CSCI 621 DATABASE SYSTEM IMPLEMENTATION

H2 Assignment 3

Group 4:
Amanraj
Kruthi Nagabhushan
Aarti Nayak
Swetna Tribhuvan

Requirement:
Implement an enhanced feature to H2 database in terms of query processing and optimization. Such features can be adding a new query operator (e.g., new aggregate function, approximate search) and adding more join operations.

SECTION A: IMPLEMENTATION OF A CUSTOMIZED AGGREGATE FUNCTION

A) Description of the Enhanced Features:

- To enhance the H2 database in terms of query processing and optimization, we have implemented a new aggregate function called G_MEAN to perform geometric operations and EXTENDED_MODE to calculate mode of the given column respectively .
- Geometric mean is a mathematical function that is useful for many statistical computations like stock indexes,
- For a set of values the geometric mean is calculated by calculating the product of all the values and then taking the nth root of the product; where n is the total number of values provided.
- Eg:
    - We have 5 values 1, 2, 3, 4, 5.
    - Therefore, in order to calculate the geometric mean of these numbers; we calculate the product of the same which is 120.
    - Thereafter we calculate the product, we take the 5th root of the product, which is 2.60.

- Our customized aggregate function is G_MEAN which is useful to find the geometric mean of the given column.
- Mode is that number that occurs the highest number of times. Example: The mode of {4 , 2, 4, 3, 2, 2} is 2 because it occurs three times, which is more than any other number.
- Our customized aggregate function is EXTENDED_MODE which is useful to find the mode of the given column. H2 already has an aggregate function known as mode for the same operation. We have still implemented it in a separate class in different ways.

B) <u>Implementation:</u>

CASE 1 : G_MEAN

H2 database has made provisions for users to implement their own customized aggregate functions depending upon their needs. It has provided a file called 'AggregateFunction.java' in the 'api' folder that contains an interface and its methods that need to be implemented by the user in order to incorporate their own aggregate function. Making use of these provisions, we have created a new java file which implements AggregateFunction.java. This new java file has implemented a few methods which are described briefly below.

1. Creating the functionality for calculating the geometric mean:

- **File created:** AggregateFunctionGeoMean.java
- **File path:** h2/src/main/AggregateFunctionGeoMean.java

The methods that were implemented by our file are:

a) getType(int[] inputTypes):

- This method expects us to return the data type of our aggregate function.
- Since here we are calculating the geometric mean, we do not expect an integer value for all the cases, therefore, we have set this type to 'DOUBLE'.
- This method also expects you to validate the number of input arguments.
- However, our function can accept any number of values with no restriction of the minimum number of values.

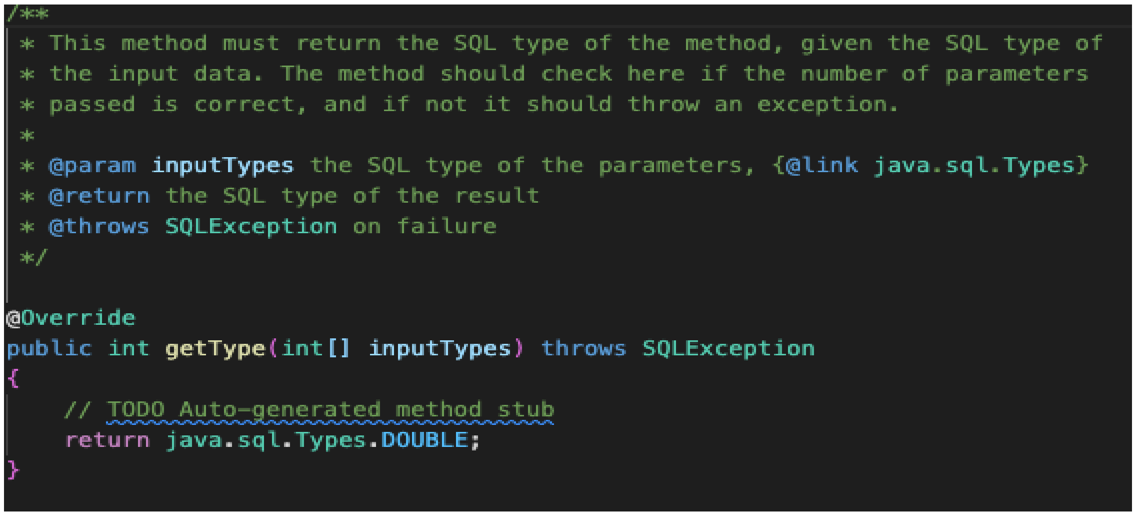

```
/**
 * This method must return the SQL type of the method, given the SQL type of
 * the input data. The method should check here if the number of parameters
 * passed is correct, and if not it should throw an exception.
 *
 * @param inputTypes the SQL type of the parameters, {@link java.sql.Types}
 * @return the SQL type of the result
 * @throws SQLException on failure
 */

@Override
public int getType(int[] inputTypes) throws SQLException
{
    // TODO Auto-generated method stub
    return java.sql.Types.DOUBLE;
}
```

Figure 1.1

b) add(Object value):

- This method is called for each row and for all the values of a particular column, these are accumulated for processing in this function.
- Here we have created an array list of the type double and all the values for a particular column shall be added to this for processing and that arraylist is returned.

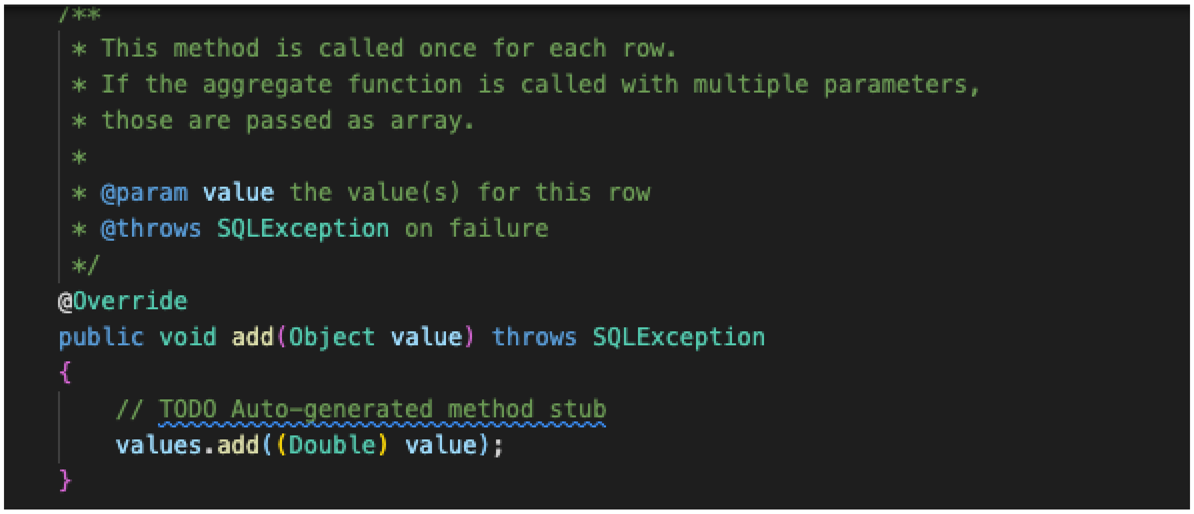
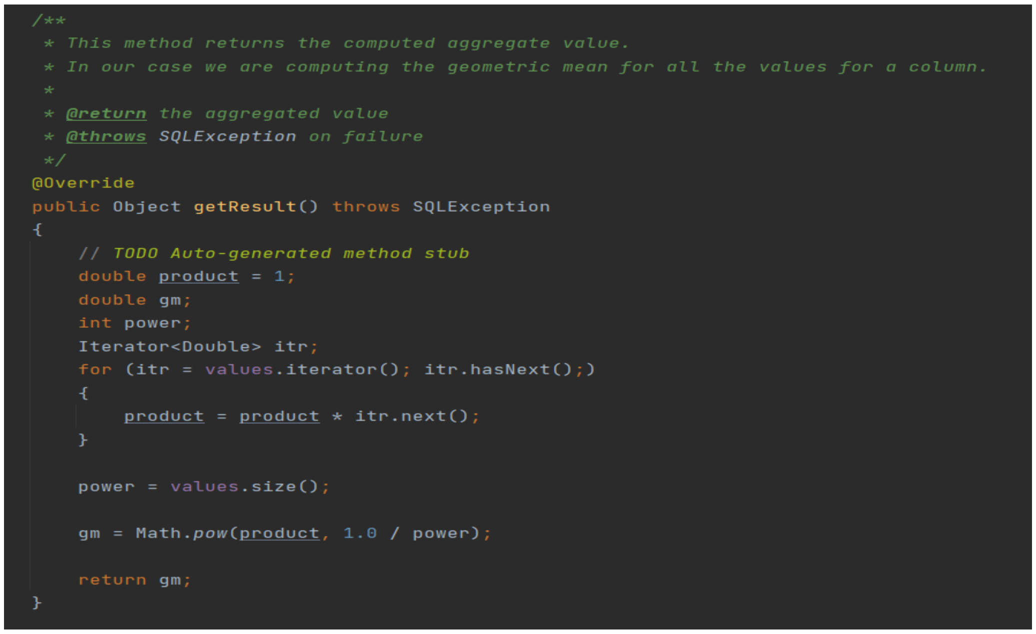

```
/**
 * This method is called once for each row.
 * If the aggregate function is called with multiple parameters,
 * those are passed as array.
 *
 * @param value the value(s) for this row
 * @throws SQLException on failure
 */
@Override
public void add(Object value) throws SQLException
{
    // TODO Auto-generated method stub
    values.add((Double) value);
}
```

