INTEGER, PRIMARY KEY("course_id","student_id"))
Student
CREATE TABLE "Student" ("student_id" TEXT, "name" TEXT, "year" INTEGER, PRIMARY KEY("student_id"))
Indices (0)
Views (0)
Triggers (0)
Edit table definition
Table
Teacher
Advanced
Fields
Constraints
Add
Remove
Move to top
Move up
Move down
Move...ttom
Name
Type
NN
PK
AI
U
Default
Check
staff_id
TEXT
name
TEXT
1 CREATE TABLE "Teacher" (
2 "staff_id" TEXT,
3 "name" TEXT,
4 PRIMARY KEY("staff_id")
5);
Cancel
OK

Screenshot on how to add a new table

Click `Browse Data` to browse the data. When click on the list from Table, you should see four tables. Select the table `Teacher` and it will display an empty table.

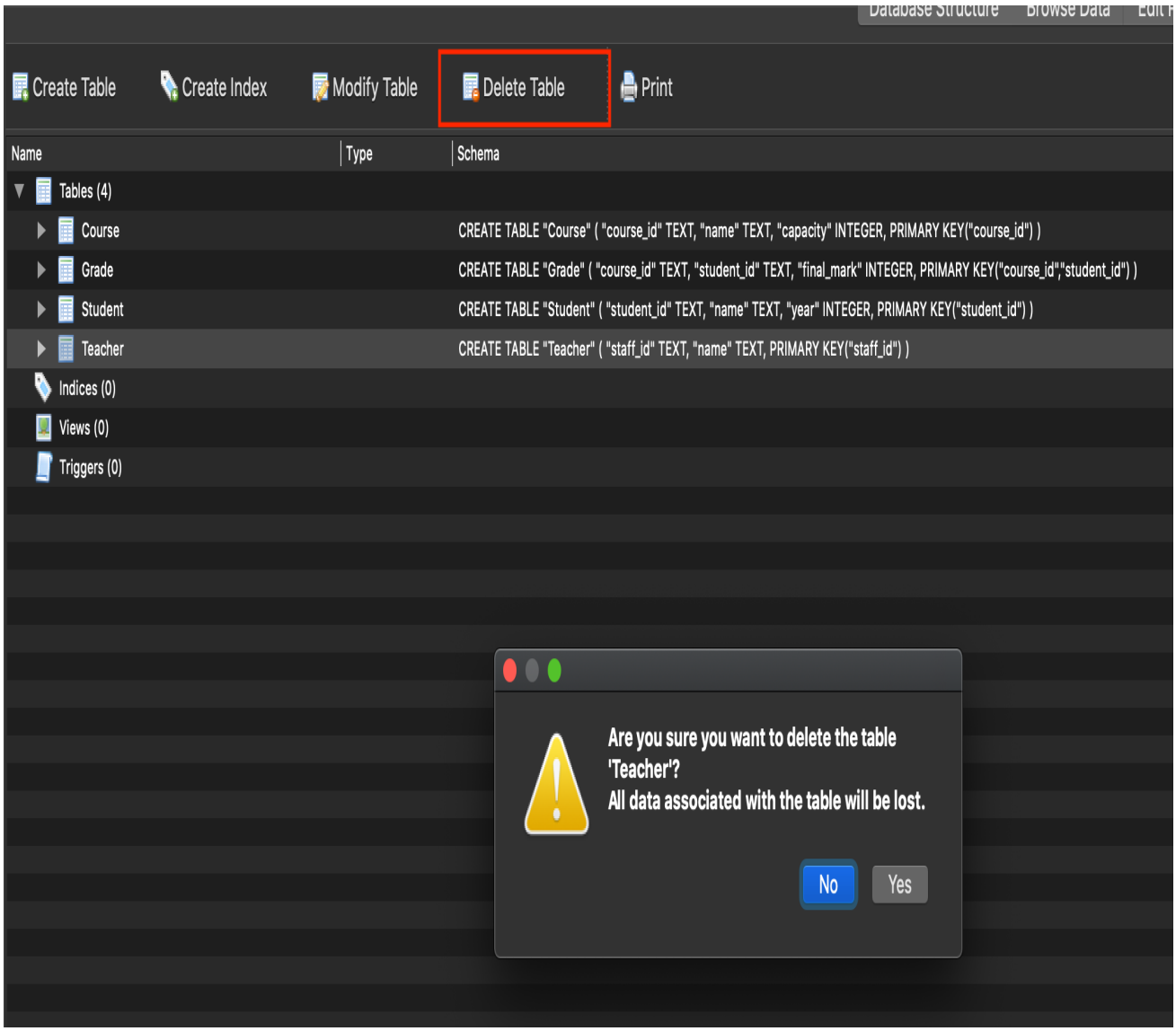


Screenshot of the newly created empty table

Delete a Table

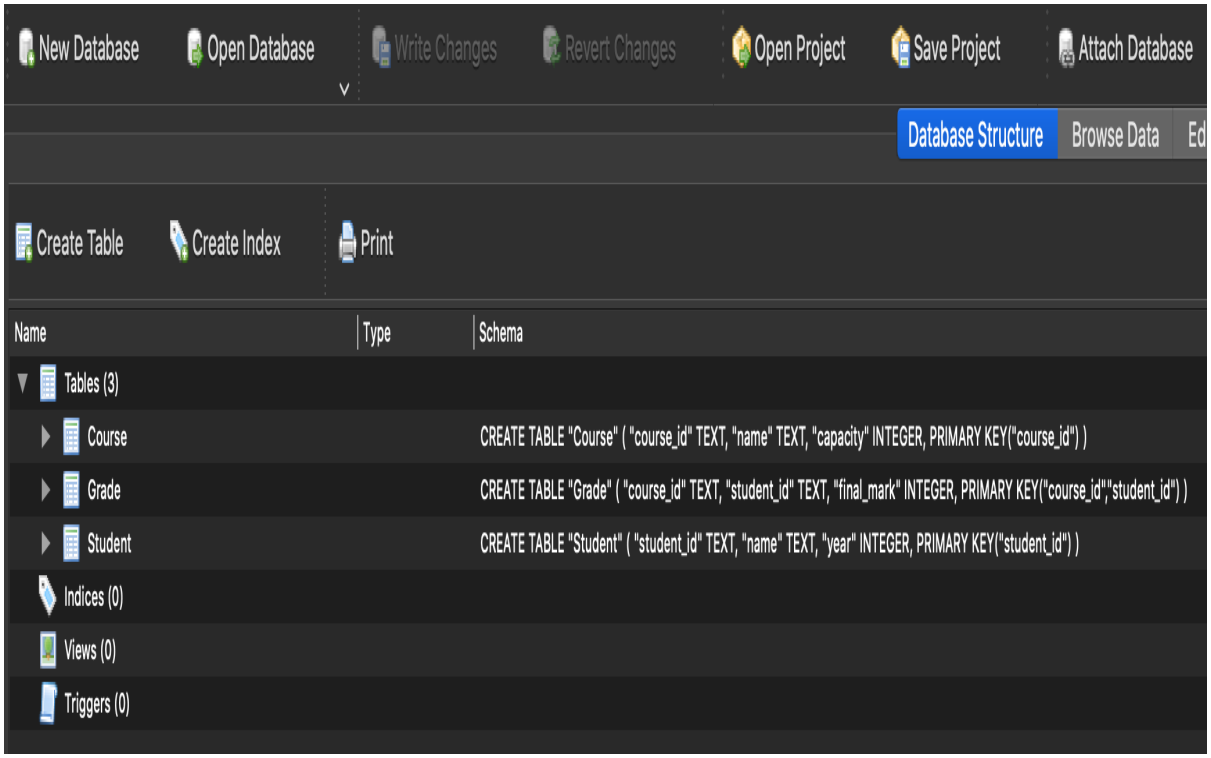
We can remove the `Teacher` table in the following way:

1. Click `Database Structure`.
2. Click on the `Teacher` table.
3. Click `Delete Table`.
4. Click `Write Changes`.



Screenshot on how to delete a table

Browse and now there are only three tables on the list from Table.

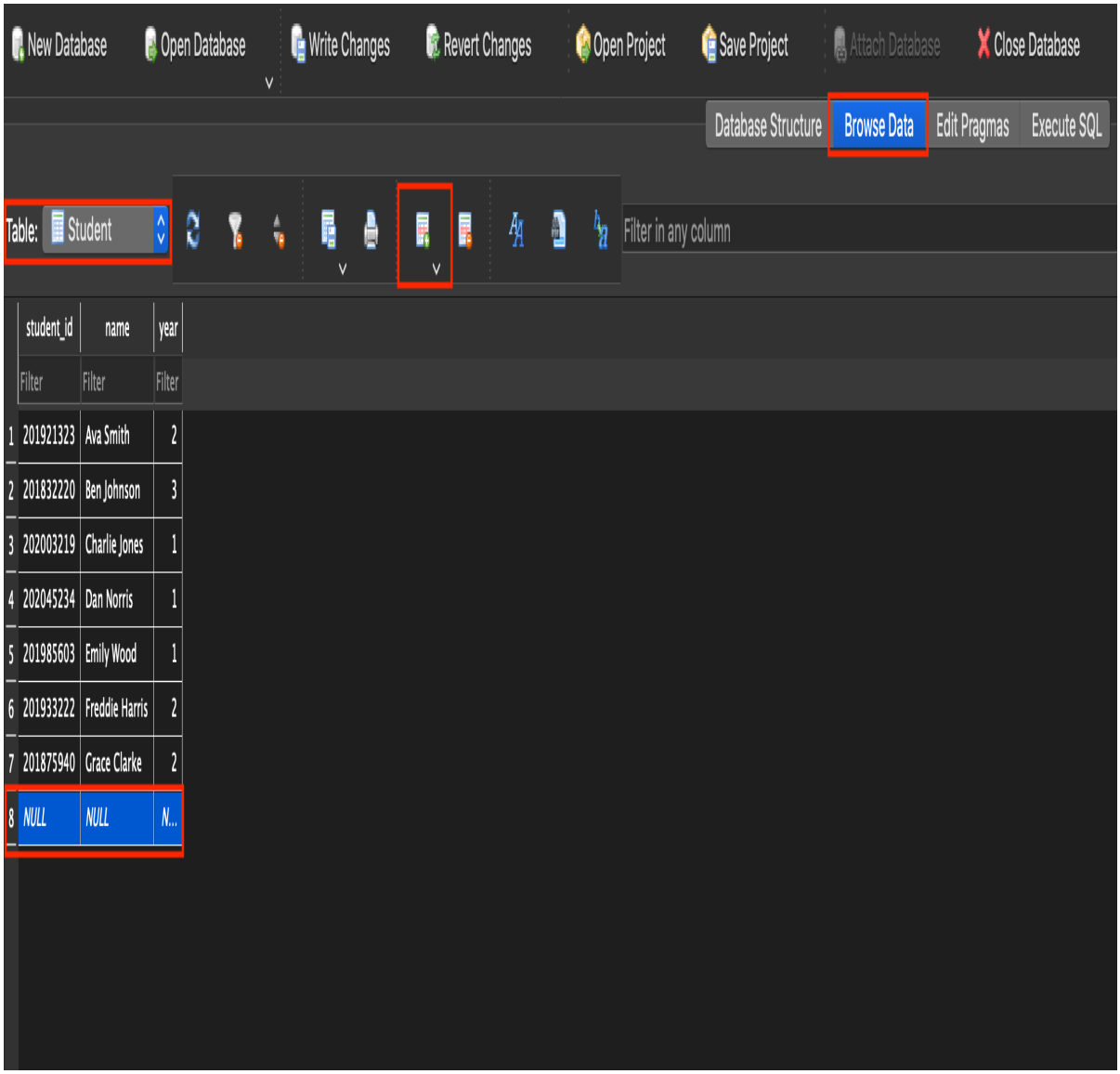


Screenshot of list of tables

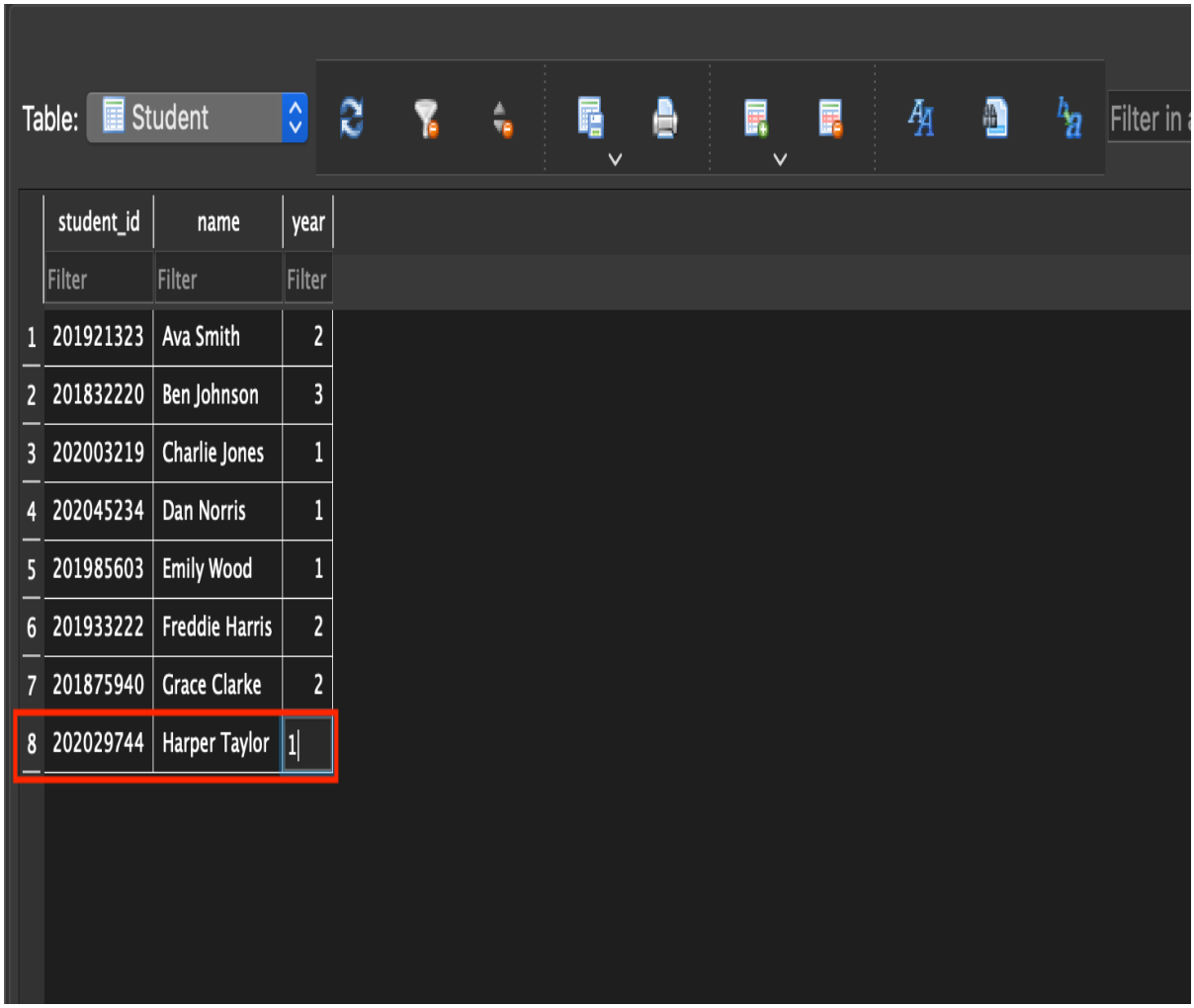
Insert Tuples/Rows

Insert the year 1 student Harper Taylor with student ID 202029744 to `Student`:

1. Click `Browse data`.
2. Select `Student` from the list from `Table`.
3. Click the button the add an empty row.
4. Manually put the information into the new row.
5. Click `Write Changes`.



Screenshot on how to insert a row



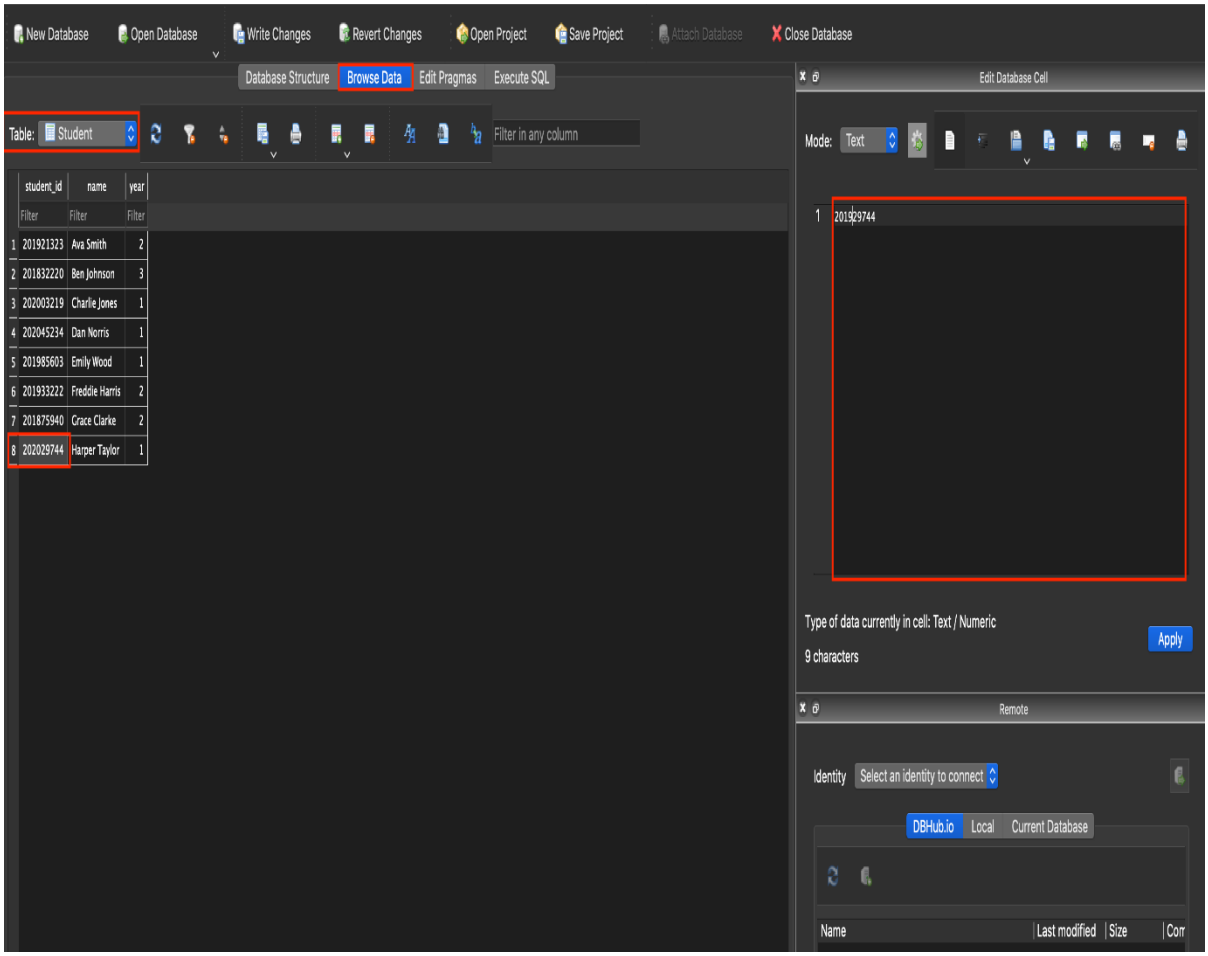
Screenshot on how to insert a row

Browse the table `Student` and makes sure the new student information is in the table.

