

H2 Database Phase 3

Akash Desai
ad3059@rit.edu

Nihal Parchand
np9603@rit.edu

Viraj Chaudhari
vc6346@rit.edu

INTRODUCTION:

For implementing an enhanced feature for H2 Database in terms of query processing and optimization we have decided to implement a new aggregate function called **GreaterThanAverage** and **LessThanAverage**. Aggregate functions are functions that combines multiple records and then perform the function and return a single value as result.

- **GreaterThanAverage** is an aggregate function that calculates the average of all the values of a column in the table and returns the count of records that have a value greater than the average. The initial input is given in integer data type format and the aggregate function returns the result in integer format.
- **LessThanAverage** is an aggregate function that calculates the average of all the values of a column in the table and returns the count of records that have a value less than and equal to the average. The initial input is given in integer data type format and the aggregate function returns the result in integer format.

For performing the same operation in H2, we have to use a subquery.

LessThanAverage

Example: `Select count(*) from employee where salary <= (Select avg(salary) from employee);`

Using implemented **LessThanAverage** aggregate function:

`Select LT_AVG(salary) from employee;`

GreaterThanAverage

Example: `Select count(*) from employee where salary > (Select avg(salary) from employee);`

Using implemented **GreaterThanAverage** aggregate function:

Select **GT_AVG(salary)** from employee;

IMPLEMENTATION:

H2 allows users to create their own aggregate functions by implementing the minimal required `AggregateFunction` interface. To use this feature, the `GreaterThanAverage.java` and `LessThanAverage.java` file should implement all the abstract methods defined in the `AggregateFunction` interface (`add`, `getResult`, `getType`, `init` methods). The created java file needs to be added to the `org.h2.api` package.

```
@Override
public void add(Object o) throws java.sql.SQLException {
    Object value = o;

    if (count == 0) {
        //Initialize all the variable on processing the first row
        sum = (Integer)value;
        max = value;
        min = value;
        numbers.add((Integer)value);
    }
    else
    {
        if ((Integer)value < (Integer)min) {
            min = value; //Stores the min value
        } else if ((Integer)value > (Integer)max) {
            max = value; //Stores the max value
        }
        sum += (Integer) value; //Add to the sum
        numbers.add((Integer)value); //Add number to the list
    }
    count++;
}

/**
 * This method returns the computed aggregate value. This method must
 * preserve previously added values and must be able to reevaluate result if
 * more values were added since its previous invocation.
 *
 * @return the aggregated value
 */
@Override
public Object getResult() throws java.sql.SQLException {

    Integer avg = sum/numbers.size(); //Average of values

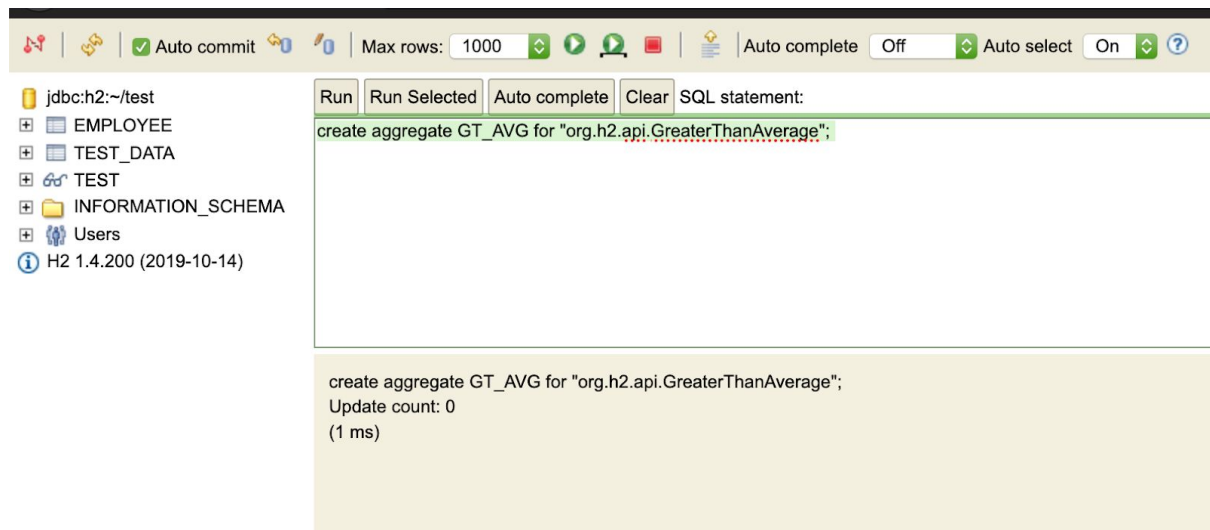
    int count = 0; //Number of values greater than average

    for(int i = 0; i < numbers.size(); i++){
        if(numbers.get(i) > avg)
            count++;
    }
    return count;
}
```