Figure 1.2

c) getResult():

- This method specifies the actual working for the aggregate function.

- In our case we have written the algorithm for calculating the geometric mean.
- We have created a product variable to store the product of all the values of the incoming rows and we have obtained those from iterating the array list created above.
- Finally by using the Java Math library, we have calculated the nth root of our product by calculating the size of our array and finally we have returned that result.



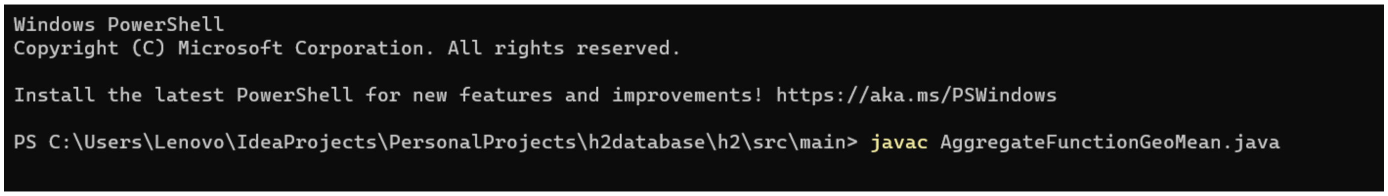
```java
/**
 * This method returns the computed aggregate value.
 * In our case we are computing the geometric mean for all the values for a column.
 *
 * @return the aggregated value
 * @throws SQLException on failure
 */
@Override
public Object getResult() throws SQLException
{
    // TODO Auto-generated method stub
    double product = 1;
    double gm;
    int power;
    Iterator<Double> itr;
    for (itr = values.iterator(); itr.hasNext();)
    {
        product = product * itr.next();
    }

    power = values.size();

    gm = Math.pow(product, 1.0 / power);

    return gm;
}
```

Figure 1.3

2. Adding the implemented class file to the JAR file:

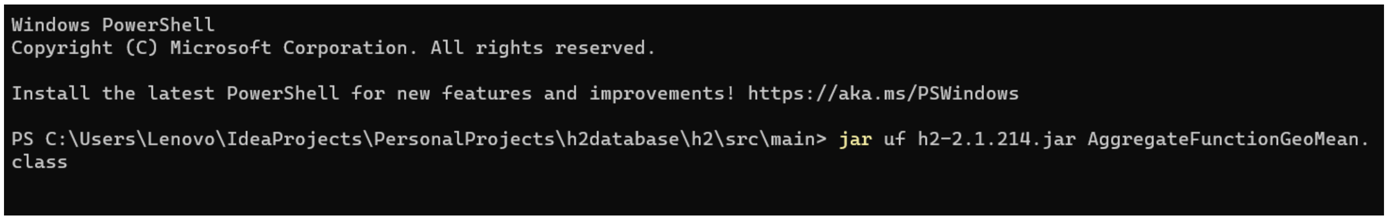
- **File Changed:** h2-2.1.214.jar
- **File Path:** ".\H2\bin\h2-2.1.214.jar"
- Now that we have created our java file AggregateFunctionGeoMean.java, in order to enable the database to read the file, we have created the class file and added it to the jar file.
- We have used the javac command, followed by our file name to create a class file.

- After we have used the properties of the jar file, (update(-u) and change directory(-C) and it’s optional) and added our file to the jar file followed by the creation of a subsequent new jar file embedded with our file.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Lenovo\IdeaProjects\PersonalProjects\h2database\h2\src\main> javac AggregateFunctionGeoMean.java
```

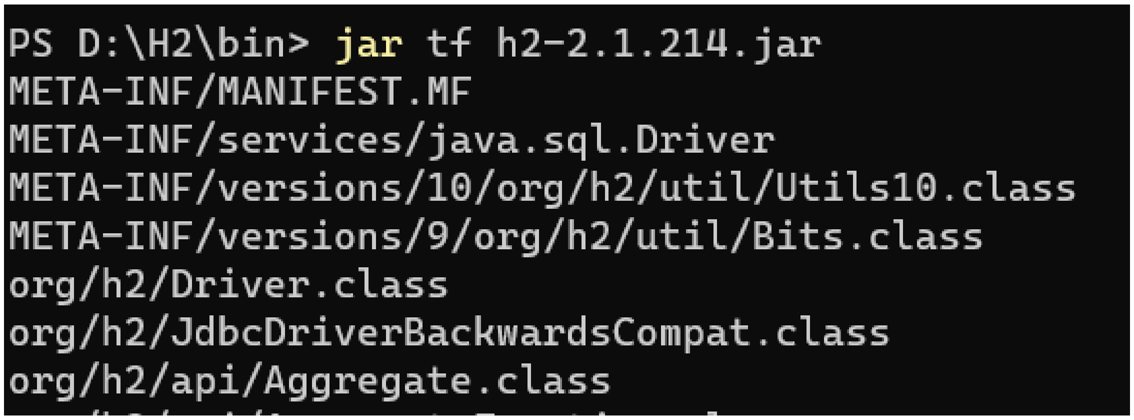
Figure 1.4 To create java class file



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Lenovo\IdeaProjects\PersonalProjects\h2database\h2\src\main> jar uf h2-2.1.214.jar AggregateFunctionGeoMean.
class
```

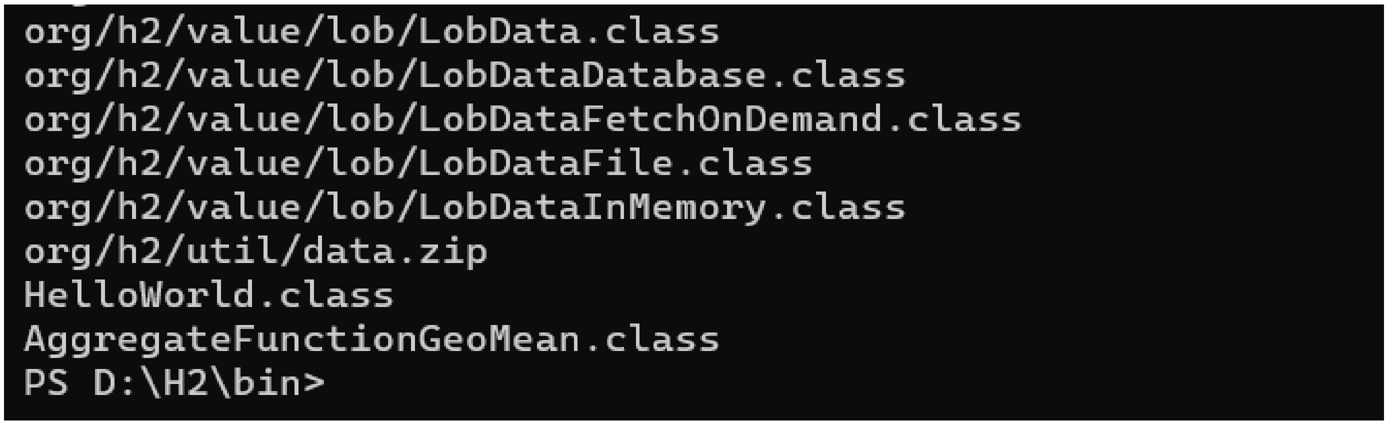
Figure 1.5 Adding java class to the jar file