Update Tuples/Rows

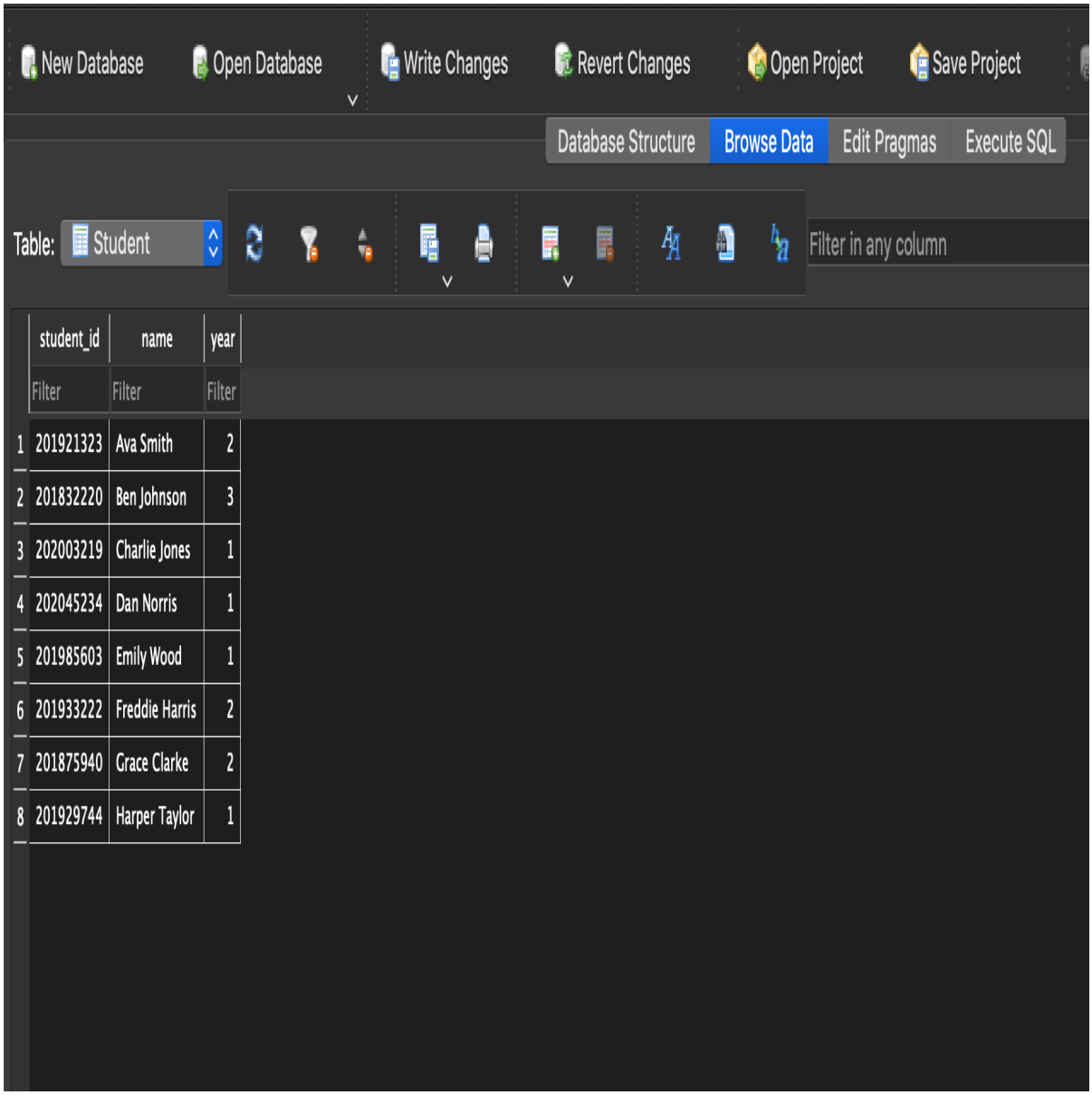
Update the student ID of student Harper Taylor to 201929744:

1. Click `Browse data`.
2. Select `Student` from the list from `Table`.
3. Click on `student_id` cell on the row with `name` "Harper Taylor."
4. In the window `Edit Database Cell`, change the value to `201929744`.
5. Click `Apply`.
6. Click `Write Changes`.



Screenshot on how to update a row

Browse the table `Student` and make sure the student information is updated in the table.

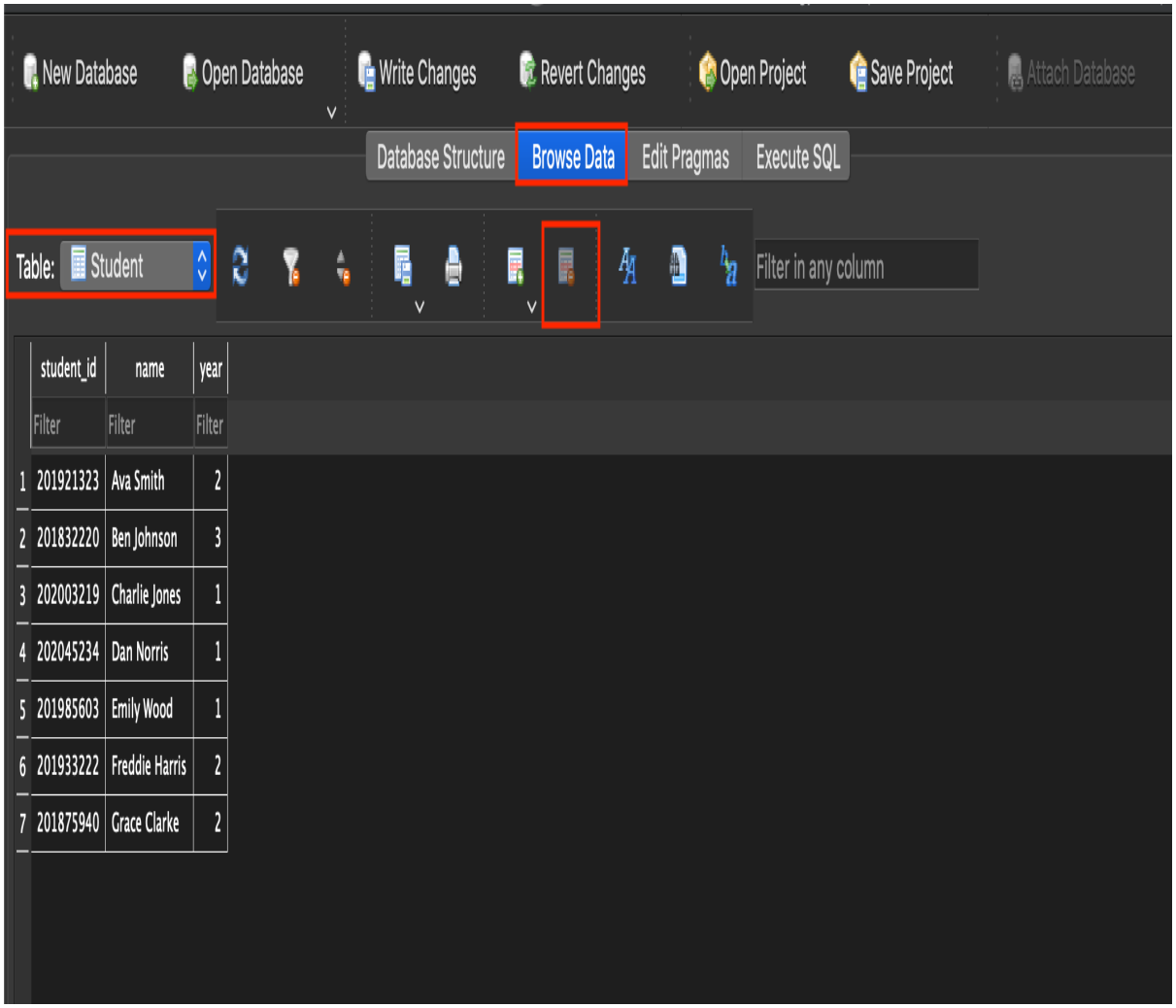


Screenshot of the table with a updated row

Delete Tuples/Rows

Delete the record for the student Harper Taylor from table `Student`:

1. Click `Browse data`.
2. Select `Student` from the list from `Table`.
3. Click on the row with the name “Harper Taylor.”
4. Click the button to remove the row.
5. Click `Write Changes`.



Screenshot on how to delete a row

Browse the table `Student` and make sure the row is deleted.

Useful Links and Resources

- What is SQLite: the official pages of SQLite
- DB Browser for SQLite: the official pages of DB Browser for SQLite
- From Wikipedia
  - SQL
  - SQLite

Basic SQL/SQLite Syntax and Queries

SQL Recap

Structured query language (SQL) is one of the most widely used database languages, designed for managing data held in a relational database management system (RDBMS). It is used to:

- Create (database/tables)
- Manipulate (insert, update and delete tables/tuples)
- Query

a relational database. SQL:

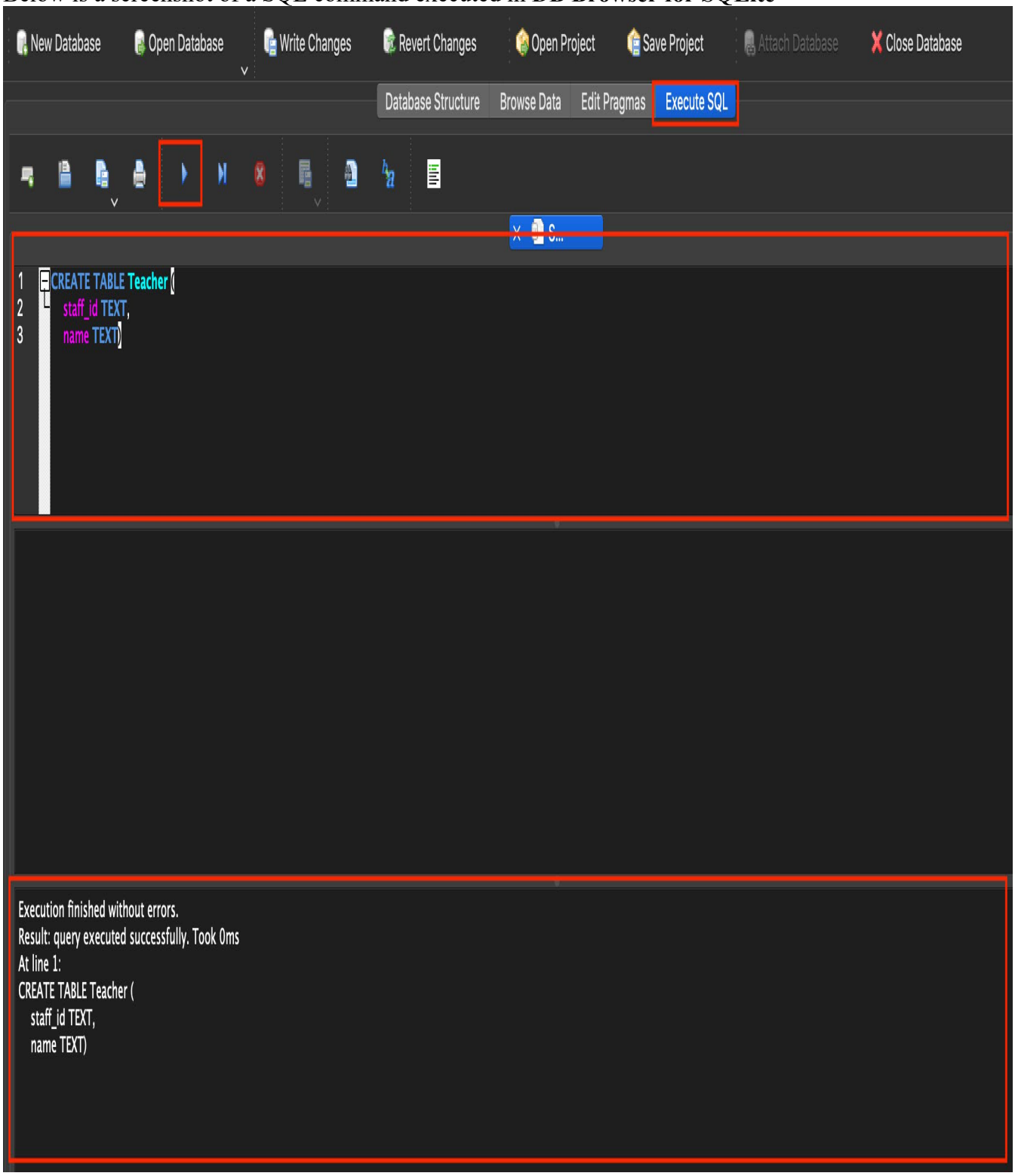
- Is easy to learn: SQL commands are like English sentences, which makes it easy to read and write SQL queries
- Has well-defined standards: American National Standards Institute (ANSI) and International Organization for Standardization (ISO) have officially adopted the standard “Database Language SQL” language definition.

Manipulating Databases

We have already seen how to use the **DB Browser for SQLite** graphical interface to add and delete tables and rows in a database. Here, we show how to do the same operations via SQL commands. We can run SQL code directly in **DB Browser for SQLite** in the following way:

1. Click on `Execute SQL` and type your SQL code in the top text box.
2. Click the run button to execute the SQL code.

Below is a screenshot of a SQL command executed in **DB Browser for SQLite**



In what follows we will work with the `University` database that we have created previously using **DB Browser for SQLite**.

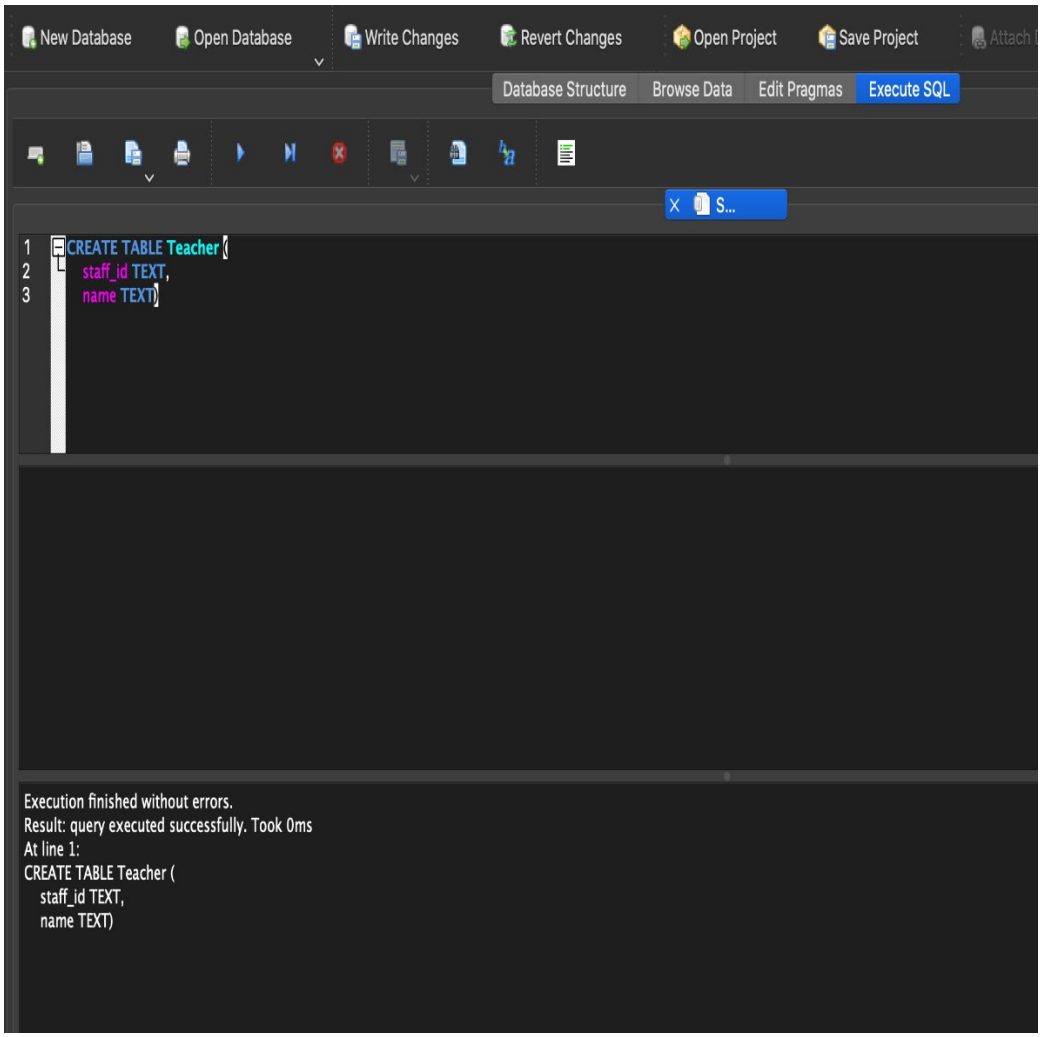
We open the database `University` in **DB Browse for SQLite** by clicking on `Open Database` and selecting the file `University.db`.

Add a Table

Add a new table `Teacher` by using the `CREATE TABLE` command:

```
CREATE TABLE Teacher (
    staff_id TEXT,
    name TEXT)
```

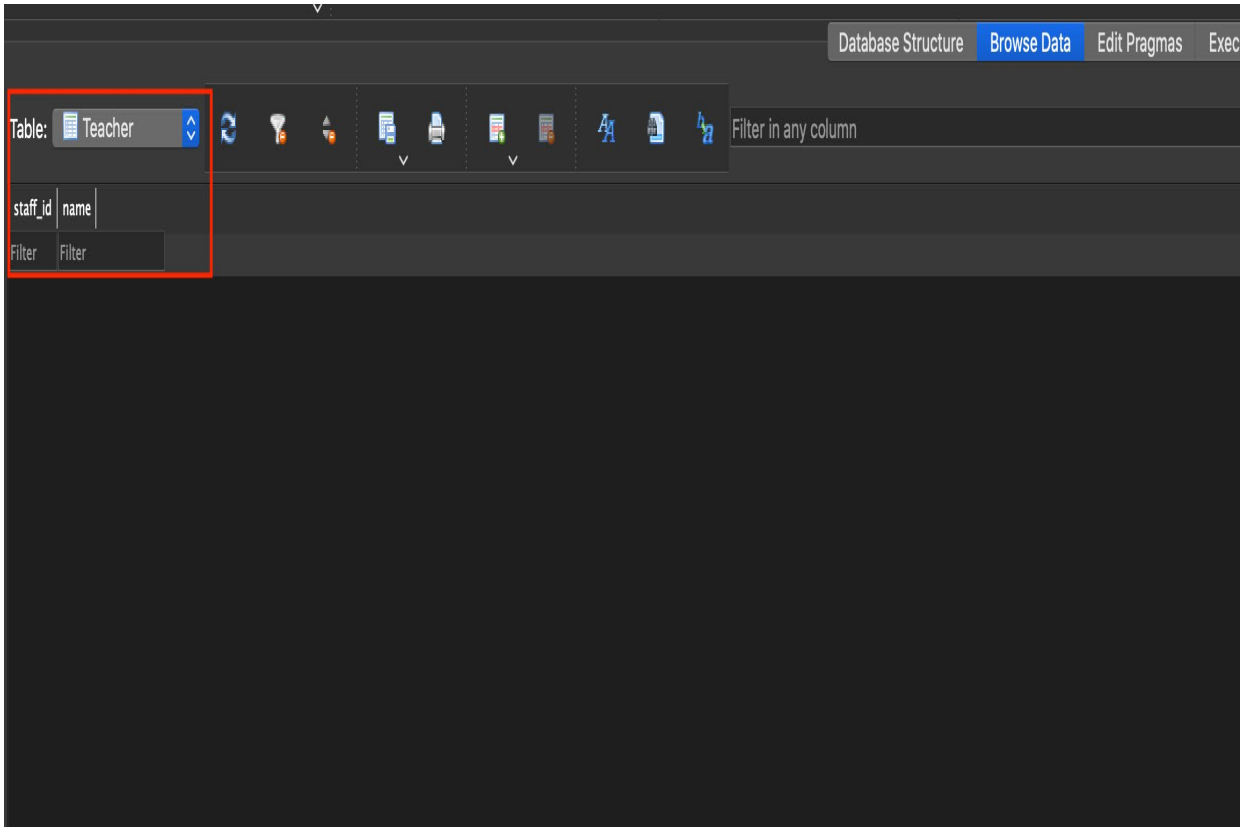
and hitting the execute button above the text box.