To check if the files are added to the jar file, we run the “ jar tf h2-2.1.214.jar” command on the console.



```
PS D:\H2\bin> jar tf h2-2.1.214.jar
META-INF/MANIFEST.MF
META-INF/services/java.sql.Driver
META-INF/versions/10/org/h2/util/Utils10.class
META-INF/versions/9/org/h2/util/Bits.class
org/h2/Driver.class
org/h2/JdbcDriverBackwardsCompat.class
org/h2/api/Aggregate.class
```

Figure 1.6 Running jar command



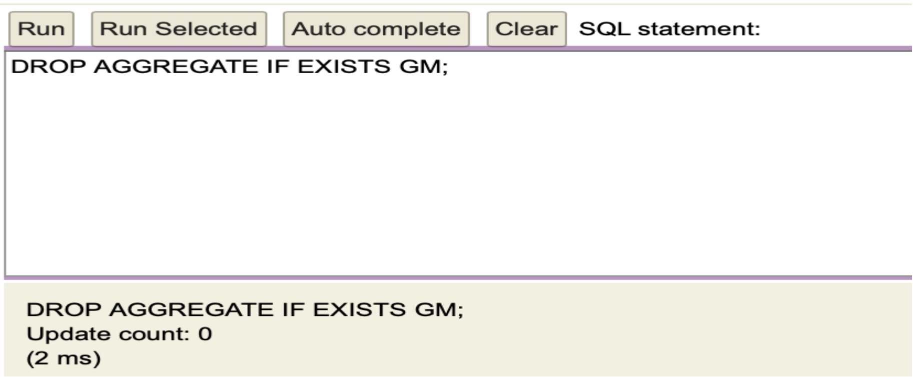
```
org/h2/value/lob/LobData.class
org/h2/value/lob/LobDataDatabase.class
org/h2/value/lob/LobDataFetchOnDemand.class
org/h2/value/lob/LobDataFile.class
org/h2/value/lob/LobDataInMemory.class
org/h2/util/data.zip
HelloWorld.class
AggregateFunctionGeoMean.class
PS D:\H2\bin>
```

Figure 1.7 AggregateFunctionGeoMean.class being added to jar file

Next, restart/relaunch H2 to make sure that the ports are not being used/ busy or occupied.

3. Creating the function:

- After making all the changes to the jar file, we recompiled the jar file and executed the same.
- Thereafter, we started with the creation of our custom aggregate function using the DDL command provided by H2, 'CREATE AGGREGATE'.
- This command helps the user to create a customized aggregate function just like creating a table or a view.
- The two commands for the same are:
  - DROP AGGREGATE IF EXISTS ARREGATE_NAME:
    - This command eliminates any aggregate function if it has been previously created with the same name.

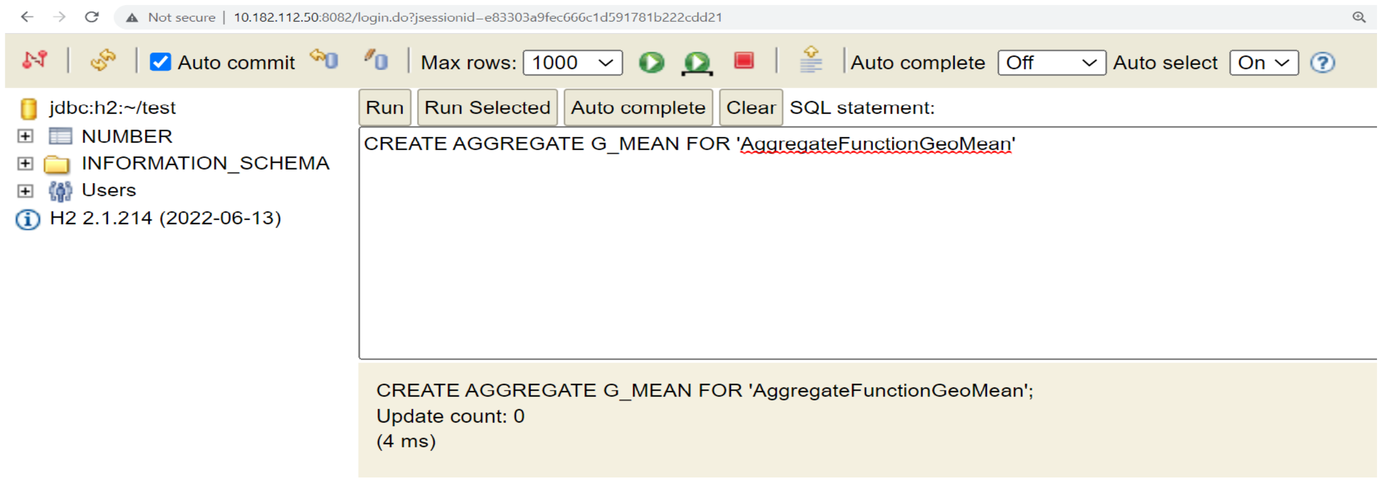


```
DROP AGGREGATE IF EXISTS GM;
```

```
DROP AGGREGATE IF EXISTS GM;
Update count: 0
(2 ms)
```

Figure 1.7

- CREATE AGGREGATE ARREGATE_NAME FOR "AGGREGATE FILE PATH":
  - This command finds the file we have given the path of and executes it to create our aggregate function.
  - In our case, ‘G_MEAN’ is the tag and 'AggregateFunctionGeoMean' is our java file.



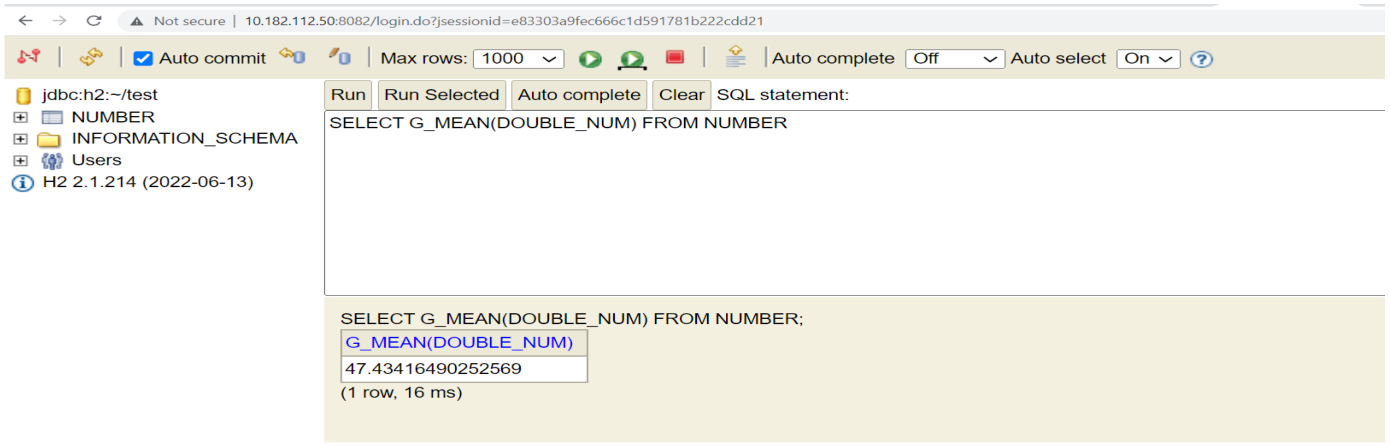
```
CREATE AGGREGATE G_MEAN FOR 'AggregateFunctionGeoMean'
```

```
CREATE AGGREGATE G_MEAN FOR 'AggregateFunctionGeoMean';
Update count: 0
(4 ms)
```

Figure 1.8

4. Testing the function G_MEAN:

- To test, we use the select clause.
- SELECT G_MEAN(COLUMN_NAME) FROM TABLE.
- In our case, the column name is DOUBLE_NUM and table is NUMBER.



```
SELECT G_MEAN(DOUBLE_NUM) FROM NUMBER
```

```
SELECT G_MEAN(DOUBLE_NUM) FROM NUMBER;
```

G_MEAN(DOUBLE_NUM)
47.43416490252569

(1 row, 16 ms)

Figure 1.9

CASE 2: EXTENDED_MODE

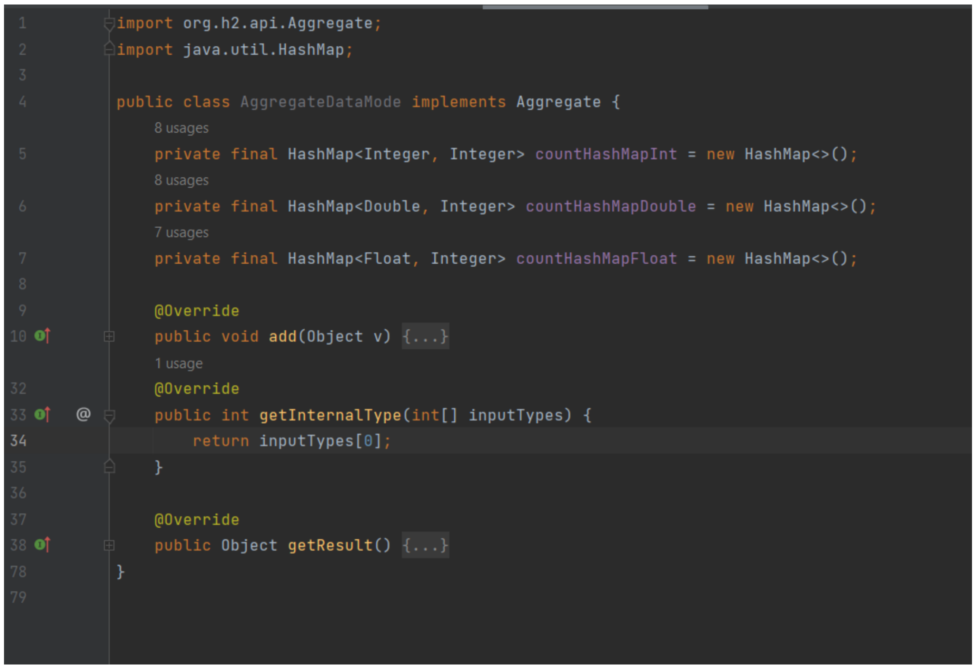
Though the Mode aggregate function is also existing, along with the geometric mean, we also tried to add a customized MODE function in a similar way.

1. **Creating the functionality for calculating the geometric mean:**

   - **File created:** AggregateDataMode.java
   - **File path:** h2/src/main/AggregateDataMode.java

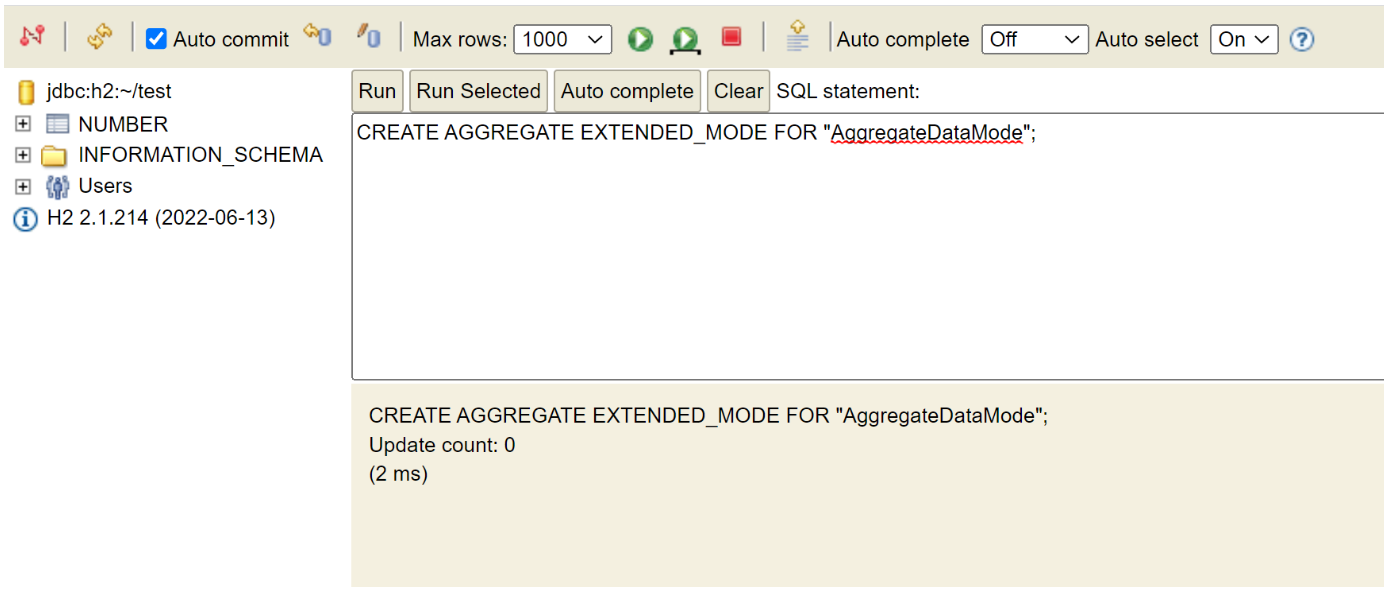
   This java file implements the Aggregate class provided by H2. It has few methods that are implemented that work similar to those of the AggregateFunctionGeoMean class as explained earlier.
   After creating the functionality, we used the javac command to get java class file which is further added to jar file using the same commands which we used incase of G_MEAN.

2. **Creating the function:**
   After adding class files to the jar, we restart the H2.
   Using 'CREATE AGGREGATE' to create the aggregate function EXTENDED_MODE.



```
import org.h2.api.Aggregate;
import java.util.HashMap;

public class AggregateDataMode implements Aggregate {
    8 usages
    private final HashMap<Integer, Integer> countHashMapInt = new HashMap<>();
    8 usages
    private final HashMap<Double, Integer> countHashMapDouble = new HashMap<>();
    7 usages
    private final HashMap<Float, Integer> countHashMapFloat = new HashMap<>();

    @Override
    public void add(Object v) {...}
    1 usage
    @Override
    public int getInternalType(int[] inputTypes) {
        return inputTypes[0];
    }

    @Override
    public Object getResult() {...}
}
```