Screenshot on how to add a table using SQL

Click `Write Changes`.

Now, click `Browse Data` to browse the database. When you click on the list `Table`, you should get four tables. Browse the table `Teacher` and it will display an empty table with attributes “staff_id” and “name.”

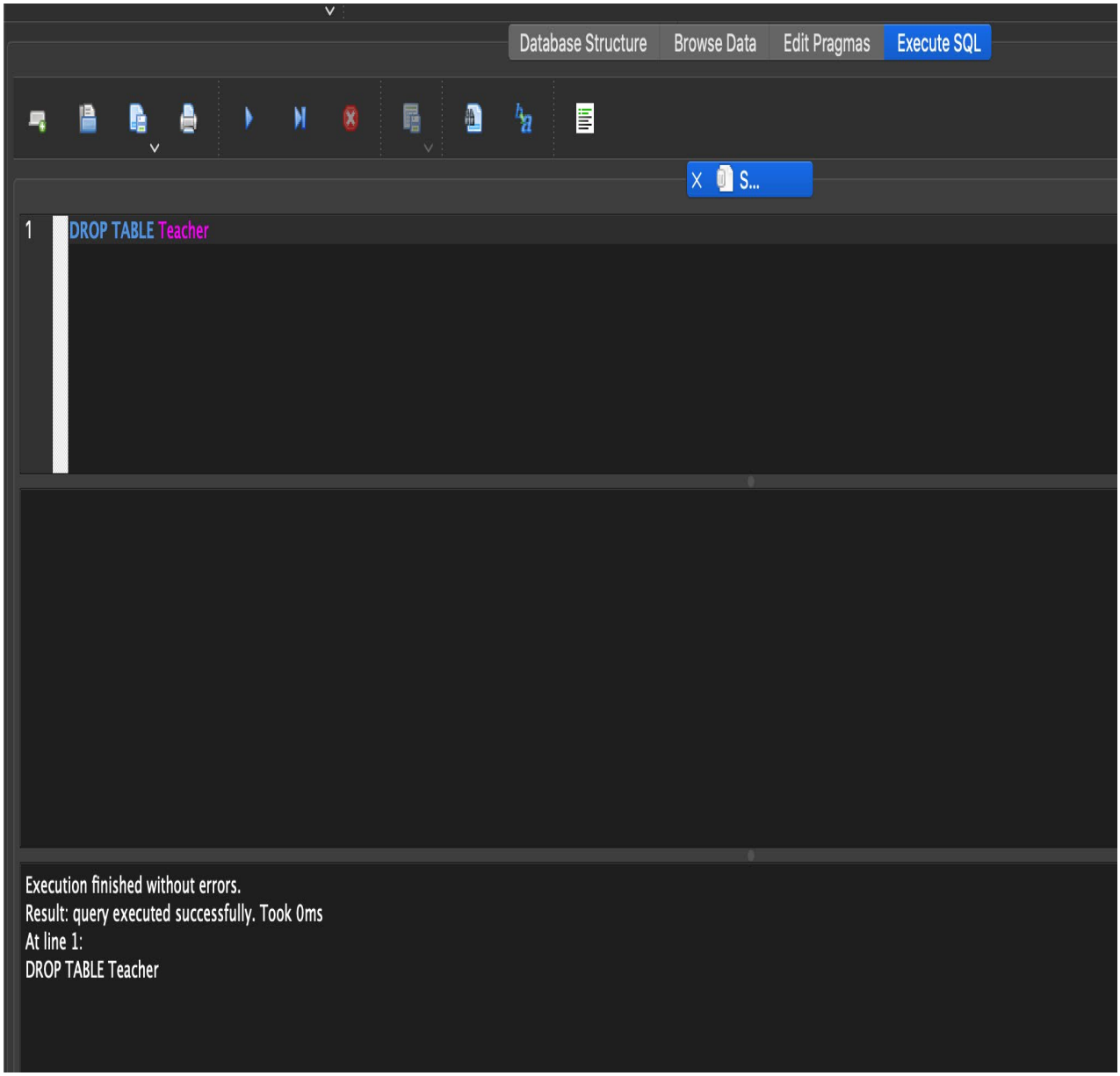


Screenshot of the newly created empty table

Delete a Table

We can remove the `Teacher` table by the `DROP TABLE` command:

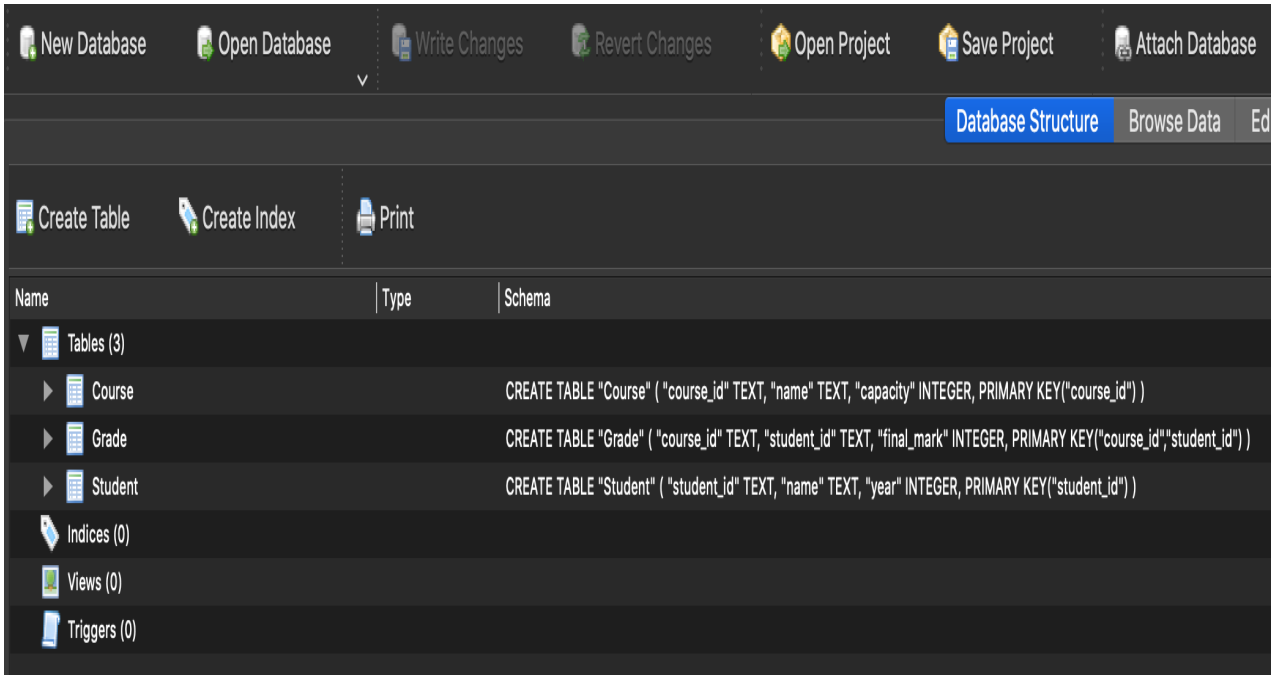
```
DROP TABLE Teacher
```



Screenshot on how to delete a table

and click `Write Changes`

If you browse now, you will see that there are only three tables left.

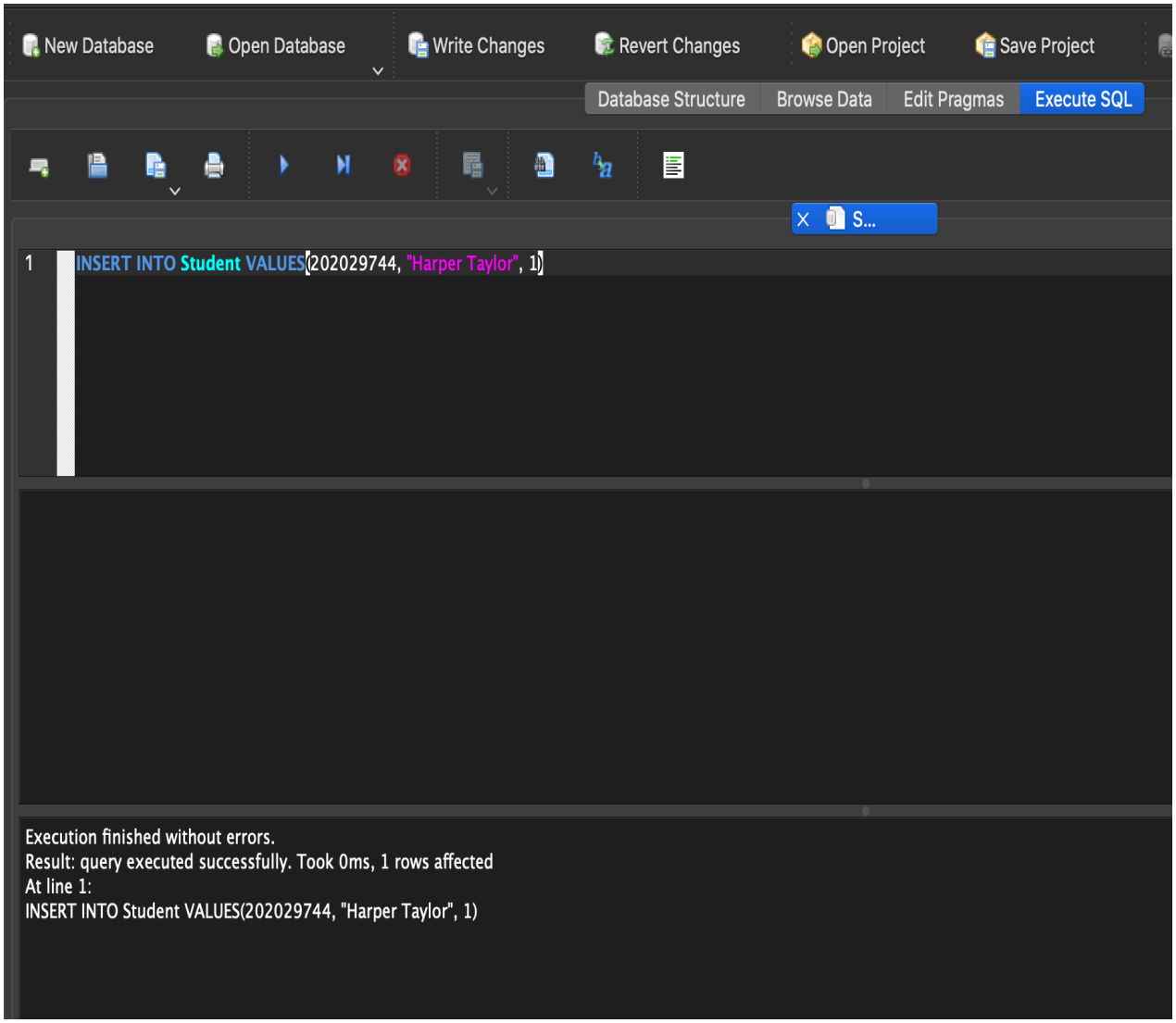


Screenshot of list of tables

Insert Tuples/Rows

Insert the year 1 student “Harper Taylor” with student ID 202029744 to `Student` by using the `INSERT INTO` command:

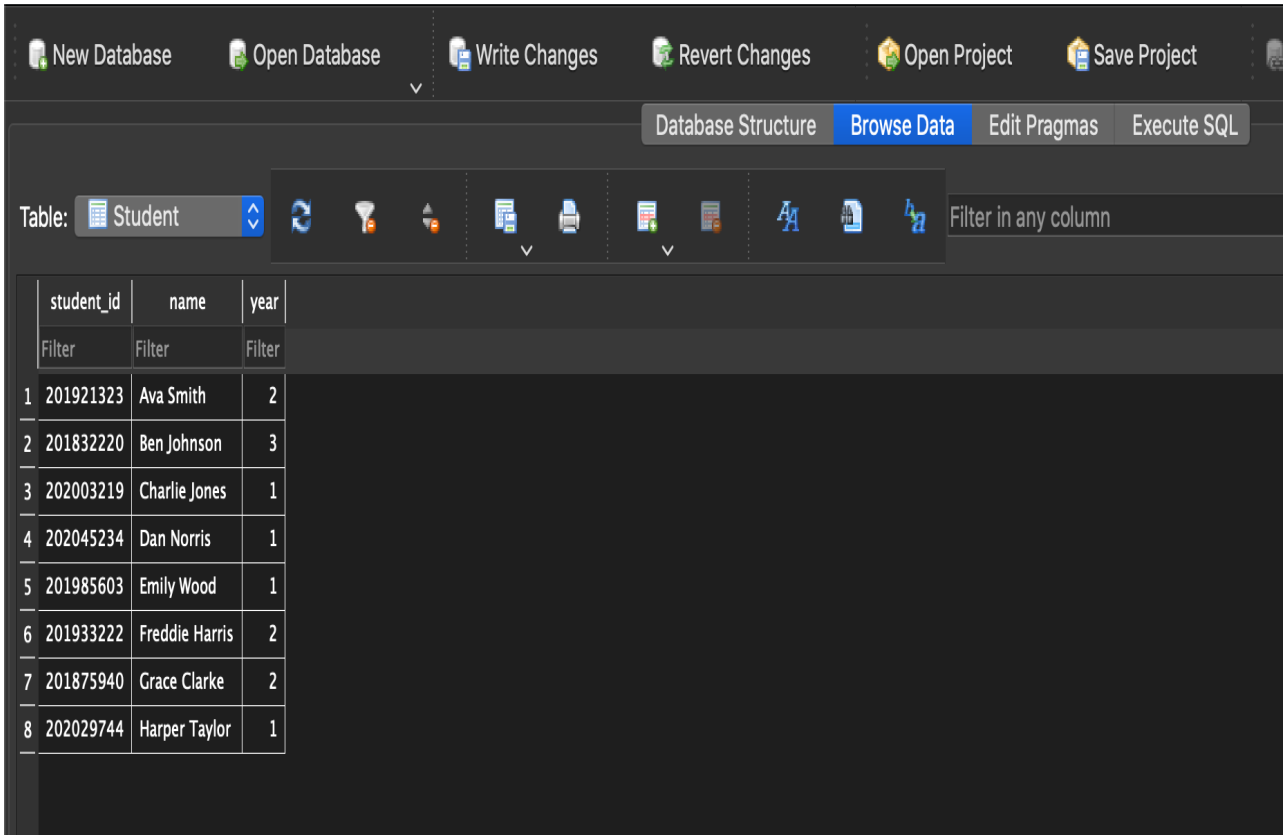
```
INSERT INTO Student VALUES(202029744, "Harper Taylor", 1)
```



Screenshot on how to insert a row

Click `Write Changes`.

Browse the table `Student` and make sure the new student information is in the table.



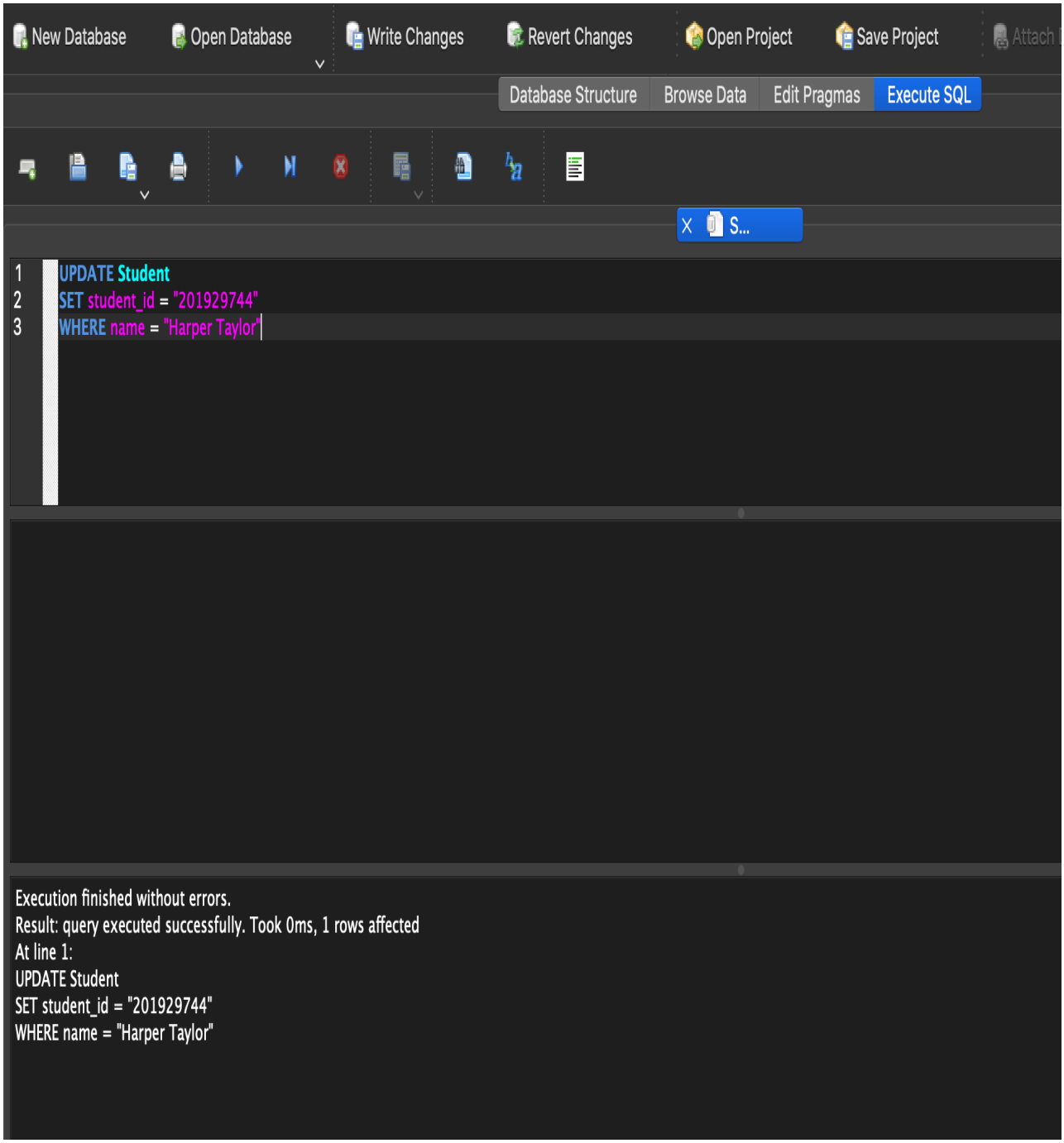
	student_id	name	year
	Filter	Filter	Filter
1	201921323	Ava Smith	2
2	201832220	Ben Johnson	3
3	202003219	Charlie Jones	1
4	202045234	Dan Norris	1
5	201985603	Emily Wood	1
6	201933222	Freddie Harris	2
7	201875940	Grace Clarke	2
8	202029744	Harper Taylor	1

Screenshot of the table with a newly added row

Update Tuples/Rows

Update the student ID of “Harper Taylor” to 201929744 by using the `UPDATE` command:

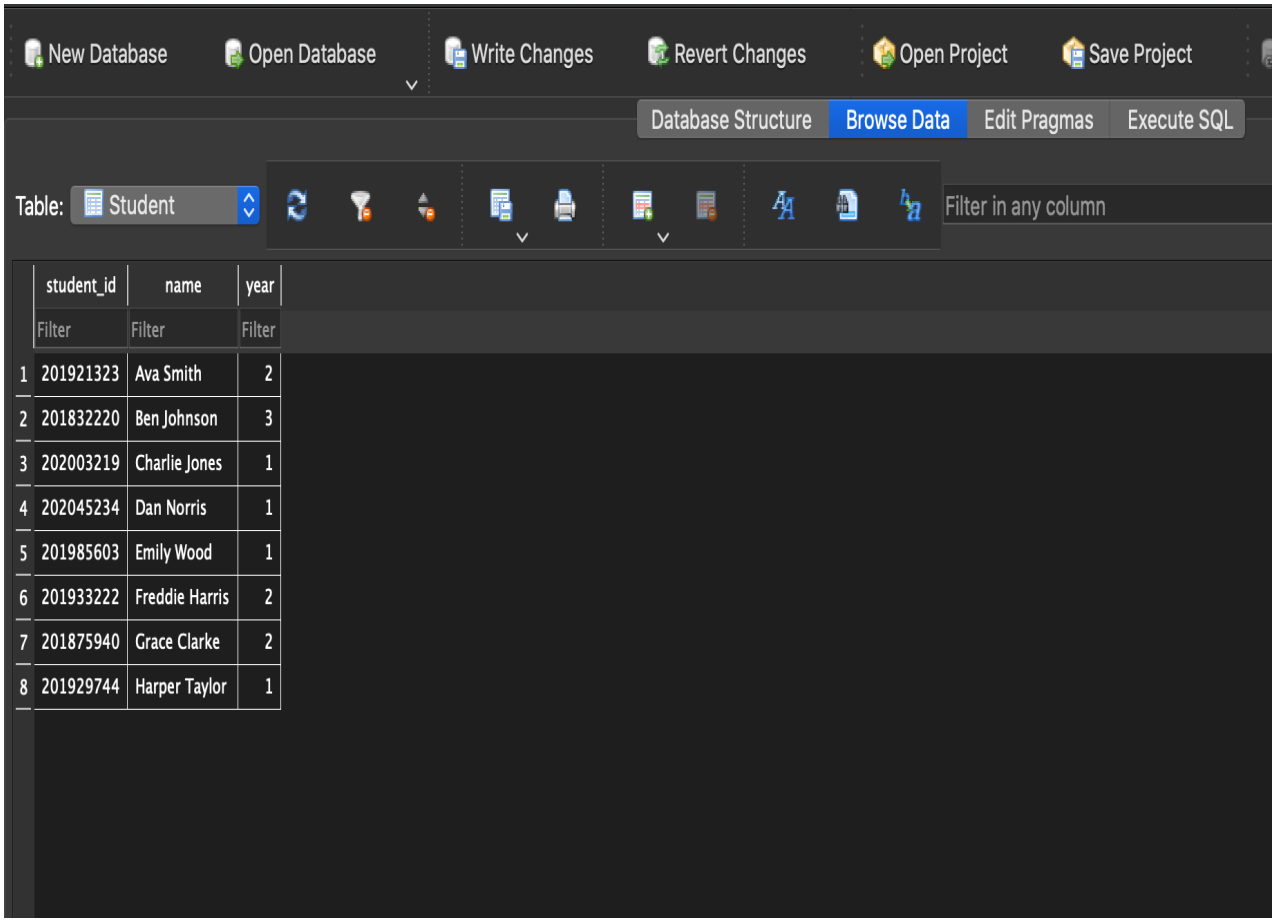
```
UPDATE Student
SET student_id = "201929744"
WHERE name = "Harper Taylor"
```



Screenshot on how to update a row

Click `Write Changes`.

Browse the table `Student` and make sure the new student information is in the table.



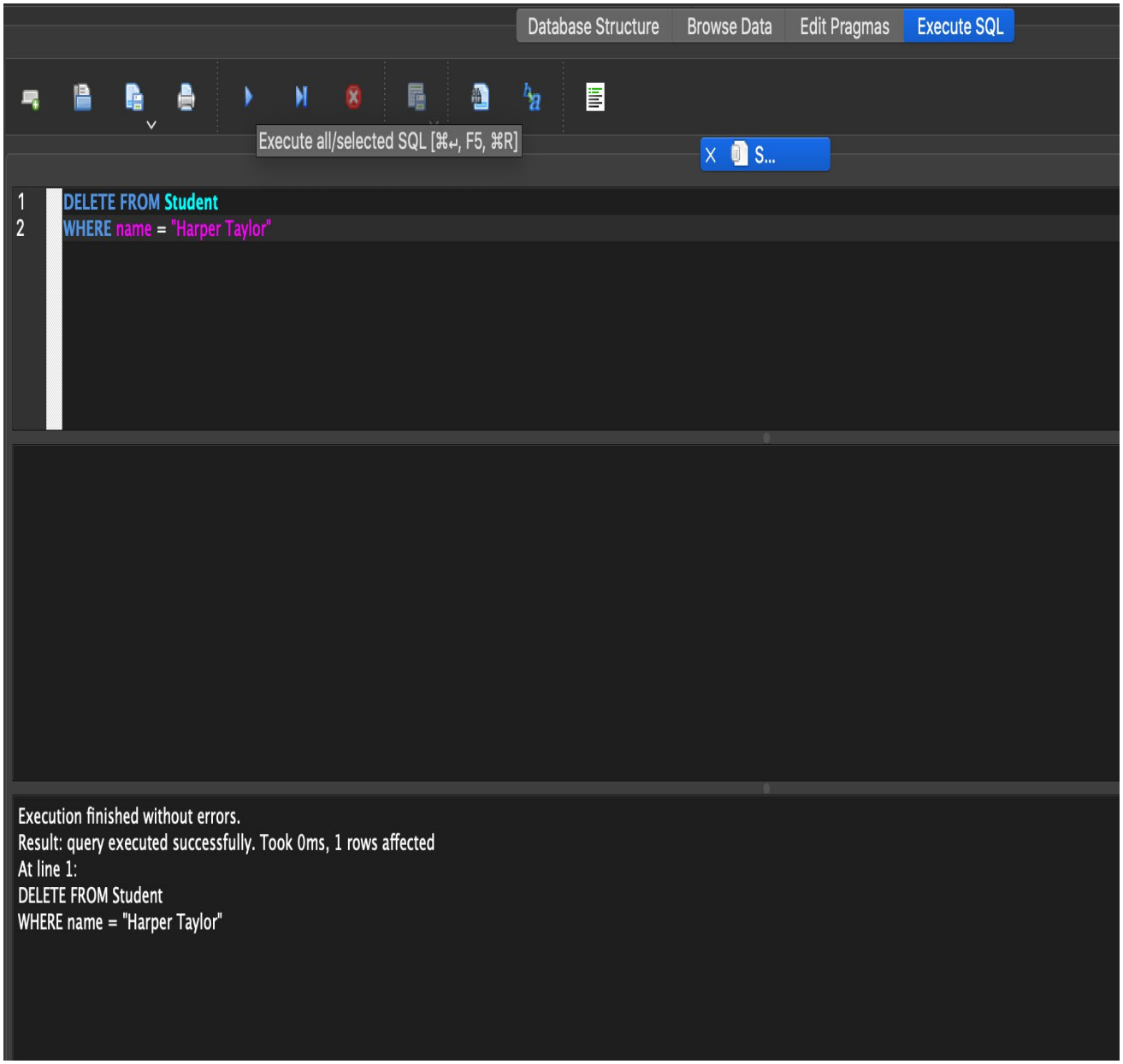
Screenshot of the table with an updated row

Delete Tuples/Rows

Now, delete the record for the student “Harper Taylor” from table `Student`:

```
DELETE FROM Student
WHERE name = "Harper Taylor"
```

Click `Write Changes`



Screenshot on how to delete a row

Browse the table `Student` and make sure that there is no row for “Harper Taylor.”

SQL Queries

SQL queries are used to select data from a database. The basic SQL query has the form:

```
SELECT A_1,A_2,…,A_n
FROM R_1,R_2, …,R_m
WHERE conditions
```

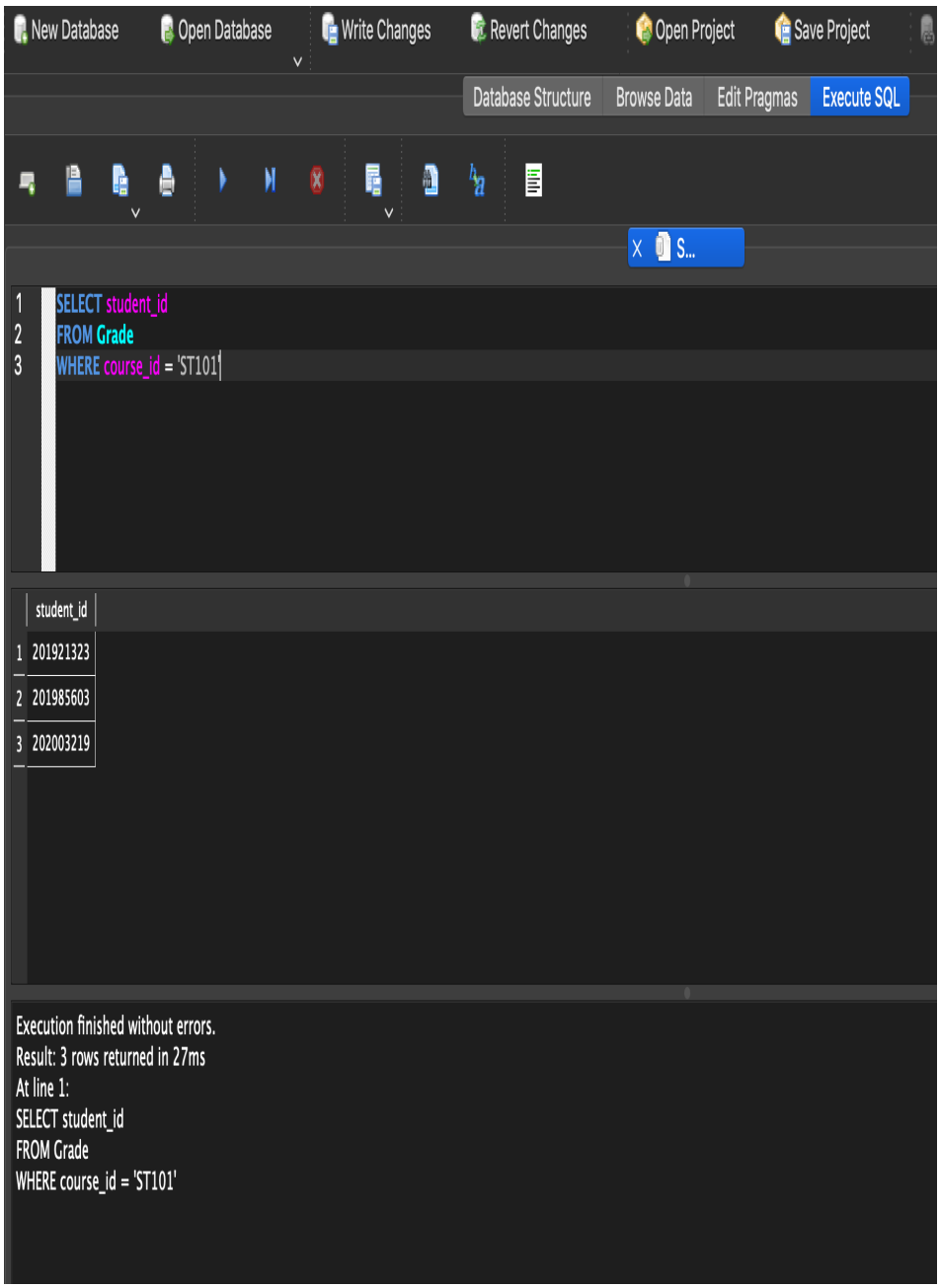
- SELECT line: Specify the attribute(s) to retain in the result.
- FROM line: Specify the table(s) to query from.
- WHERE line: Determine which tuple(s) to select.

The SQL keywords `SELECT`, `FROM`, `WHERE` are case-insensitive, however the common convention is to write them in capital letters.

Example 1: Conditions

We would like to return the student ID of all students who are in ST101:

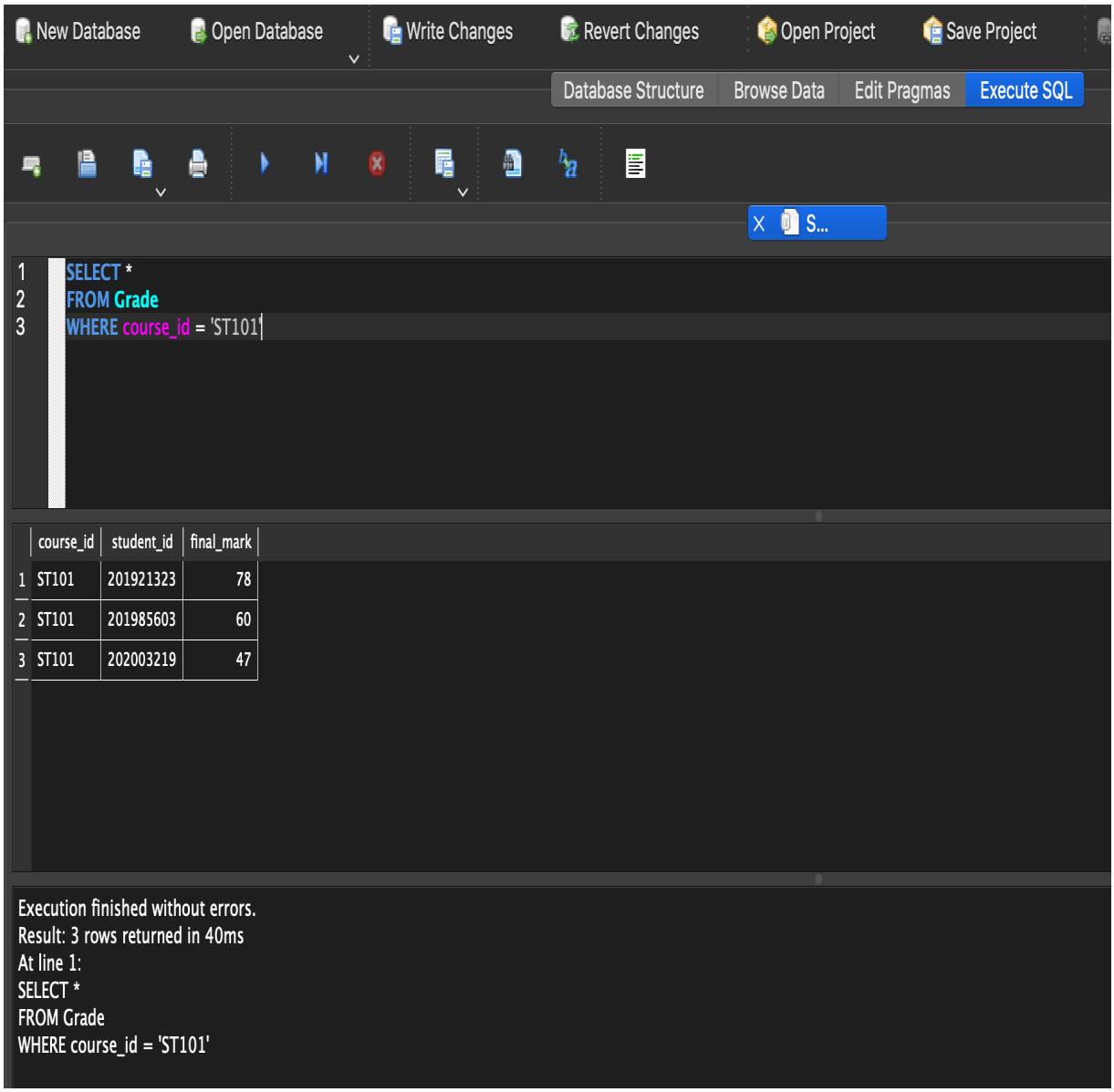
```
SELECT student_id
FROM Grade
WHERE course_id = 'ST101'
```



Screenshot of the query result

If we want to show all attributes (`course_id`, `student_id`, `final_mark`), then we can use the symbol `*`, which means that all attributes should be selected:

```
SELECT *
FROM Grade
WHERE course_id = 'ST101'
```



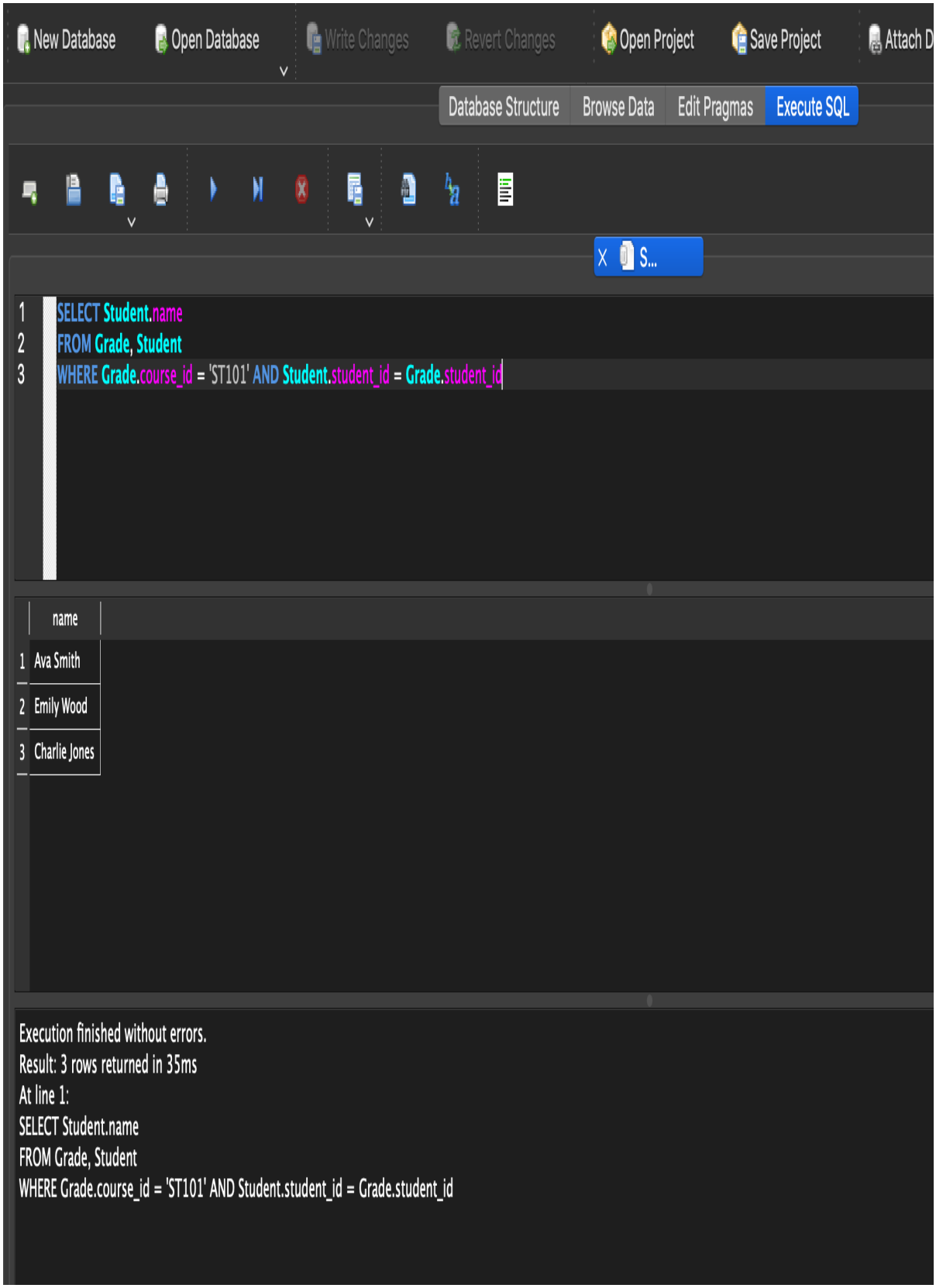
Screenshot of the query result

Example 2: Several Tables

Suppose we want to get the names of the students who took the course with course ID `ST101`. Note the student name information is in the `Student` table whereas the information about which course the student took is in `Grade`. In order to perform the query we need to combine information from the `Student` and `Grade` tables.

```
SELECT Student.name
```

```
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
```