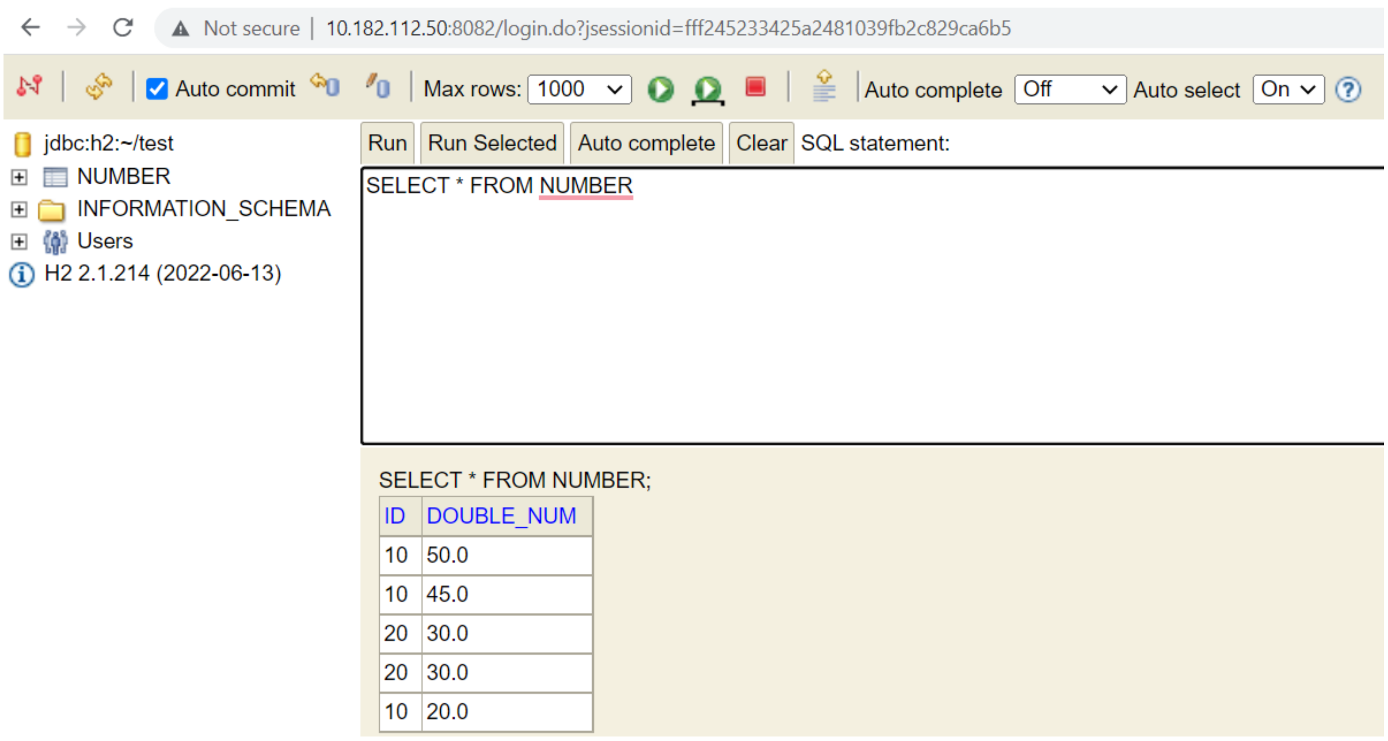
Figure 1.10



```
CREATE AGGREGATE EXTENDED_MODE FOR "AggregateDataMode";
```

```
CREATE AGGREGATE EXTENDED_MODE FOR "AggregateDataMode";
Update count: 0
(2 ms)
```

Figure 1.11 Creating the function



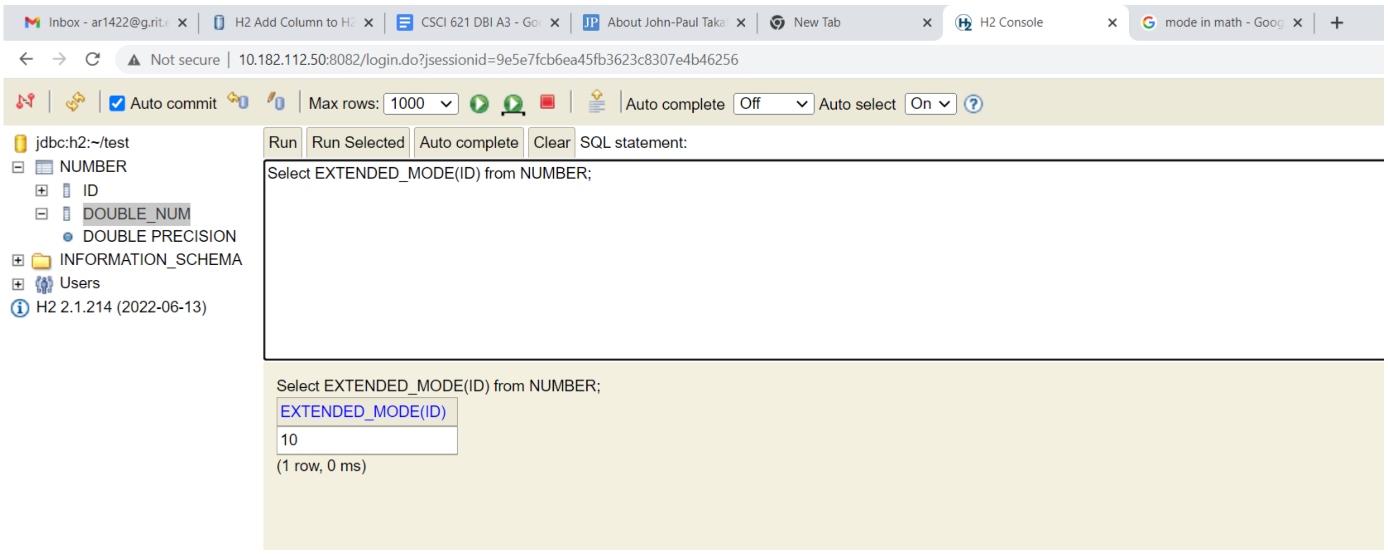
Auto commit | Max rows: 1000 | Auto complete: Off | Auto select: On

```
SELECT * FROM NUMBER
```

```
SELECT * FROM NUMBER;
```

ID	DOUBLE_NUM
10	50.0
10	45.0
20	30.0
20	30.0
10	20.0

Figure 1.12 Data in table



Auto commit | Max rows: 1000 | Auto complete: Off | Auto select: On

```
Select EXTENDED_MODE(ID) from NUMBER;
```

```
Select EXTENDED_MODE(ID) from NUMBER;
```

EXTENDED_MODE(ID)
10

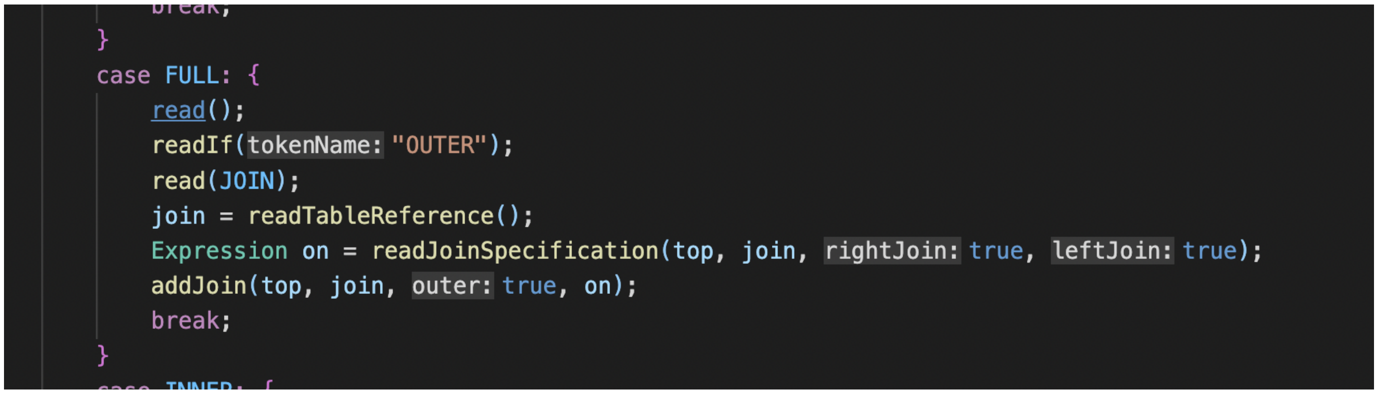
(1 row, 0 ms)

Figure 1.12 Testing EXTENDED_MODE

SECTION B : Enhancing of H2 database feature by adding Join conditions:

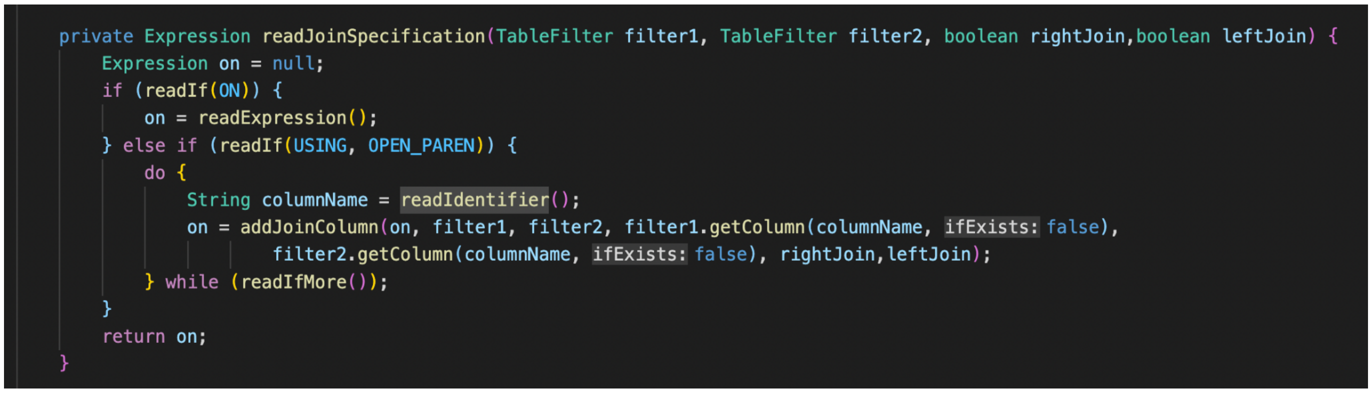
- We have added the Full Outer Join method to our H2.
- H2 database already supports the Outer and Inner as well as Natural Join conditions.
- Background about what Full outer Join is:
  Full Outer Join is a type of outer join in which the two tables are joined on the common column and values from both the tables are kept even if they include null values with the other table.
  Full Outer Join can help to ensure that there is no data loss due to the join operation since it retains all its values.
  In the code we made the following changes in Parser.java file
  Location of file: /…/h2/src/main/org/h2/command/

  As we can see in figure 2.1 the case for full outer join has been added in the parser file. This condition to ensure that the full case is analyzed along with the outer keyword being optional since it works only if that value is read by using readIf function.
  Join keyword is essential in this situation. When the expression is read the method join specification (Figure 2.2) is called upto which provides either a null value if the conditions are not met or else it calls up to the addJoinColumn method to give the output of the columns to be joined based on the condition of what type of join it is.



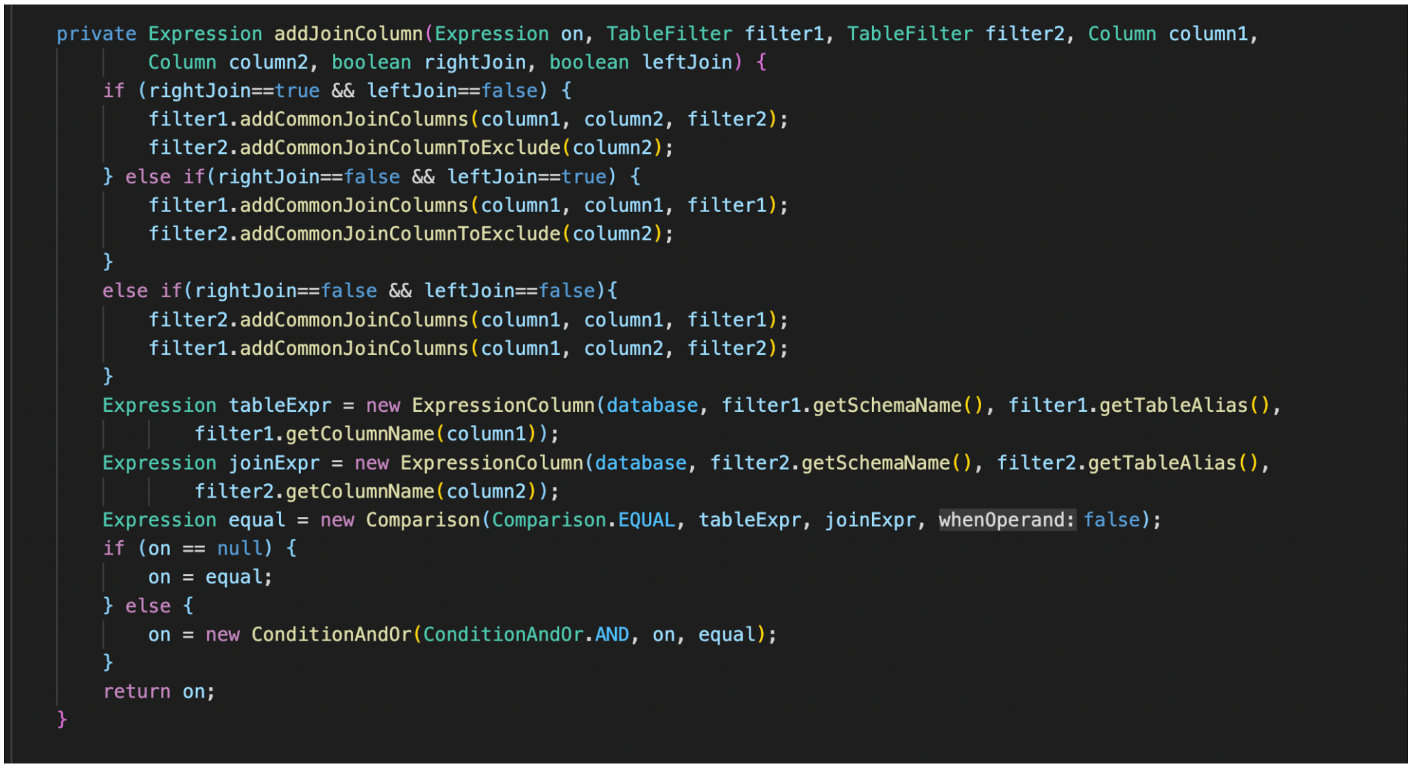
```
        break;
    }
    case FULL: {
        read();
        readIf(tokenName: "OUTER");
        read(JOIN);
        join = readTableReference();
        Expression on = readJoinSpecification(top, join, rightJoin: true, leftJoin: true);
        addJoin(top, join, outer: true, on);
        break;
    }
    case INNER: {
```

Figure:2.1



```java
private Expression readJoinSpecification(TableFilter filter1, TableFilter filter2, boolean rightJoin,boolean leftJoin) {
    Expression on = null;
    if (readIf(ON)) {
        on = readExpression();
    } else if (readIf(USING, OPEN_PAREN)) {
        do {
            String columnName = readIdentifier();
            on = addJoinColumn(on, filter1, filter2, filter1.getColumn(columnName, ifExists: false),
                    filter2.getColumn(columnName, ifExists: false), rightJoin,leftJoin);
        } while (readIfMore());
    }
    return on;
}
```

Figure:2.2

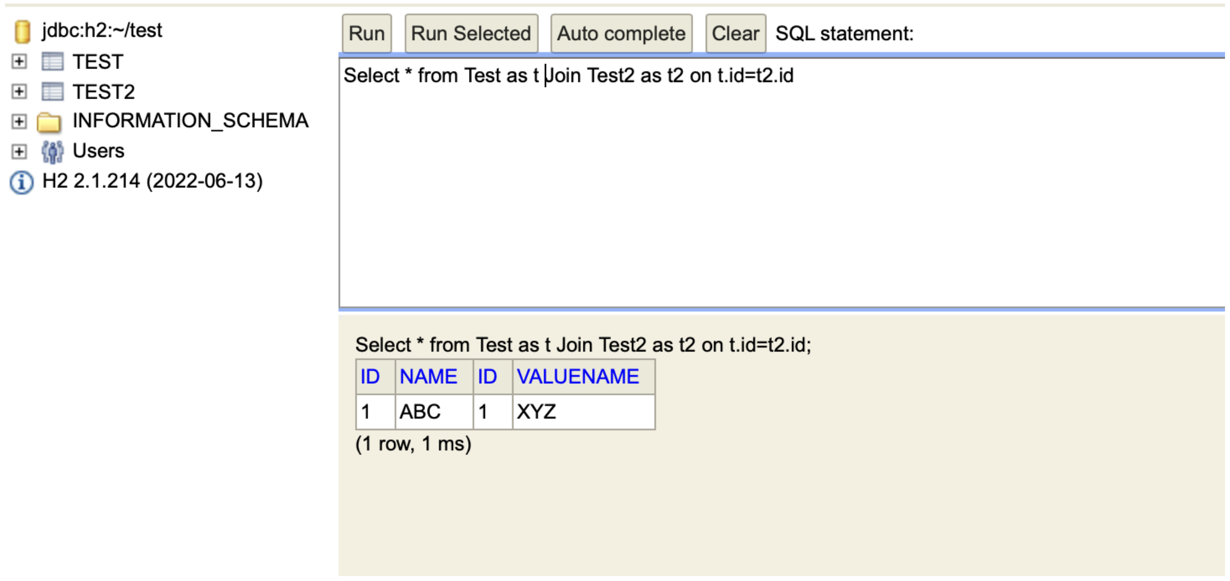


```java
private Expression addJoinColumn(Expression on, TableFilter filter1, TableFilter filter2, Column column1,
        Column column2, boolean rightJoin, boolean leftJoin) {
    if (rightJoin==true && leftJoin==false) {
        filter1.addCommonJoinColumns(column1, column2, filter2);
        filter2.addCommonJoinColumnToExclude(column2);
    } else if(rightJoin==false && leftJoin==true) {
        filter1.addCommonJoinColumns(column1, column1, filter1);
        filter2.addCommonJoinColumnToExclude(column2);
    }
    else if(rightJoin==false && leftJoin==false){
        filter2.addCommonJoinColumns(column1, column1, filter1);
        filter1.addCommonJoinColumns(column1, column2, filter2);
    }
    Expression tableExpr = new ExpressionColumn(database, filter1.getSchemaName(), filter1.getTableAlias(),
            filter1.getColumnName(column1));
    Expression joinExpr = new ExpressionColumn(database, filter2.getSchemaName(), filter2.getTableAlias(),
            filter2.getColumnName(column2));
    Expression equal = new Comparison(Comparison.EQUAL, tableExpr, joinExpr, whenOperand: false);
    if (on == null) {
        on = equal;
    } else {
        on = new ConditionAndOr(ConditionAndOr.AND, on, equal);
    }
    return on;
}
```

Figure: 2.3

The method addJoinColumn has 3 conditions that is if it is a right join,if left or full as seen in the figure. These changes enable the full outer join to be operated on the h2 database.

Apart from this H2 already supported other join operations examples of those provided below in the images.



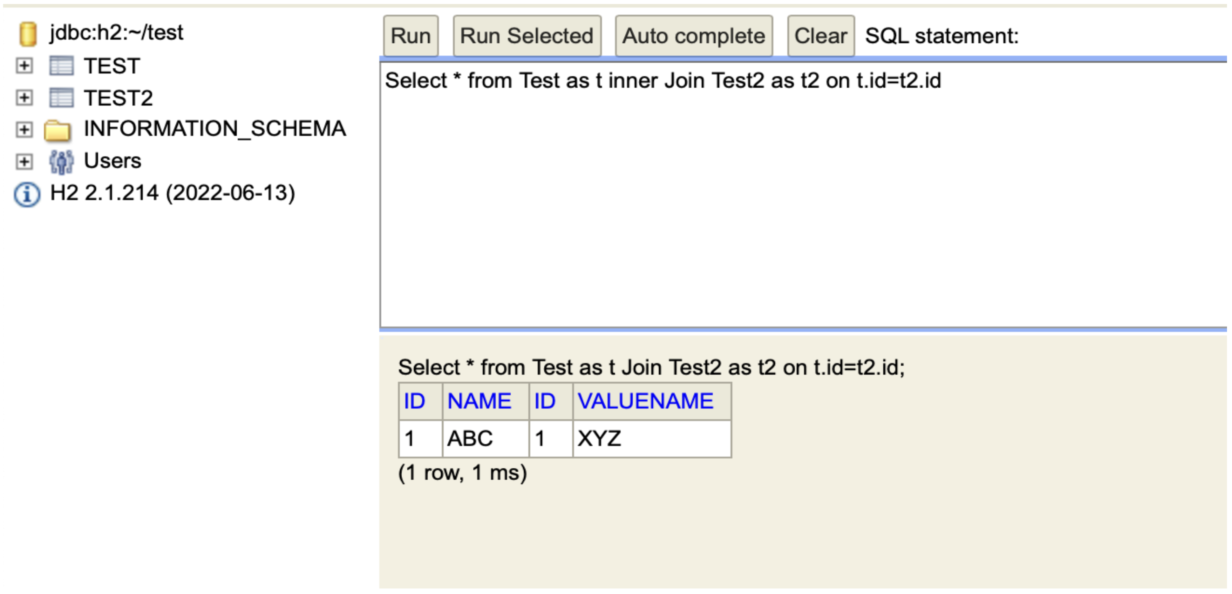
```
Select * from Test as t Join Test2 as t2 on t.id=t2.id
```

```
Select * from Test as t Join Test2 as t2 on t.id=t2.id;
```

ID	NAME	ID	VALUENAME
1	ABC	1	XYZ

(1 row, 1 ms)

Figure 2.4- Natural Join with on condition



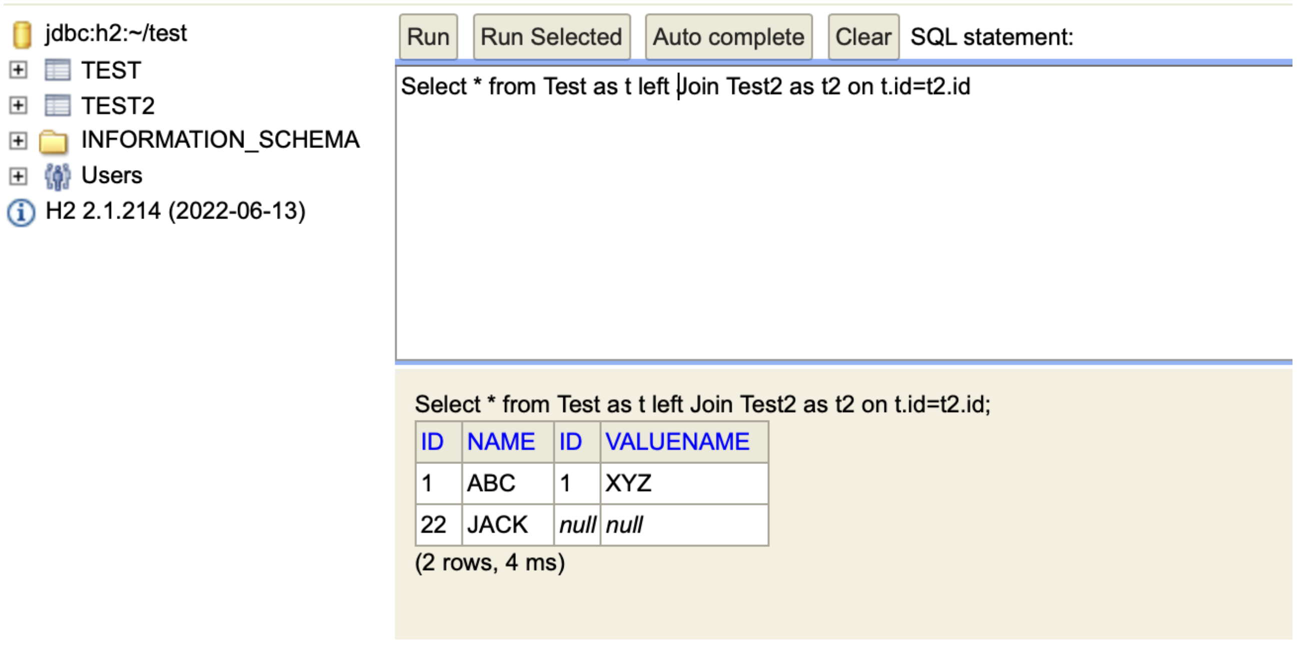
```
Select * from Test as t inner Join Test2 as t2 on t.id=t2.id
```

```
Select * from Test as t Join Test2 as t2 on t.id=t2.id;
```

ID	NAME	ID	VALUENAME
1	ABC	1	XYZ

(1 row, 1 ms)