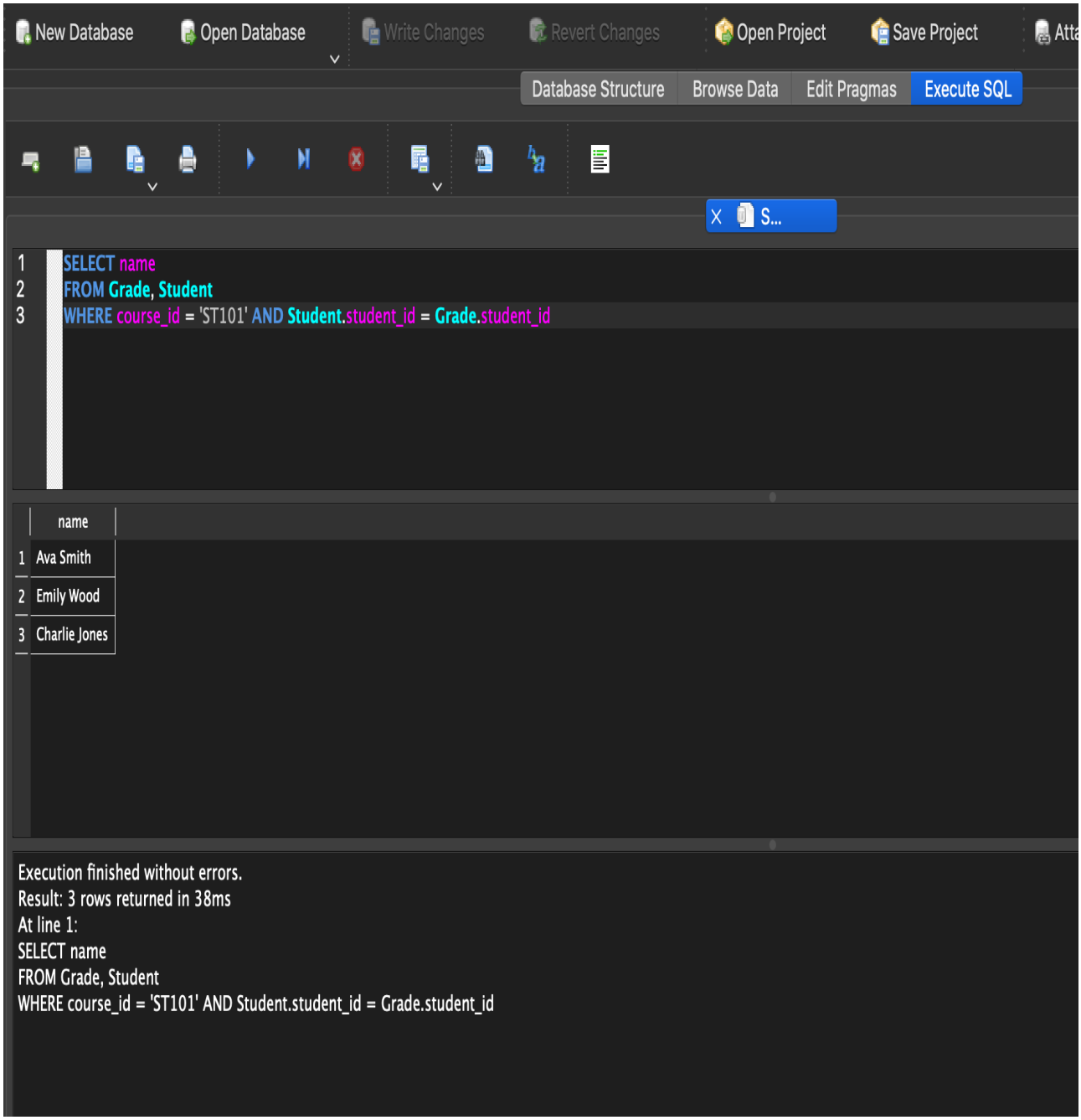


Screenshot of the query result

Note:

- On the `WHERE` line we use `AND` to specify that both conditions `Grade.course_id = 'ST101'` and `Student.student_id = Grade.student_id` need to be satisfied.
- In this query we use `Student.student_id` and `Grade.student_id` instead of `student_id` to specify which table the attribute is from. This is because the attribute `student_id` exists in both `Student` and `Grade`, and we need to indicate which table the attribute is from to eliminate ambiguity.
- We have `Student.name` instead of `name` and `Grade.course_id` instead of `course_id` to specify which table the attribute is from. However, this is not necessary here; it is perfectly fine to instead write:

```
SELECT name
FROM Grade, Student
WHERE course_id = 'ST101' AND Student.student_id = Grade.student_id
```

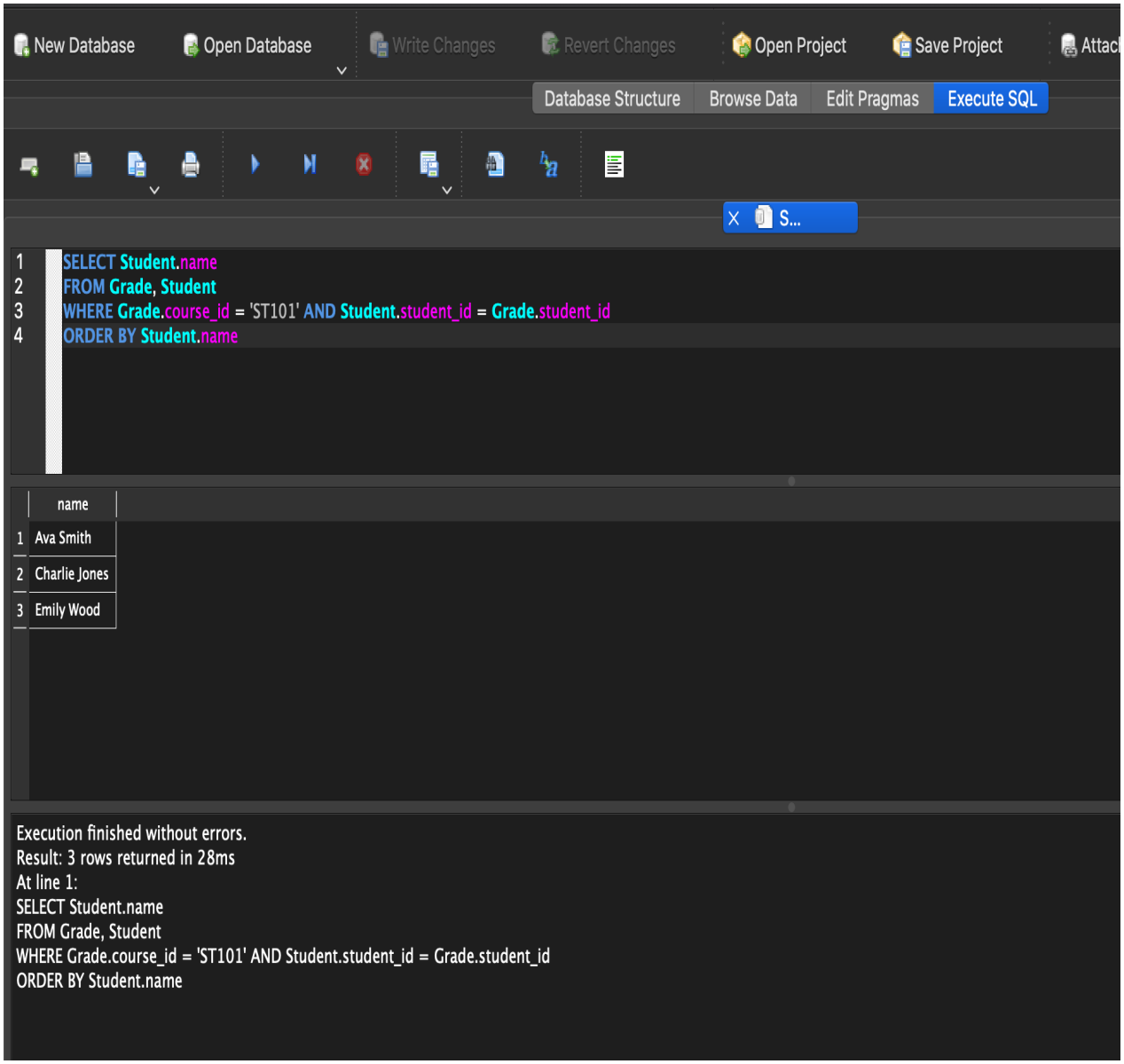


Screenshot of the query result

The reason we can do this is because the attribute `name` is only in the table `Student` and the attribute `course_id` is only in the table `Grade`.

If we want to order the result, we can use the `ORDER BY` clause. For example, if we want to order the names of the students in alphabetical order:

```
SELECT Student.name
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name
```

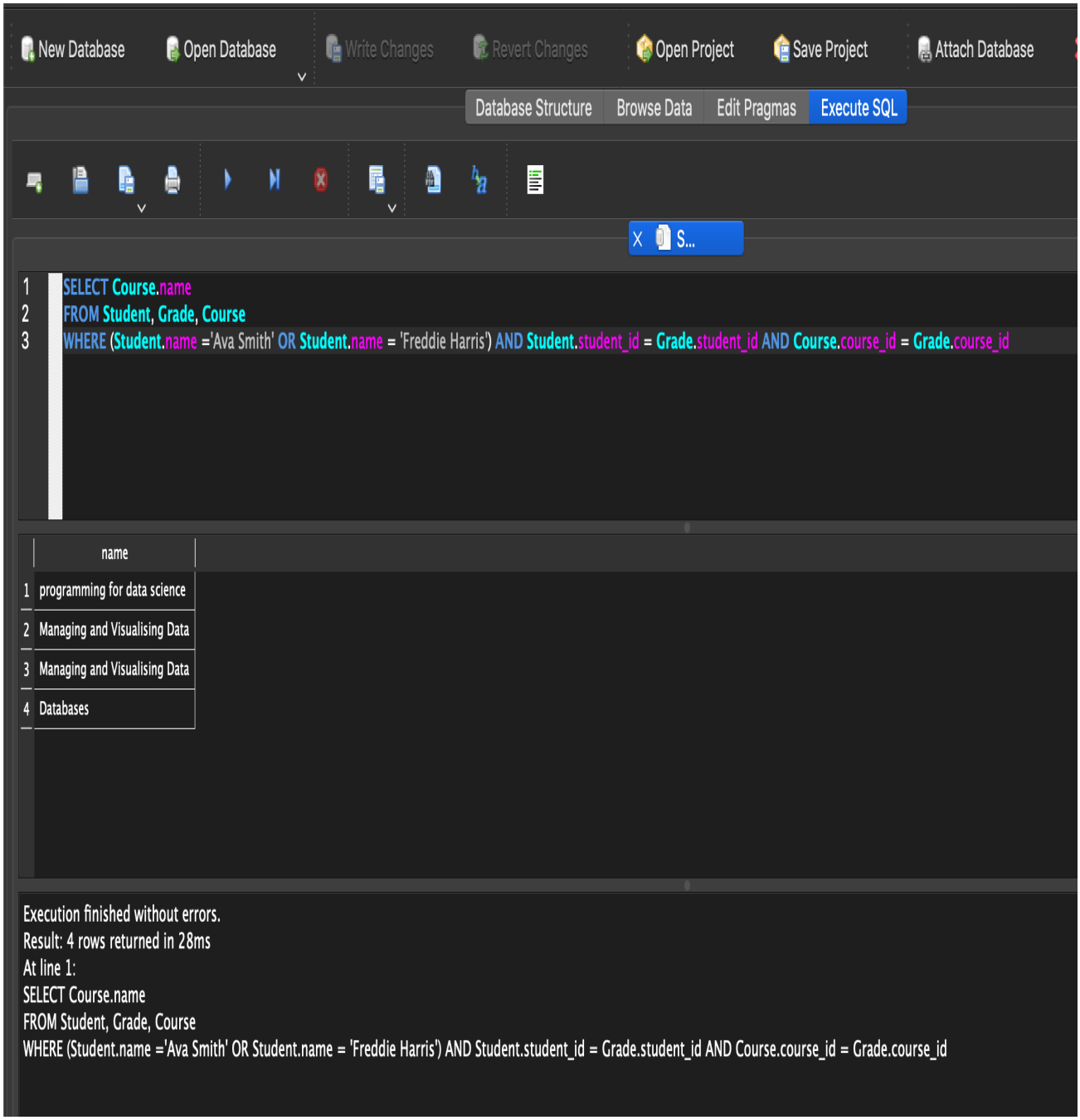


Screenshot of the query result

Example 3: Multiple Conditions

We would like to get the name of the courses taken by the student Ava Smith or Freddie Harris:

```
SELECT Course.name
FROM Student, Grade, Course
WHERE (Student.name ='Ava Smith' OR Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id =
Grade.course_id
```



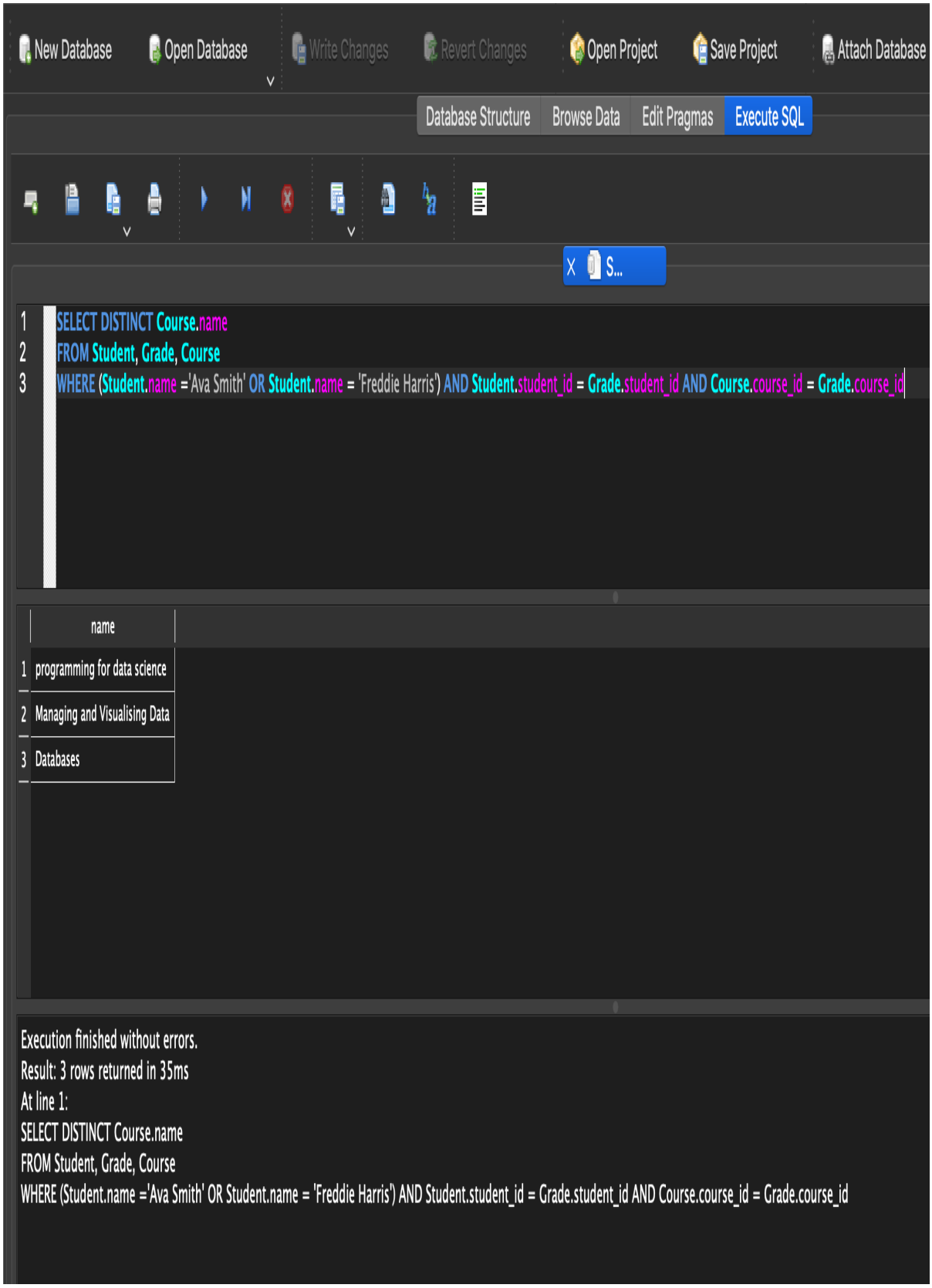
Screenshot of the query result

Note:

- `OR` is used because only one of the conditions `Student.name ='Ava Smith'` and `Student.name = 'Freddie Harris'` is needed for `department`.
- If you look at the output closely, there are a few duplicate rows, which we can remove using `DISTINCT`:

```
SELECT DISTINCT Course.name
FROM Student, Grade, Course
```

```
WHERE (Student.name ='Ava Smith' OR Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id =
Grade.course_id
```

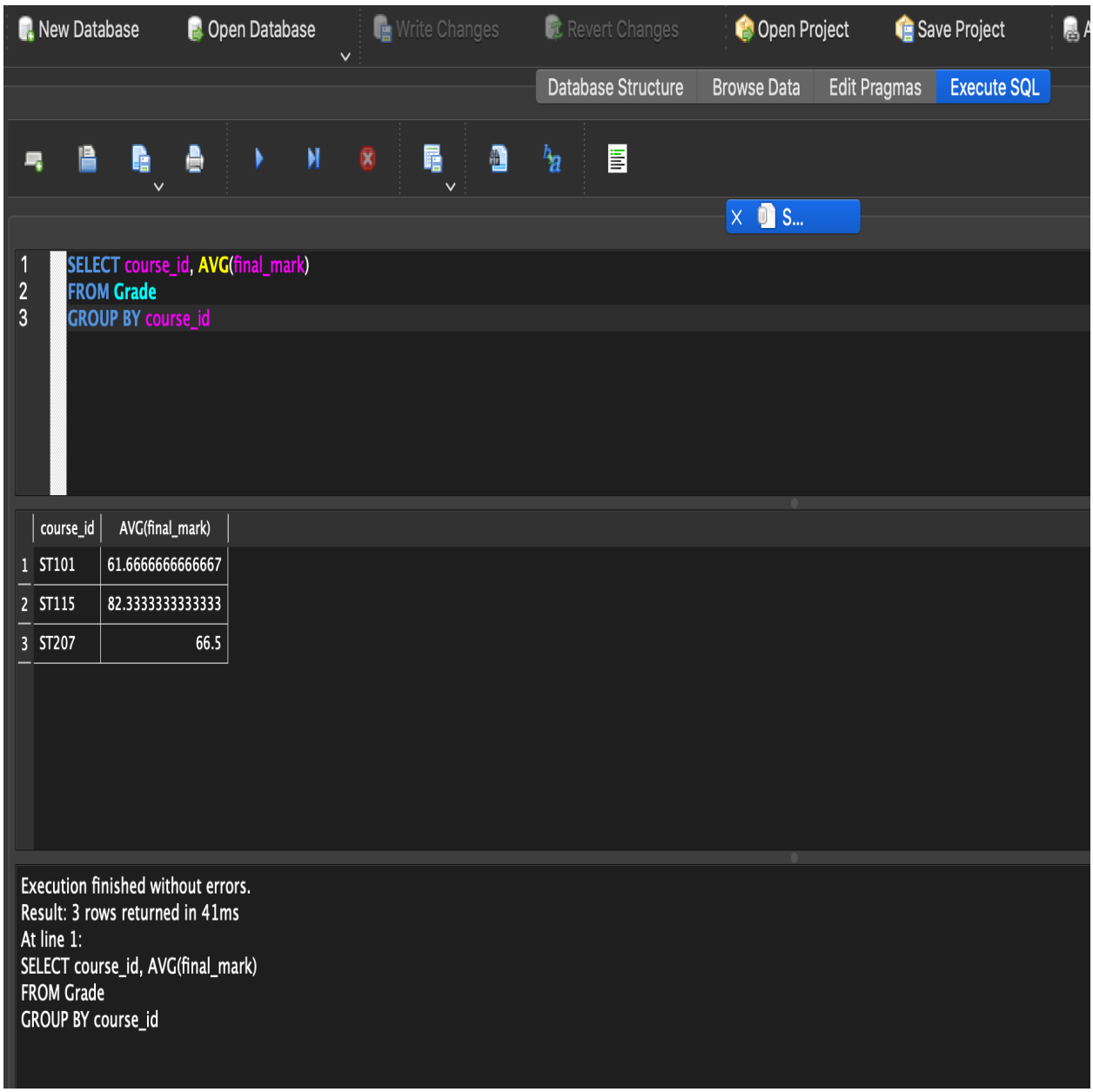


Screenshot of the query result

Example 4: Aggregation

We would like to calculate the average mark for each course according to the value of `course_id`:

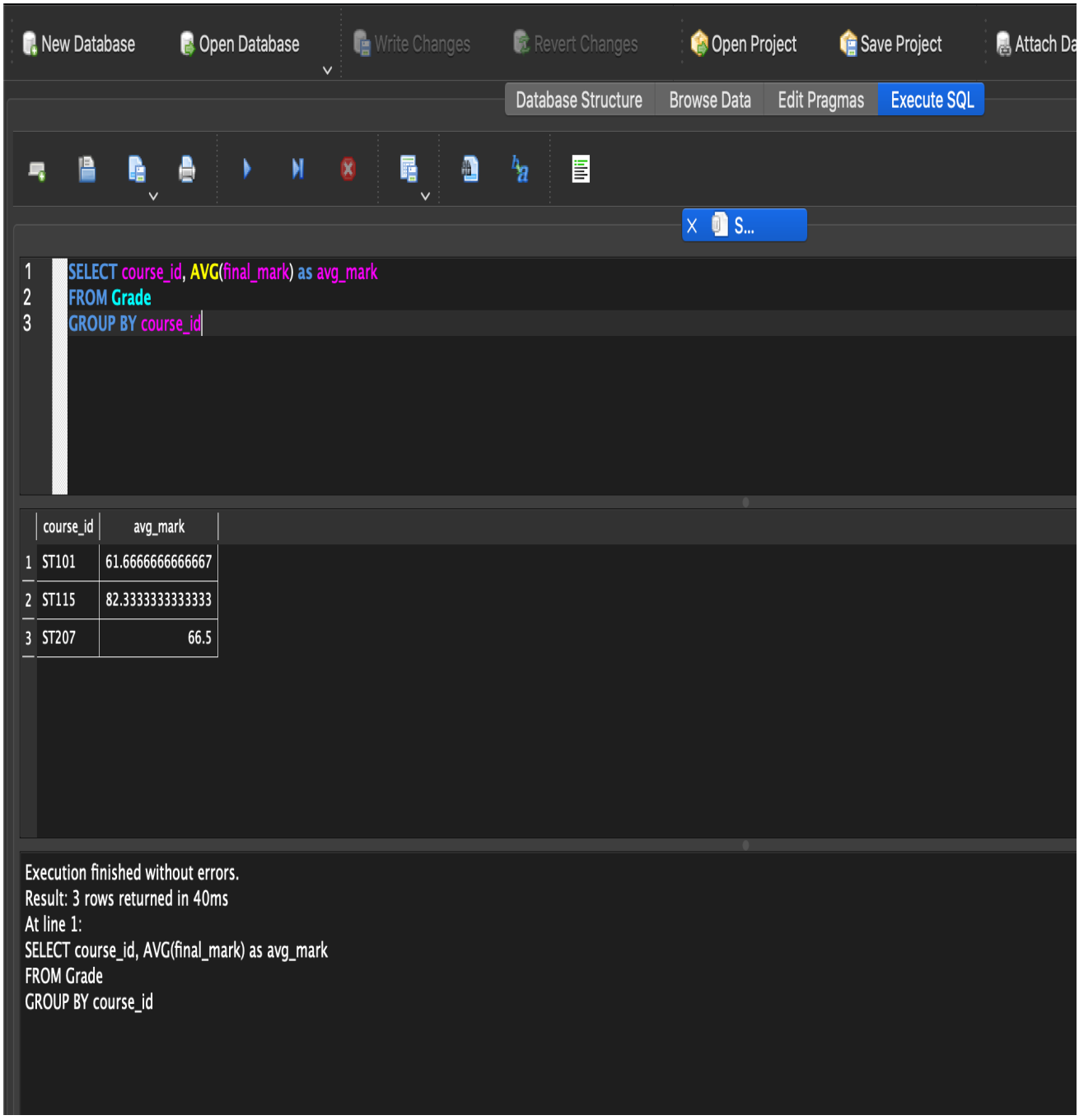
```
SELECT course_id, AVG(final_mark)
FROM Grade
GROUP BY course_id
```



Screenshot of the query result

The attribute name for the average mark looks different from other attributes. We can rename it using the `AS` clause:

```
SELECT course_id, AVG(final_mark) as avg_mark
FROM Grade
GROUP BY course_id
```



Screenshot of the query result

`GROUP BY` groups rows that have the same attribute value. It is often used to group rows before applying aggregation functions to an attribute of the group. Common aggregation functions are:

- `AVG`
- `COUNT`
- `SUM`

- AVG
- MAX
- MIN

In our example, we group the rows based on the course_id, and summarize the grouped rows by the average final_mark for each group.

SQL Joins

In the previous examples, we have used conditions like Student.student_id = Grade.student_id in the WHERE to make use of multiple tables for query. The same queries can be written more concisely using JOIN clauses.

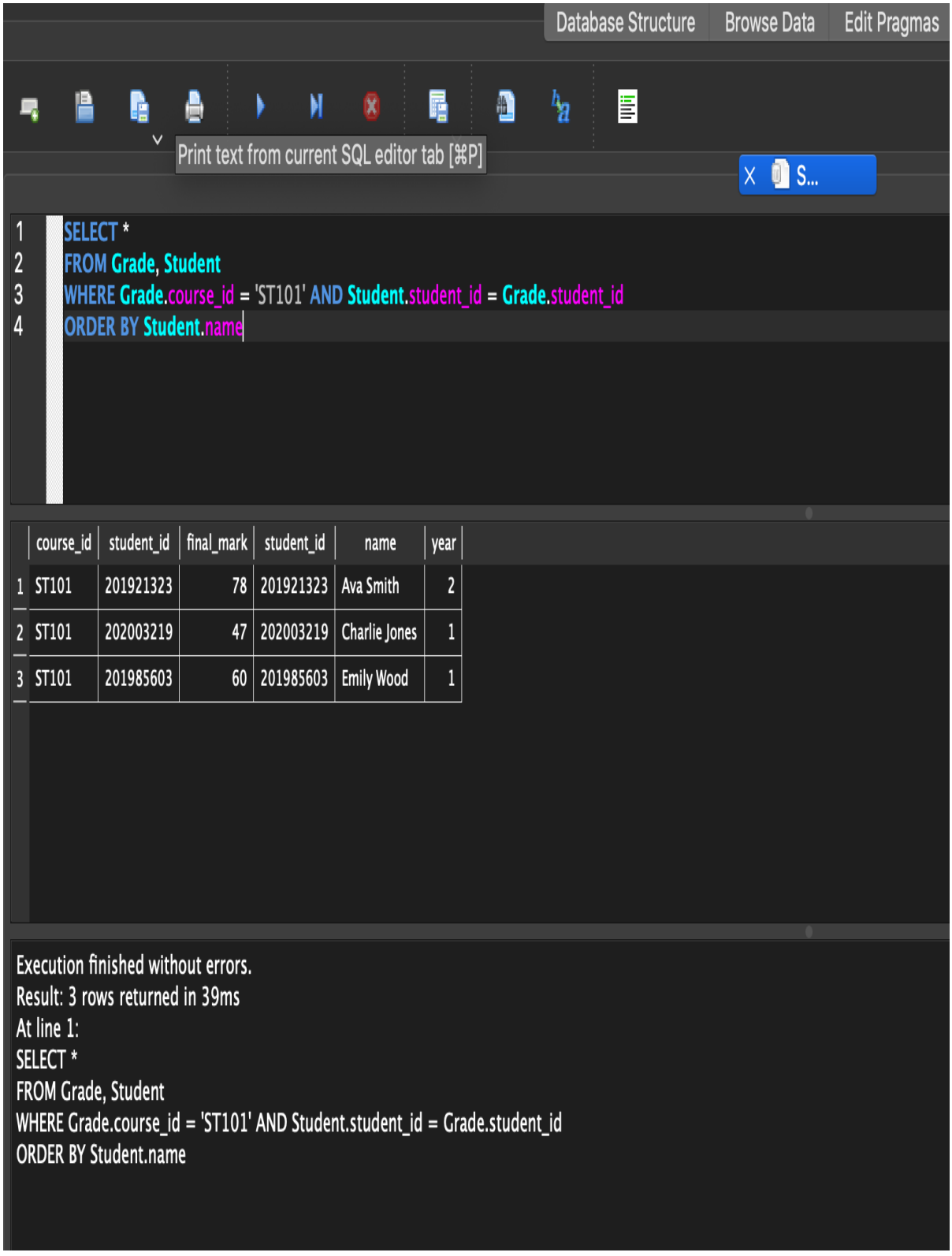
A JOIN clause combines rows from two or more tables based on related column(s) between them. The SQL language offers many different types of joins. Note that not all types of JOIN are implemented by the database engines.

- INNER JOIN: Select rows that have matching values in *both* tables based on the given columns
- NATURAL JOIN: Similar to INNER JOIN except that there is no need to specify which columns are used for matching values
- OUTER JOIN: Unlike INNER JOIN, unmatched rows in one or both tables can be returned. There are LEFT, RIGHT and FULL OUTER JOIN. SQLite only supports LEFT OUTER JOIN. For LEFT OUTER JOIN, all the records from the left table are included in the result.
- CROSS JOIN: Return the Cartesian product of the two joined tables, by matching all the values from the left table with all the values from the right table.

INNER JOIN

As in Example 2, below we get the records about students who took the course with course ID ST101, but we also sort the student names in alphabetical order:

```
SELECT *
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name
```

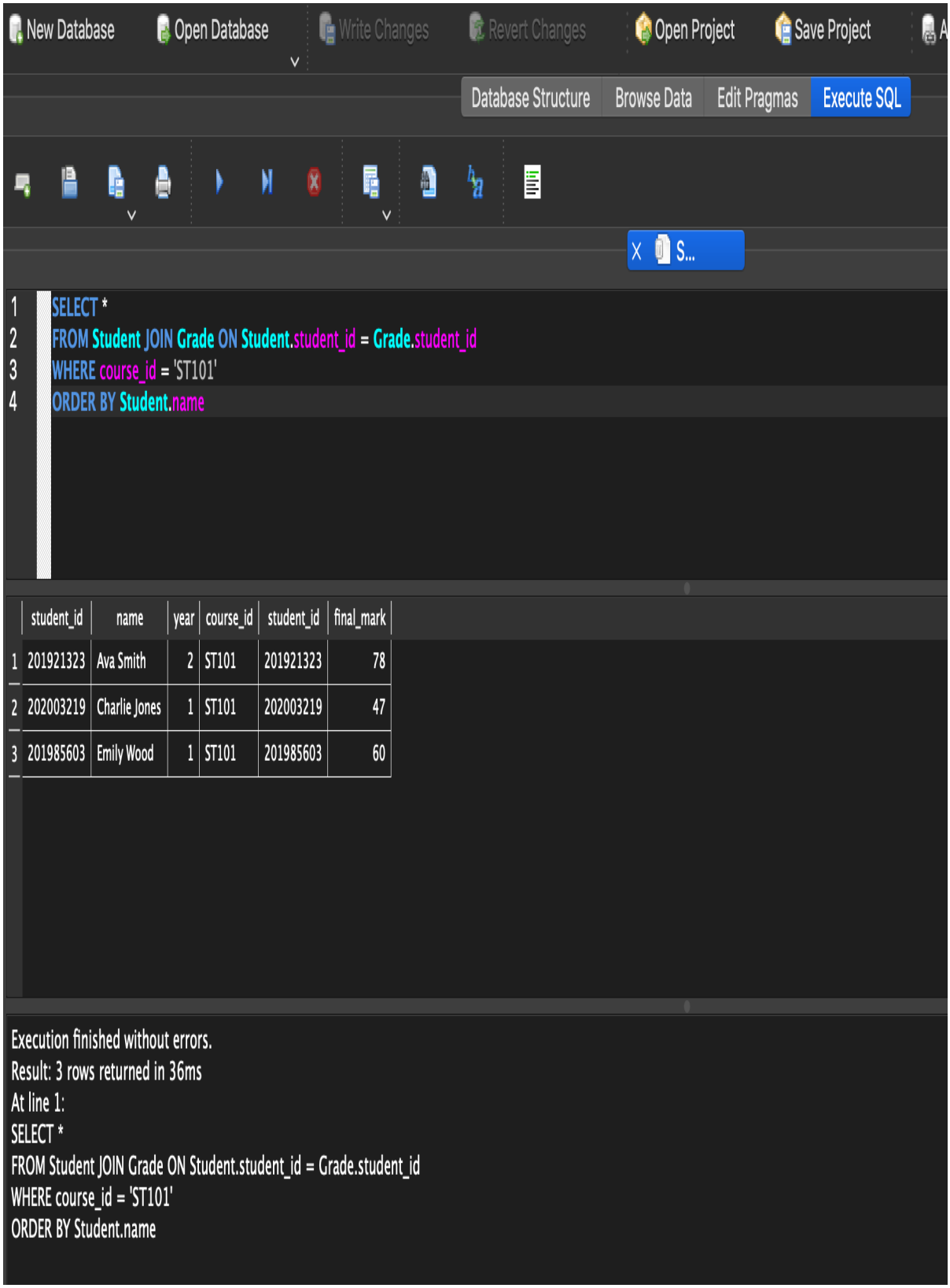


Screenshot of the query result

We can rewrite the above query using `INNER JOIN`:

```
SELECT *
FROM Student JOIN Grade ON Student.student_id = Grade.student_id
WHERE course_id = 'ST101'
```

```
ORDER BY Student.name
```



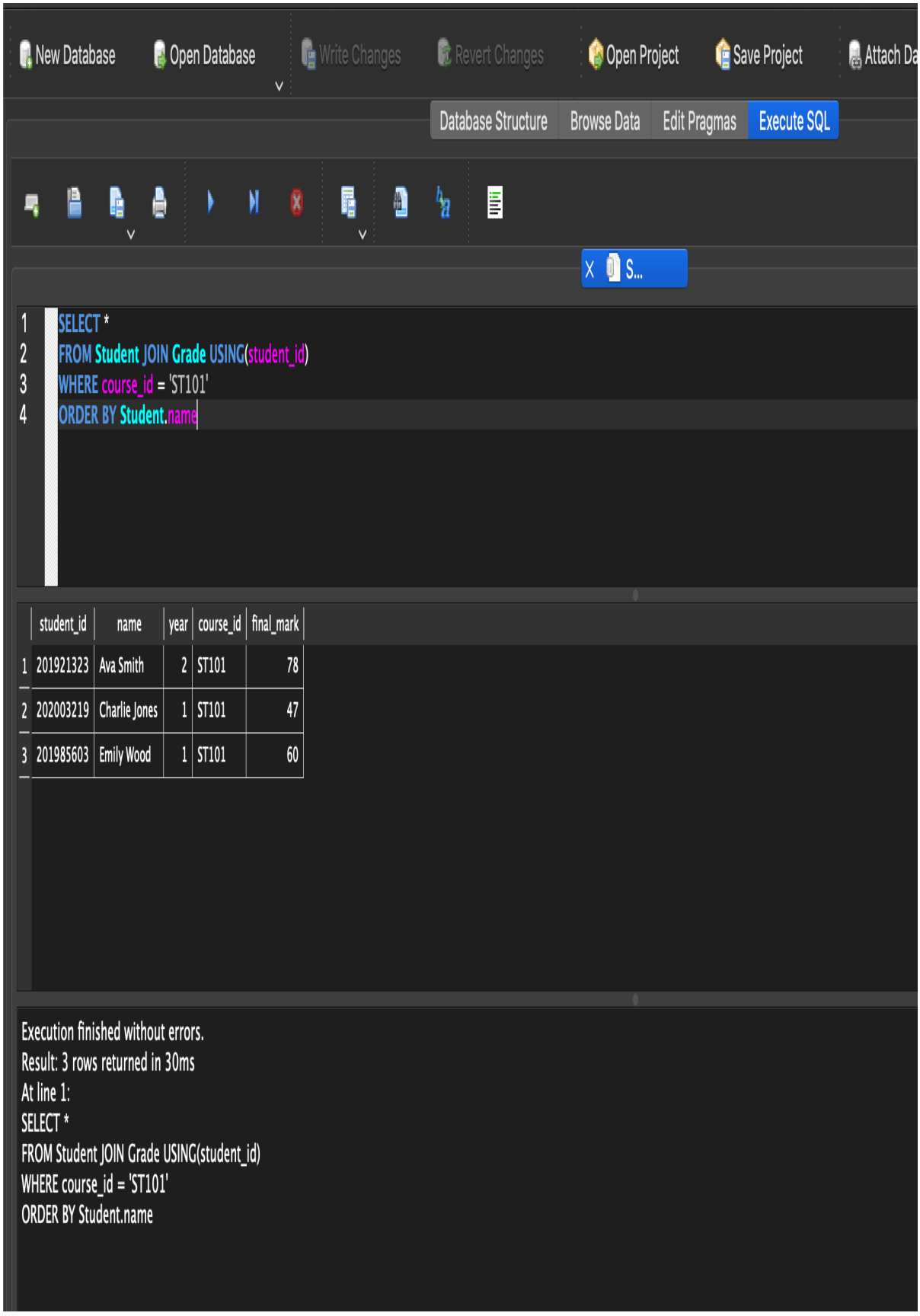
Screenshot of the query result

Note that

- We write `JOIN` instead of `INNER JOIN` as by default `INNER JOIN` is used when you do not specify the join type.
- The `ON` keyword specifies on what condition you want to join the tables.
- If you look at the result, `student_id` appears twice—this is because *all* the columns from both tables are returned.
- By using the `JOIN` clause, we separate the logic of combining the tables (`Student.student_id = Grade.student_id`) and the other condition (`course_id = 'ST101'`), which makes the SQL query more readable.