Figure:2.5- Inner Join



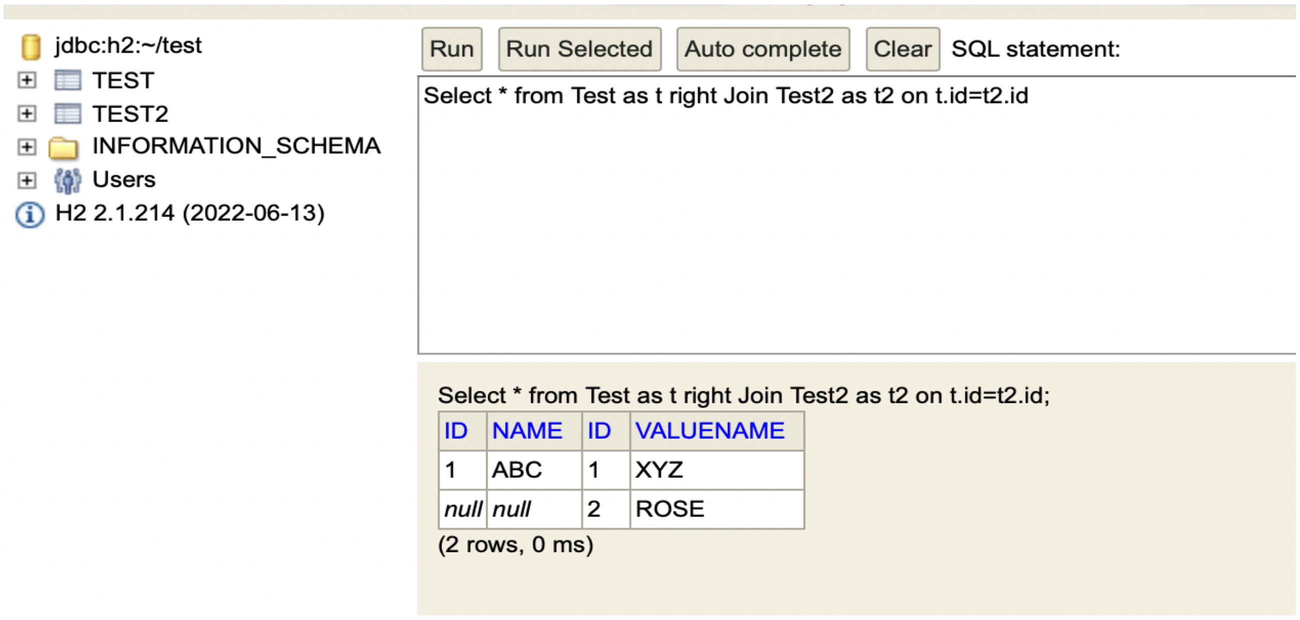
```
Select * from Test as t left Join Test2 as t2 on t.id=t2.id
```

```
Select * from Test as t left Join Test2 as t2 on t.id=t2.id;
```

ID	NAME	ID	VALUENAME
1	ABC	1	XYZ
22	JACK	*null*	*null*

(2 rows, 4 ms)

Figure:2.6- Left Join



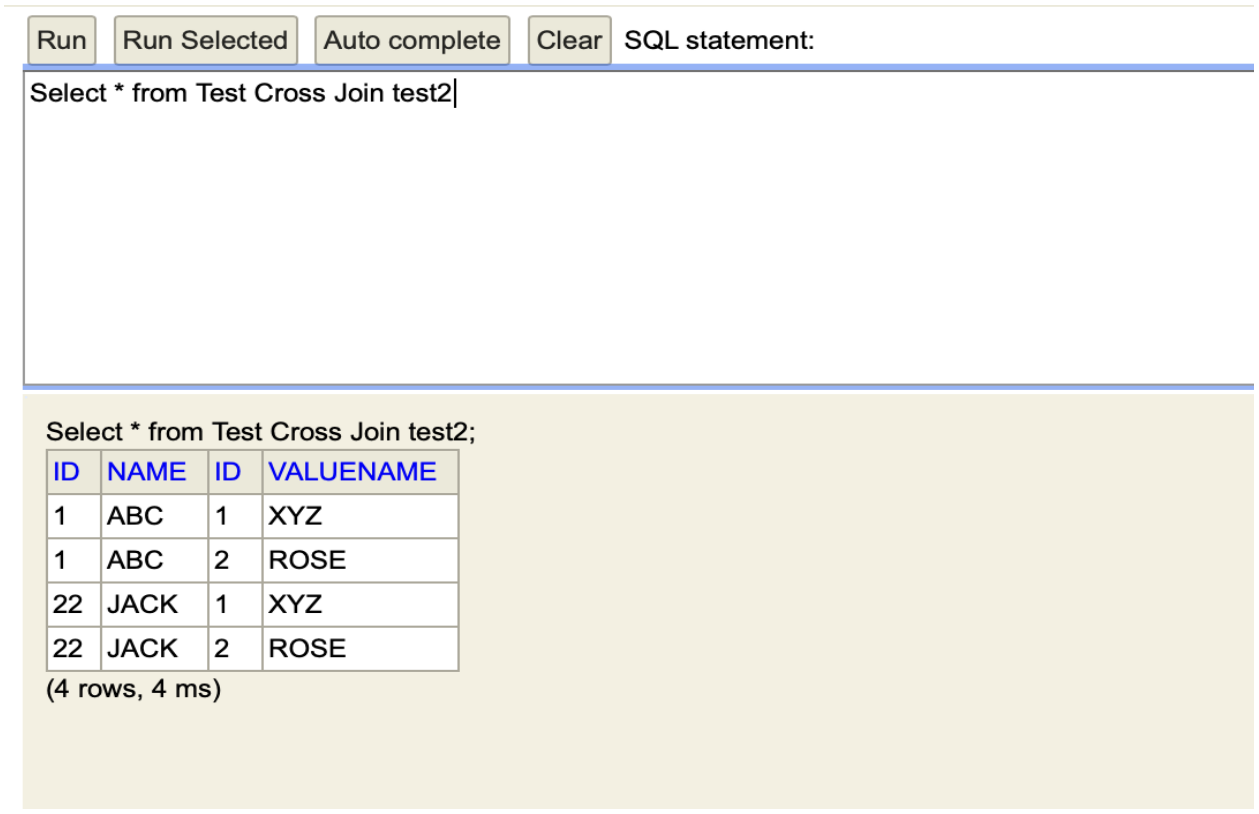
```
Select * from Test as t right Join Test2 as t2 on t.id=t2.id
```

```
Select * from Test as t right Join Test2 as t2 on t.id=t2.id;
```

ID	NAME	ID	VALUENAME
1	ABC	1	XYZ
null	*null*	2	ROSE

(2 rows, 0 ms)

Figure:2.7- Right Join



```
Select * from Test Cross Join test2
```

```
Select * from Test Cross Join test2;
```

ID	NAME	ID	VALUENAME
1	ABC	1	XYZ
1	ABC	2	ROSE
22	JACK	1	XYZ
22	JACK	2	ROSE

(4 rows, 4 ms)

Figure.2.8 : Cross Join

References:

- https://www.h2database.com/
- https://stackoverflow.com/questions/24611868/include-a-class-file-into-a-jar-file
- https://stackoverflow.com/questions/33047121/h2-user-defined-aggregate-function-listagg-cant-use-distinct-or-trim-on-the
- https://www.mail-archive.com/search?l=h2-database@googlegroups.com&q=subject:%22%5C%5Bh2%5C%5D+Adding+a+new+aggregate+function%22&o=newest&f=1

WORKLOAD DISTRIBUTION:

TEAM MATE	CONTRIBUTION
1. AMANRAJ LNU	Worked on understanding the addition of aggregate function and on enhancing H2 with joins .Worked on figuring out how joins worked in H2.
2. KRUTHI NAGABHUSHAN	Worked on adding and implementing the customized function EXTENDED_MODE completely and also executed Geometric mean. Worked on figuring out the addition of files to the jar file and making the functions work successfully. Worked on section A of this document.
3. AARTI NAYAK	Worked on adding and implementing the customized agg function geometric mean. Worked on its functionality. Worked on section A of this document for the geometric mean. Worked on figuring out the addition of files to the jar file.
4. SWETNA TRIBHUVAN	Worked on the addition of a full outer join. Implementation of the code along with the testing. Worked on the write up of Section B of this document in addition to adding screenshots for the same. Worked on understanding the addition of a user implemented aggregate function to H2 database.