Instead of joining using `ON`, we can use `USING` with `JOIN` if the columns that we are joining have the same name. For example:

```
SELECT *
FROM Student JOIN Grade USING(student_id)
WHERE course_id = 'ST101'
ORDER BY Student.name
```



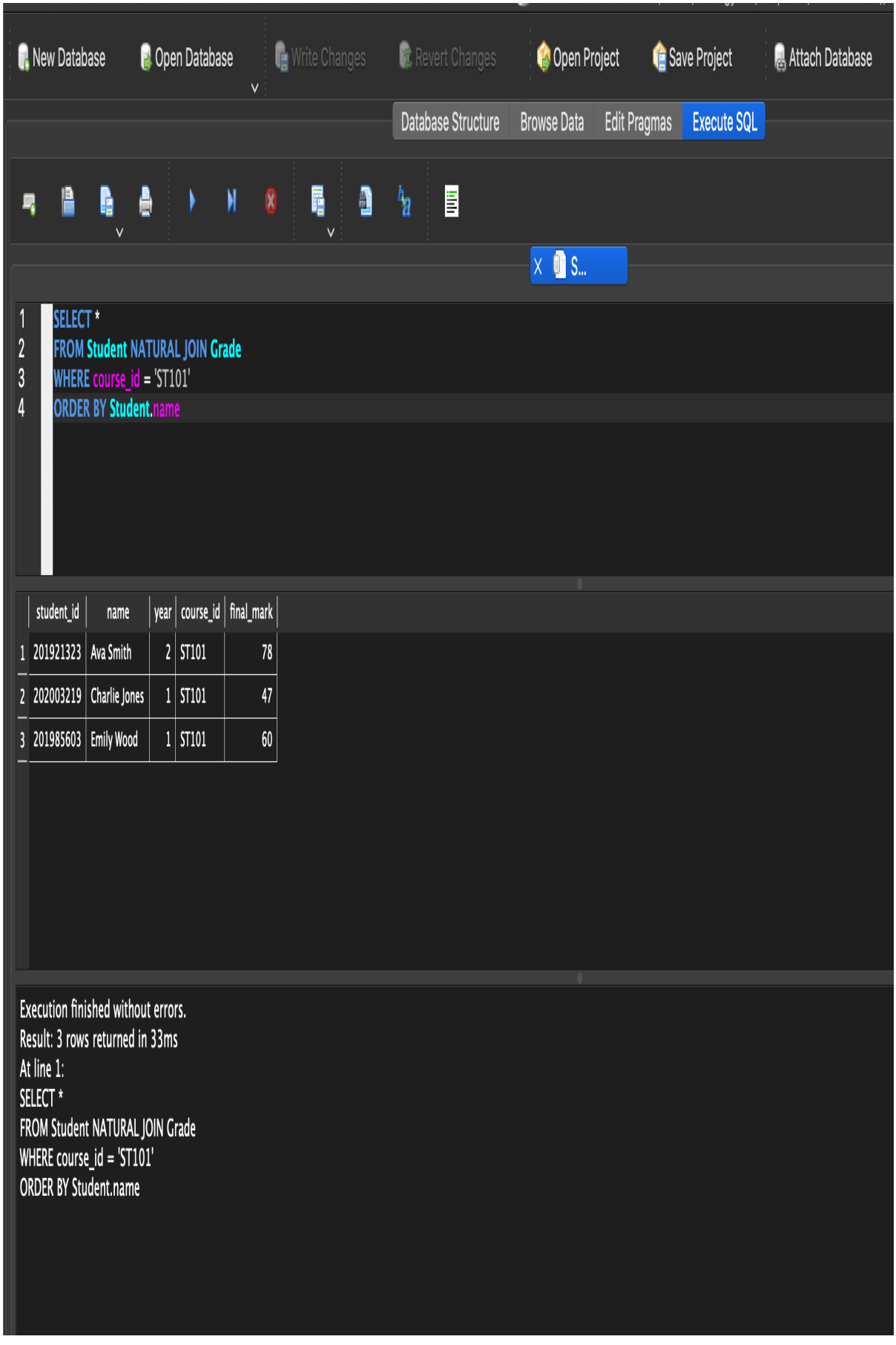
Screenshot of the query result

Note that

- The `USING` keyword specifies which column is used to select rows that have matching values in both tables.
- If you look at the result, `student_id` appears only once now.

NATURAL JOIN

```
SELECT *
FROM Student NATURAL JOIN Grade
WHERE course_id = 'ST101'
ORDER BY Student.name
```


New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
X S...
1 SELECT *
2 FROM Student NATURAL JOIN Grade
3 WHERE course_id = 'ST101'
4 ORDER BY Student.name
student_id | name | year | course_id | final_mark
1 201921323 | Ava Smith | 2 | ST101 | 78
2 202003219 | Charlie Jones | 1 | ST101 | 47
3 201985603 | Emily Wood | 1 | ST101 | 60
Execution finished without errors.
Result: 3 rows returned in 33ms
At line 1:
SELECT *
FROM Student NATURAL JOIN Grade
WHERE course_id = 'ST101'
ORDER BY Student.name

Screenshot of the query result

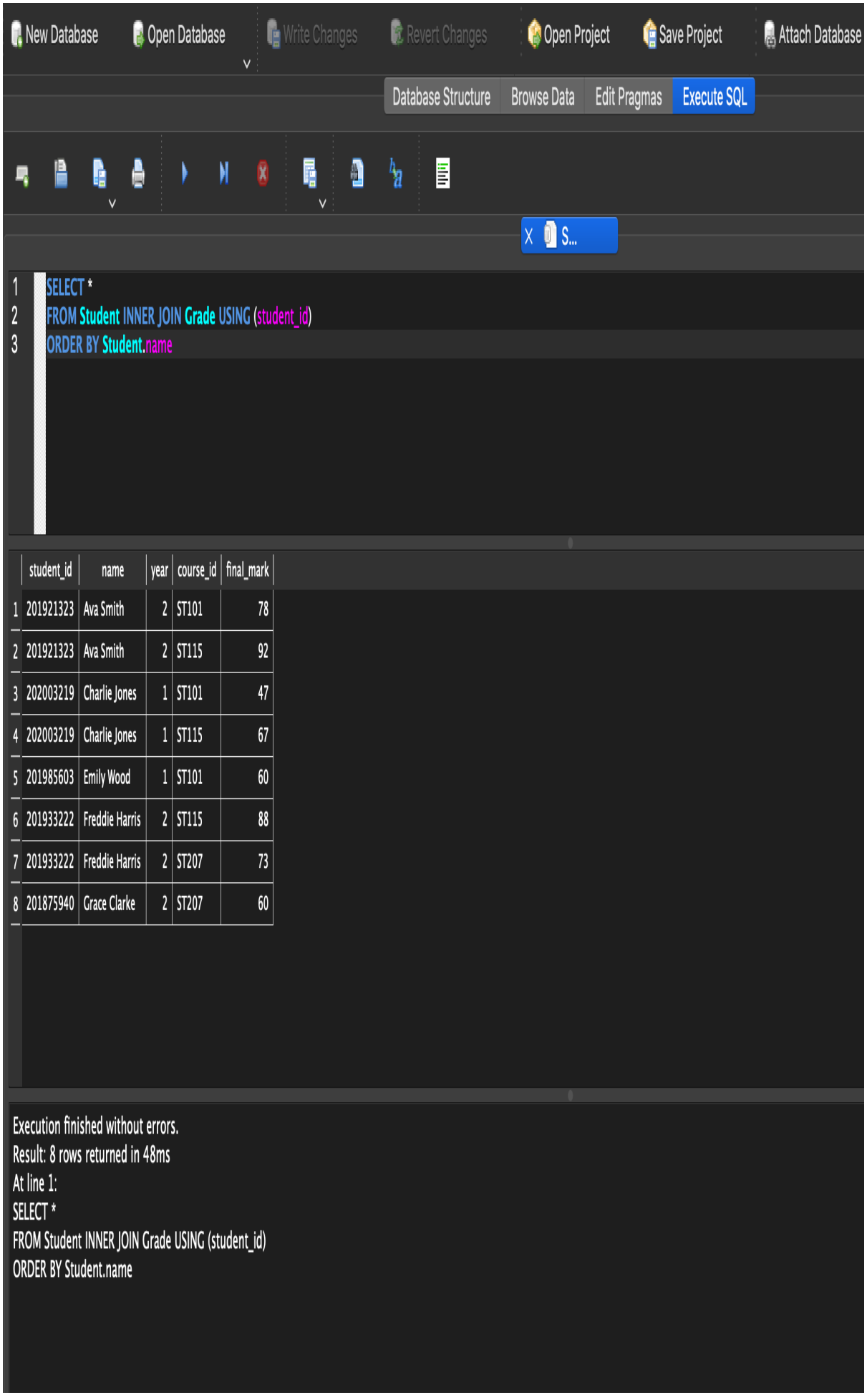
Note that:

- We do not specify how to join the two tables. The join condition is automatically identified.
- If you look at the result, `student_id` appears only once.

`LEFT JOIN`

When we run the following SQL commands to use `INNER JOIN` to combine the tables `Student` and `Grade`

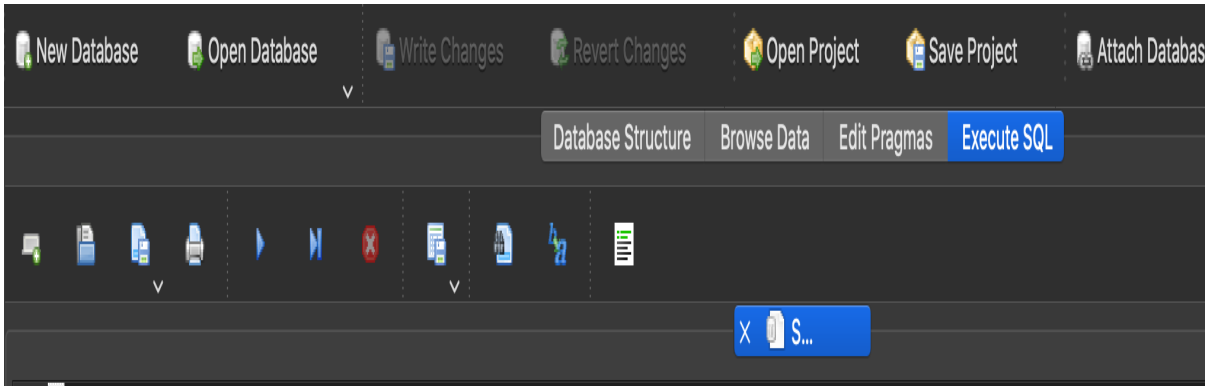
```
SELECT *
FROM Student INNER JOIN Grade USING (student_id)
ORDER BY Student.name
```


New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
X S...
1 SELECT *
2 FROM Student INNER JOIN Grade USING (student_id)
3 ORDER BY Student.name
student_id name year course_id final_mark
1 201921323 Ava Smith 2 ST101 78
2 201921323 Ava Smith 2 ST115 92
3 202003219 Charlie Jones 1 ST101 47
4 202003219 Charlie Jones 1 ST115 67
5 201985603 Emily Wood 1 ST101 60
6 201933222 Freddie Harris 2 ST115 88
7 201933222 Freddie Harris 2 ST207 73
8 201875940 Grace Clarke 2 ST207 60
Execution finished without errors.
Result: 8 rows returned in 48ms
At line 1:
SELECT *
FROM Student INNER JOIN Grade USING (student_id)
ORDER BY Student.name

Screenshot of the query result

the record of students `Ben Johnson` and `Dan Norris` are not shown, because there are not corresponding records for these two students in the table `Grade`. If we instead use `LEFT OUTER JOIN` to combine the tables `Student` and `Grade`:

```
SELECT *
FROM Student LEFT JOIN Grade USING (student_id)
ORDER BY Student.name
```




```
1  SELECT *
2  FROM Student LEFT JOIN Grade USING (student_id)
3  ORDER BY Student.name
```

	student_id	name	year	course_id	final_mark
1	201921323	Ava Smith	2	ST101	78
2	201921323	Ava Smith	2	ST115	92
3	201832220	Ben Johnson	3	*NULL*	*NULL*
4	202003219	Charlie Jones	1	ST101	47
5	202003219	Charlie Jones	1	ST115	67
6	202045234	Dan Norris	1	*NULL*	*NULL*
7	201985603	Emily Wood	1	ST101	60
8	201933222	Freddie Harris	2	ST115	88
9	201933222	Freddie Harris	2	ST207	73
10	201875940	Grace Clarke	2	ST207	60

```
Execution finished without errors.
Result: 10 rows returned in 31ms
At line 1:
SELECT *
FROM Student LEFT JOIN Grade USING (student_id)
ORDER BY Student.name
```

Screenshot of the query result

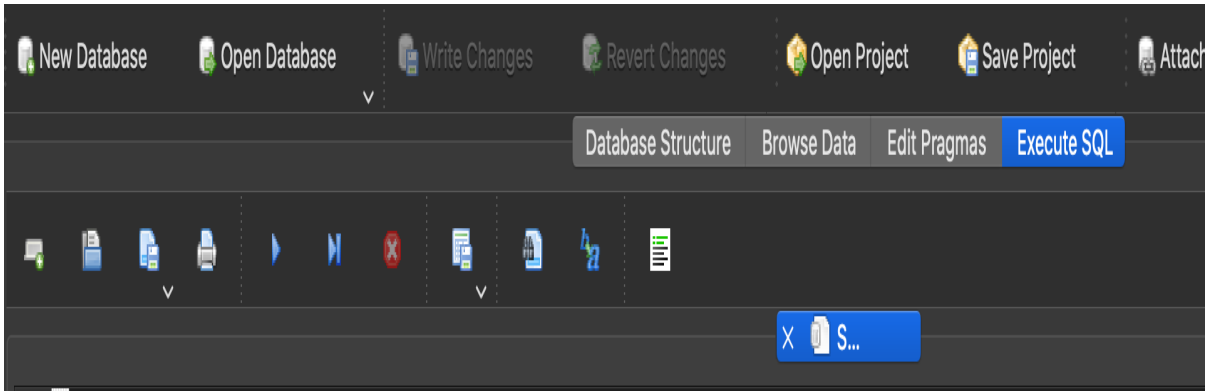
we get a result where:

- All the students from the left table `Student` are included (including `Ben Johnson` and `Dan Norris`)
- The students with no corresponding record in the `right table Grade`, have `NULL` value in attributes `course_id` and `final_mark` from `Grade`.

`CROSS JOIN`

When we use `CROSS JOIN`:

```
SELECT *
FROM Student CROSS JOIN Grade
ORDER BY Student.name
```




```
1  SELECT *
2  FROM Student CROSS JOIN Grade
3  ORDER BY Student.name
```

	student_id	name	year	course_id	student_id	final_mark
1	201921323	Ava Smith	2	ST101	201921323	78
2	201921323	Ava Smith	2	ST101	201985603	60
3	201921323	Ava Smith	2	ST101	202003219	47
4	201921323	Ava Smith	2	ST115	201921323	92
5	201921323	Ava Smith	2	ST115	202003219	67
6	201921323	Ava Smith	2	ST115	201933222	88
7	201921323	Ava Smith	2	ST207	201933222	73
8	201921323	Ava Smith	2	ST207	201875940	60
9	201832220	Ben Johnson	3	ST101	201921323	78
10	201832220	Ben Johnson	3	ST101	201985603	60
11	201832220	Ben Johnson	3	ST101	202003219	47
12	201832220	Ben Johnson	3	ST115	201921323	92
13	201832220	Ben Johnson	3	ST115	202003219	67
14	201832220	Ben Johnson	3	ST115	201933222	88
15	201832220	Ben Johnson	3	ST207	201933222	73
16	201832220	Ben Johnson	3	ST207	201875940	60
17	202003219	Charlie Jones	1	ST101	201921323	78
18	202003219	Charlie Jones	1	ST101	201985603	60
19	202003219	Charlie Jones	1	ST101	202003219	47
20	202003219	Charlie Jones	1	ST115	201921323	92
21	202003219	Charlie Jones	1	ST115	202003219	67

Screenshot of the query result

we get 56 rows, which is number of rows in `Student` (7) times number of rows in `Grade` (8).

Useful Links and Resources

- SQLite join to learn more about how to join tables via SQLite.
- SQLite select to learn more about how to query via SQLite.
- SQL from Wikipedia.

Creating and Manipulating Databases in R Using DBI

Using Databases With R

So far, we have seen how to create, update and query a database using **DB Browser for SQLite**. Often the process does not stop there—we may want to do further analysis on the queried data, or we may want to store the data analyzed using R and Python back to the database. For data analysis, we could export the table / query result into a file (say a CSV file) and then read the file into R and Python. For storing the processed data, we could again export the data from R and Python to a file and then import the data into the database using **DB Browser for SQLite**.

Another (and probably a better) way is to do it all in R and Python—we can connect R and Python to the database to create, update and query as appropriate. We then can work on the query result directly without the need of saving and loading the query result to an external file. Similarly, we can directly store the analyzed data back to the database.

In these pages we will focus on how to interact with databases in R and Python. Here we will continue to use SQLite as the database engine, and we will use the **RSQLite** and **DBI** R packages to show that what we have been done so far on **DB Browser for SQLite** can also be done within in R. To highlight this point, we will use the same examples we used before. In this notebook, we will show how to connect and create databases and tables using R.

Connecting to Databases

We load the library **DBI** and use the function `dbConnect()` to create an object, `conn`, to connect to the SQLite driver to manipulate the database `university.db`. If the database `university.db` exists in your working directory, the following code chunk will remove it.

```
# install.packages("RSQLite")
library(DBI)

if (file.exists("university.db"))
  file.remove("university.db")
[1] TRUE
conn <- dbConnect(RSQLite::SQLite(), "university.db")
```

We can list all tables in `university.db` using the function `dbListTable()` from **DBI**.

```
# list all tables
dbListTables(conn)
character(0)
```

Nothing is returned because we have not created any tables in the database yet.

Creating Tables

Now we are going to create some tables to the database `university.db`. Like before, we will create the tables using data saved in the CSV files. We first read the CSV files into `data.frame` in R:

```
course <- read.csv("course.csv", header = TRUE)
student <- read.csv("Student.csv", header = TRUE)
grade <- read.csv("grade.csv", header = TRUE)
```

We then copy the data frames `student`, `grade` and `course` to tables in the database `university.db` using **DBI**'s `dbWriteTable()` function:

```
dbWriteTable(conn, "Course", course)
dbWriteTable(conn, "Student", student)
dbWriteTable(conn, "Grade", grade)
```

Now we can see there are three tables in the database:

```
dbListTables(conn)
[1] "Course"  "Grade"   "Student"
```

We can also browse any table in the database using the function `dbReadTable()` from **DBI**. For example, we can browse the `Student` table by:

```
dbReadTable(conn, "Student")
  student_id           name year
1  201921323      Ava Smith    2
2  201832220    Ben Johnson    3
3  202003219  Charlie Jones    1
4  202045234     Dan Norris    1
5  201985603     Emily Wood    1
6  201933222 Freddie Harris    2
7  201875940   Grace Clarke    2
```

Or we can see the attributes of `Student` by:

```
dbListFields(conn, "Student")
[1] "student_id" "name"       "year"
```

We can also check if the database has been properly created by opening the database in **DB Browser for SQLite** and browse the tables from there.

Manipulating Databases

The simplest way to manipulate databases is to use the `dbExecute()` function. This functions executes SQL statements and returns the number of rows affected. This allows us to leverage SQL commands to manipulate databases from within R. There are also some other functions in **DBI** that are specialized to perform certain tasks.

Adding a New Table

We can add a new table by using the function `dbCreateTable()`. Alternatively, you can use `dbExecute()` to run the SQL command to create a new table.

```
dbCreateTable(conn, "Teacher", c(staff_id = "TEXT", name = "TEXT"))

# Alternative:
# dbExecute(conn,
# "CREATE TABLE Teacher (
#     staff_id TEXT PRIMARY KEY,
#     name TEXT)")
```

When we list the tables, we can see four tables.

```
dbListTables(conn)
[1] "Course"  "Grade"   "Student" "Teacher"
```

The table `Teacher` has two attributes with no rows.

```
dbListFields(conn, "Teacher")
[1] "staff_id" "name"
dbReadTable(conn, "Teacher")
[1] staff_id name
<0 rows> (or 0-length row.names)
```

Deleting a Table

We can remove a table by using the function `dbRemoveTable()`:

```
dbRemoveTable(conn, "Teacher")

# Alternative:
# dbExecute(conn,
# "DROP TABLE Teacher")
```

When we list the tables, we can now see three tables.

```
dbListTables(conn)
[1] "Course"  "Grade"   "Student"
```

Inserting Tuples/Rows

Below we insert the year 1 student “Harper Taylor” with student ID 202029744 to `Student` by using the function `dbAppendTable()`:

```
dbAppendTable(conn, "Student", data.frame(student_id = "202029744",
                                          name = "Harper Taylor",
                                          year = 1))
[1] 1
# Alternative:
# dbExecute(conn,
# "INSERT INTO Student VALUES(202029744, 'Harper Taylor', 1)")
```

When we browse the table, we can see the new row has been added.

```
dbReadTable(conn, "Student")
  student_id           name year
1  201921323      Ava Smith    2
2  201832220    Ben Johnson    3
3  202003219  Charlie Jones    1
4  202045234     Dan Norris    1
5  201985603     Emily Wood    1
6  201933222 Freddie Harris    2
7  201875940   Grace Clarke    2
8  202029744  Harper Taylor    1
```

Updating Tuples/Rows

Below, we update the student ID of student Harper Taylor to 201929744 by `dbExecute()`. There is no specific function for updating a row in **DBI**.

```
dbExecute(conn,
"Update Student
SET student_id = '201929744'
WHERE name = 'Harper Taylor'")
[1] 1
```

When we browse the table, we can see the row has changed.

```
dbReadTable(conn, "Student")
  student_id           name year
1  201921323      Ava Smith    2
2  201832220    Ben Johnson    3
3  202003219  Charlie Jones    1
4  202045234     Dan Norris    1
5  201985603     Emily Wood    1
6  201933222 Freddie Harris    2
7  201875940   Grace Clarke    2
8  201929744  Harper Taylor    1
```

Deleting Tuples/Rows

Below, we delete the record for the student Harper Taylor from table `Student` using `dbExecute()`. There is no specific function for deleting a row in **DBI**.

```
dbExecute(conn,
"DELETE FROM Student
WHERE name = 'Harper Taylor'")
[1] 1
```

When we browse the table, we can see the row has been removed.

```
dbReadTable(conn, "Student")
  student_id           name year
1  201921323      Ava Smith    2
2  201832220    Ben Johnson    3
3  202003219  Charlie Jones    1
4  202045234     Dan Norris    1
5  201985603     Emily Wood    1
6  201933222 Freddie Harris    2
7  201875940   Grace Clarke    2
```

Disconnecting From the Database

After we finish manipulating the database, we can close the connection using the function `dbDisconnect()` from **DBI**:

```
dbDisconnect(conn)
```

Useful Links and Resources

- Using **DBI**: A guide on how to use **DBI** from RStudio
- **DBI** reference manual
- **RSQLite** vignettes

Querying Databases in R Using DBI

Using Databases With R

In the previous section, we have seen how to use R to connect, create and manipulate a database. Here, we will use R to query `university.db`. We first connect to the database `university.db` and list all the tables.

```
library(DBI)
conn <- dbConnect(RSQLite::SQLite(), "university.db")
dbListTables(conn)
[1] "Course"  "Grade"   "Student"
```

We should see all the three tables `Student`, `Course`, and `Grade` that we have created before.

Querying Databases Using DBI

Once we formulate the query into an SQL SELECT statement, we can get the query result in `R` using the function `dbGetQuery()` or `dbSendQuery()` from **DBI**. The following examples show how we can run queries from within `R`.

Getting Grades of a Course With `course_id "ST101"`

```
q1 <- dbGetQuery(conn,
"SELECT final_mark
FROM Grade
WHERE course_id = 'ST101'")
q1
  final_mark
1         78
2         60
3         47
```

The second argument in `dbGetQuery()` is the SQL query statement we used in previous Sections, when queries were sent using DB Browser for SQLite.

```
class(q1)
[1] "data.frame"
```

Note that the `dbGetQuery()` returns a `data.frame`.

Alternatively, we can use `dbSendQuery()` and `dbFetch()`:

```
q1 <- dbSendQuery(conn,
"SELECT final_mark
FROM Grade
WHERE course_id = 'ST101'")
q1
<SQLiteResult>
  SQL  SELECT final_mark
FROM Grade
WHERE course_id = 'ST101'
  ROWS Fetched: 0 [incomplete]
       Changed: 0
```

Note `dbSendQuery()` only sends and executes the SQL query to the database engine. It does not extract any records.

```
dbFetch(q1)
  final_mark
1         78
2         60
3         47
```

When we run `dbFetch()`, the executed query result will then be fetched.

Getting Names of Students in Alphabetical Order

```
dbGetQuery(conn,
"SELECT Student.name
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name")
Warning: Closing open result set, pending rows
           name
1     Ava Smith
2 Charlie Jones
3    Emily Wood
```

Or we can do it with `NATURAL JOIN`:

```
dbGetQuery(conn,
"SELECT Student.name
FROM Student NATURAL JOIN Grade
WHERE course_id = 'ST101'
ORDER BY Student.name")
           name
1     Ava Smith
2 Charlie Jones
3    Emily Wood
```

Getting Courses Taken by Students Ava Smith or Freddie Harris

```
dbGetQuery(conn,
"SELECT DISTINCT Course.name
FROM Student, Grade, Course
```

```
WHERE (Student.name ='Ava Smith' or Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id =
Grade.course_id")
                            name
1  programming for data science
2 Managing and Visualising Data
3                      Databases
```

Or we can do it using `JOIN`:

```
dbGetQuery(conn,
"SELECT DISTINCT Course.name
FROM (Student NATURAL JOIN Grade) S JOIN Course on Course.course_id =
S.course_id
WHERE S.name ='Ava Smith' or S.name ='Freddie Harris'")
                            name
1  programming for data science
2 Managing and Visualising Data
3                      Databases
```

Calculating Average Mark for Each Course

```
dbGetQuery(conn,
"SELECT course_id, AVG(final_mark) as avg_mark
FROM Grade
GROUP BY course_id")
  course_id avg_mark
1     ST101 61.66667
2     ST115 82.33333
3     ST207 66.50000
```

After we finish manipulating the database, we can close the connection using the function `dbDisconnect()`:

```
dbDisconnect(conn)
```

Useful Links and Resources

- Using **DBI**: A guide on how to use **DBI** from RStudio
- **DBI** reference manual
- **RSQLite** vignettes

Querying Databases in R Using dplyr

Introduction to dplyr

dplyr is an R package for data manipulation. It provides a set of functions, named as verbs, that can be used to carry out a wide range of data manipulation operations:

- The `mutate()` function adds new variables that are transformations of existing variables.
- The `select()` function picks variables based on their names.
- The `filter()` function picks cases based on their values.
- The `summarize()` function reduces multiple values down to a single summary.
- The `arrange()` function changes the ordering of the rows.

dplyr verbs operate on data frames, but they also, almost seamlessly apply to database tables! In particular **dplyr** allows you to use database tables as if they are data frames, by internally converting **dplyr** code into SQL commands using **dbplyr**. The following table compares the syntax used in SQL with **dplyr** syntax:

action	SQL	dplyr
select a column	SELECT	select
select a row	WHERE	filter
sort	ORDER BY	arrange
group	GROUP BY	group_by
aggregation	aggregation functions (e.g. `AVG()`) in SELECT	summarize

See the SQL Translation article in **dbplyr**'s pages for more information.

Pipe Operator

All of the **dplyr** functions take a data frame (or a `tibble`) as the first argument. In this way, the `%>%` operator from the **magrittr** R package can be used, so that user does not need to save intermediate objects or nest verbs. For example, the statement `x %>% f(y)` is equivalent to `f(x, y)`, and the result from one step is "piped" into the next step. You can think of the pipe operator as "then."

Querying Databases Using dplyr

Connecting to the Database

Again, we first connect to the database `university.db` and list all the tables. We should see all the three tables `student`, `course` and `grade` that we have created before.

```
library(DBI)
conn <- dbConnect(RSQLite::SQLite(), "university.db")
dbListTables(conn)
[1] "Course"  "Grade"   "Student"
```

Creating a Reference to Table

```
library(dplyr)
student_db <- tbl(conn, "Student")
grade_db <- tbl(conn, "Grade")
course_db <- tbl(conn, "Course")
```

By creating references to the tables as done above, we can treat `student_db`, `grade_db`, and `course_db` as data frames, and use **dplyr** functionality to query the database.

Getting Grades of the Course With `course_id "ST101"`

```
q1 <- grade_db %>% filter(course_id == "ST101")
q1
# Source:   lazy query [?? x 3]
# Database: sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
  course_id student_id final_mark
  <chr>          <int>      <int>
1 ST101      201921323         78
2 ST101      201985603         60
3 ST101      202003219         47
```

The function `filter()` selects the rows in the `grade_db` which satisfy the condition `course_id == "ST101"`.

We can use the function `show_query()` to show the SQL query that **dbplyr** produced, when we run the code above:

```
show_query(q1)
<SQL>
SELECT *
FROM `Grade`
WHERE (`course_id` = 'ST101')
```

Getting Names of Students in Alphabetic Order

If we want to work on more than one table in **dplyr**, we can use join or set operations. In this example we show how to use `inner_join()`. `arrange()` is then used to order the query result.

```
q2 <- inner_join(student_db, grade_db) %>%
  filter(course_id == "ST101") %>%
  select(name) %>%
  arrange(name)
q2
# Source:     lazy query [?? x 1]
# Database:   sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
# Ordered by: name
  name
  <chr>
1 Ava Smith
2 Charlie Jones
3 Emily Wood
```

The corresponding SQL query is:

```
show_query(q2)
<SQL>
SELECT `name`
FROM (SELECT `LHS`.`student_id` AS `student_id`, `name`, `year`,
`course_id`, `final_mark`
FROM `Student` AS `LHS`
INNER JOIN `Grade` AS `RHS`
ON (`LHS`.`student_id` = `RHS`.`student_id`)
)
WHERE (`course_id` = 'ST101')
ORDER BY `name`
```

Getting Courses Taken by Ava Smith or Freddie Harris

Here we use `inner_join` to specify which attribute should be used to join by, in this case `course_id`. As both `student_db` and the `course_db` have the attribute `name`, we use the argument `suffix` to rename the attribute `name` to `name.student` and `name.course`, correspondingly. In this way we eliminate ambiguity.

```
q3 <- inner_join(student_db, grade_db, by = "student_id") %>%
    inner_join(course_db, by = "course_id", suffix = c(".student",
".course")) %>%
    filter(name.student == 'Ava Smith' | name.student == 'Freddie Harris')
%>%
    select(name.course) %>%
    distinct()
q3
# Source:   lazy query [?? x 1]
# Database: sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
  name.course
  <chr>
1 programming for data science
```

```
2 Managing and Visualising Data
3 Databases
```

The corresponding SQL query is:

```
show_query(q3)
<SQL>
SELECT DISTINCT `name.course`
FROM (SELECT `student_id`, `LHS`.`name` AS `name.student`, `year`,
`LHS`.`course_id` AS `course_id`, `final_mark`, `RHS`.`name` AS
`name.course`, `capacity`
FROM (SELECT `LHS`.`student_id` AS `student_id`, `name`, `year`,
`course_id`, `final_mark`
FROM `Student` AS `LHS`
INNER JOIN `Grade` AS `RHS`
ON (`LHS`.`student_id` = `RHS`.`student_id`)
) AS `LHS`
INNER JOIN `Course` AS `RHS`
ON (`LHS`.`course_id` = `RHS`.`course_id`)
)
WHERE (`name.student` = 'Ava Smith' OR `name.student` = 'Freddie Harris')
```

Note the computer-generated SQL code that **dplyr** created internally is more complicated than the SQL code we wrote for the same query.

Calculating Average Mark for Each Course

The combination of the verbs `group_by()` and `summarize()` are used to calculate the average `final_mark` for each `course_id`.

```
q4 <- grade_db %>%
  group_by(course_id) %>%
  summarize(avg_mark = mean(final_mark, na.rm = TRUE))
q4
# Source:   lazy query [?? x 2]
# Database: sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
  course_id avg_mark
  <chr>        <dbl>
1 ST101         61.7
2 ST115         82.3
3 ST207         66.5
```

The corresponding SQL query is:

```
show_query(q4)
<SQL>
SELECT `course_id`, AVG(`final_mark`) AS `avg_mark`
FROM `Grade`
GROUP BY `course_id`
```

Disconnecting From the Database

After we finish manipulating the database, we can close the connection using the function `dbDisconnect()` from `DBI`:

```
dbDisconnect(conn)
```

Useful Links and Resources

- Using **dplyr** with databases: A guide on how to use **dplyr** with databases from RStudio
- **dplyr** vignettes: A introduction to **dplyr**
- **dbplyr** SQL Translation

Creating and Manipulating Databases in Python

** Note: The code chunks below should be run in the following order **

Using Databases With Python

We have shown how to create, update and query a database using DB Browser for SQLite and in R. Now we will illustrate how the same thing can be done in Python. Again we will continue using the university example.

Connecting to Databases Using Python

We import the module **sqlite3** and use the function `connect()` to create an object, `conn`, to connect to the SQLite driver to manipulate the database `University.db`. If the database `University.db` exists in your working directory, the following code chunk will remove it.

```
# This makes sure you can run this notebook multiple times without errors
import os
try:
    os.remove('University.db')
except OSError:
    pass
import sqlite3
conn = sqlite3.connect('University.db')
```

Creating Tables Using Python

Now we are going to create some tables to the database `University.db`. Like before, we will create the tables using the data saved in the CSV files. We first load the CSV files into `DataFrame` in Python:

```
import pandas as pd


student = pd.read_csv("student.csv")
course = pd.read_csv("course.csv")
grade = pd.read_csv("grade.csv")
```

We then write record stored in `DataFrame`s `student`, `grade` and `course` as tables to the database `University.db` using the `DataFrame` method `to_sql()`.

```
# index = False to ensure the DataFrame row index is not written into the SQL table
s

student.to_sql('Student', con = conn, index = False)

course.to_sql('Course', con = conn, index = False)

grade.to_sql('Grade', con = conn, index = False)
```

Again, we can check if the database is created properly by opening the database in DB Browser for SQLite and browse the tables.

Manipulate Databases Using Python

We can manipulate databases in Python by the `execute()` and `fetchall()` methods from the **sqlite3** module which performs SQL commands. This allows us to leverage the SQL commands we have learned to manipulate the databases in Python. We first need to create a cursor object `c`:

```
c = conn.cursor()
```

After that we can execute the SQL commands we learned before using the function `execute()` and `fetchall()`. For example, if we want to get all the tables in the database, we can run:

```
c.execute('''

SELECT name

FROM sqlite_master

WHERE type='table'

''')

<sqlite3.Cursor object at 0x1447c1f80>
```

The result is not returned until we run `fetchall`:

```
c.fetchall()

[('Student',), ('Course',), ('Grade',)]
```

We can see there are three tables in the database. If we want to browse the table `Student` we can run (here we display the results as `pandas` `DataFrame`):

```
import pandas as pd

q = c.execute("SELECT * FROM Student").fetchall()

pd.DataFrame(q)

           0               1  2

0  201921323       Ava Smith  2

1  201832220     Ben Johnson  3

2  202003219   Charlie Jones  1

3  202045234      Dan Norris  1
```

```
4  201985603      Emily Wood  1
5  201933222  Freddie Harris  2
6  201875940    Grace Clarke  2
```

Note here we combine the use of `execute()` and `fetchall()` in one line.

Add a New Table

We can add a new table by running the SQL command through `execute()`:

```
c.execute('''
CREATE TABLE Teacher (staff_id TEXT PRIMARY KEY,
name TEXT)
''')
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit() # save (commit) the changes
```

When we list the tables, we can see four tables.

```
c.execute('''
SELECT name
FROM sqlite_master
WHERE type='table'
''').fetchall()
[('Student',), ('Course',), ('Grade',), ('Teacher',)]
```

Delete a Table

We can delete a table by running the SQL command through `execute()`:

```
c.execute("DROP TABLE Teacher")
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit()
```

When we list the tables, we can see three tables.

```
c.execute('''
SELECT name
FROM sqlite_master
WHERE type='table'
''').fetchall()
[('Student',), ('Course',), ('Grade',)]
```

Insert Tuples/Rows

Insert the year 1 student Harper Taylor with student ID 202029744 to Student:

```
c.execute("INSERT INTO Student VALUES(202029744, 'Harper Taylor', 1)")
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit()
```

When we browse the table, we can see the new row is added.

```
q = c.execute("SELECT * FROM Student").fetchall()
pd.DataFrame(q)
           0              1  2
0  201921323       Ava Smith  2
1  201832220     Ben Johnson  3
2  202003219   Charlie Jones  1
3  202045234      Dan Norris  1
4  201985603      Emily Wood  1
5  201933222  Freddie Harris  2
6  201875940    Grace Clarke  2
7  202029744   Harper Taylor  1
```

Update Tuples/Rows

Update the student ID of student Harper Taylor to 201929744:

```
c.execute('''
UPDATE Student
SET student_id = "201929744"
WHERE name = "Harper Taylor"
''')
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit()
```

When we browse the table, we can see the row has changed.

```
q = c.execute("SELECT * FROM Student").fetchall()
pd.DataFrame(q)
           0              1  2
0  201921323       Ava Smith  2
1  201832220     Ben Johnson  3
2  202003219   Charlie Jones  1
3  202045234      Dan Norris  1
4  201985603      Emily Wood  1
5  201933222  Freddie Harris  2
```

```
6  201875940    Grace Clarke  2
7  201929744   Harper Taylor  1
```

Delete Tuples/Rows

Delete the record for the student Harper Taylor from table `Student`:

```
c.execute('''
DELETE FROM Student
where name = "Harper Taylor"
''')
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit()
```

When we browse the table, we can see the row has been removed.

```
q = c.execute("SELECT * FROM Student").fetchall()
pd.DataFrame(q)
           0              1  2
0  201921323       Ava Smith  2
1  201832220     Ben Johnson  3
2  202003219   Charlie Jones  1
3  202045234      Dan Norris  1
4  201985603      Emily Wood  1
5  201933222  Freddie Harris  2
6  201875940    Grace Clarke  2
```

Disconnecting From the Database

After we finish manipulating the database, we can close the connection using the method `close()` on `conn`:

```
conn.close()
```

Useful Links and Resources

- **sqlite3**

Querying Databases in Python

** Note: The code chunks below should be run in the following order **

Using Databases With Python

In the first notebook we have seen how to use Python to connect, create and manipulate a database. Now we will use Python to query databases using the database `University.db` that we have created in the first notebook.

Connecting to Databases Using Python

We first connect to the database `University.db` and list all the tables. We should see all the three tables `Student`, `Course`, and `Grade` that we have created in the first notebook.

```
import sqlite3

conn = sqlite3.connect('University.db')

c = conn.cursor()

c.execute("SELECT name FROM sqlite_master WHERE type='table'").fetchall()

[('Student',), ('Course',), ('Grade',)]
```

Querying Databases Using Python

We can query databases in Python by the `execute()` and `fetchall()` methods from the `sqlite3` module which performs SQL commands. Here we display the results as a `Pandas` data frame. The SQL commands used here have been discussed in the previous notebooks.

Example 1: Get Grades of the Course `course_id` `ST101`

```
q1 = c.execute('''

SELECT final_mark

FROM Grade

WHERE course_id = 'ST101'

''').fetchall()


import pandas as pd

pd.DataFrame(q1)

    0

0  78

1  60
```

```
2  47
```

Example 2: Get Names of Students in Alphabetical Order

```
q2 = c.execute('''
SELECT Student.name
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name
''').fetchall()

pd.DataFrame(q2)
               0
0      Ava Smith
1  Charlie Jones
2     Emily Wood
```

Example 3: Get Courses Taken by Ava Smith or Freddie Harris

```
q3 = c.execute('''
SELECT DISTINCT Course.name
FROM Student, Grade, Course
WHERE (Student.name ='Ava Smith' OR Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id = Grade.course_id
''').fetchall()

pd.DataFrame(q3)
                                0
0   programming for data science
1  Managing and Visualising Data
2                      Databases
```

Example 4: Calculate Average Mark for Each Course

```
q4 = c.execute('''
SELECT course_id, AVG(final_mark) as avg_mark
FROM Grade
GROUP BY course_id
```

```
''').fetchall()


pd.DataFrame(q4)
       0          1
0  ST101  61.666667
1  ST115  82.333333
2  ST207  66.500000
```

Disconnecting From Databases

After we finish manipulating the database, we can close the connection using the method `close` on `conn`:

```
conn.close()
```

Useful Links and Resources

- **`sqlite3`**