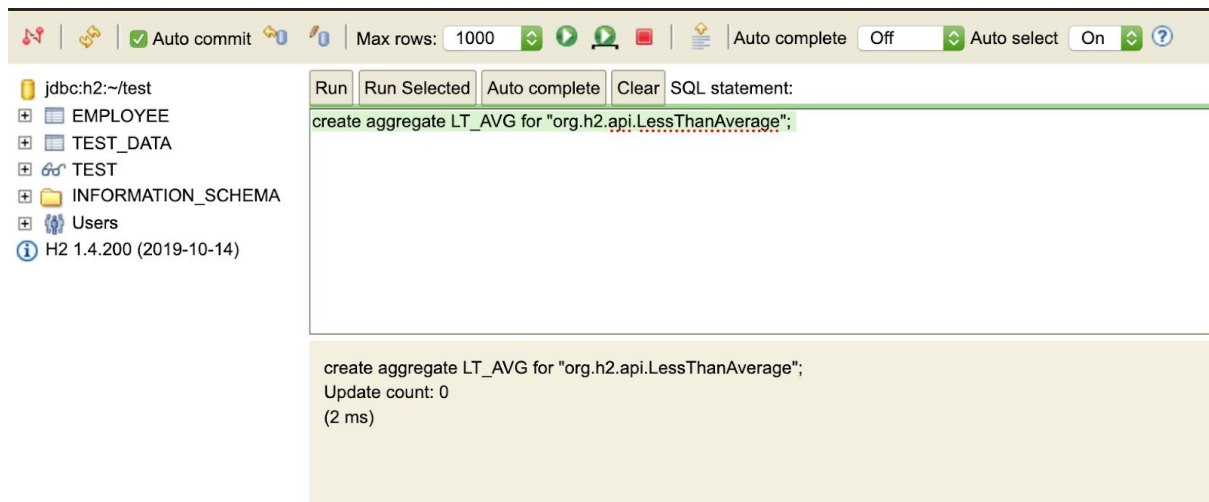
Arraylist is used to store the values of a column. In the implementation of add function we are finding minimum and maximum value of a column. Each value is added to the arraylist and sum is incremented by value. The purpose of storing the value in arraylist is to get the total number of records. This will be used for calculating average. In the implementation of getResult function the average value is calculated using the sum and size of the arraylist. Each value in the arraylist is compared with the calculated average. In LessThanAverage aggregate function count will be incremented if the value is less than or equal to the calculated average. In GreaterThanAverage aggregate function count will be incremented if the value is less than or equal to the calculated average.

For adding the aggregate function in H2 database, we need to execute the following command in the H2 interface.

Create aggregate <function_name> for “org.h2.api.<file_name>”;



Create aggregate GT_AVG for “org.h2.api.GreaterThanAverage”;



Create aggregate LT_AVG for “org.h2.api.LessThanAverage”;

DEMO:

1. Creating a table employee

Create a table employee with attributes ID INT, name VARCHAR2(10) and salary INT.

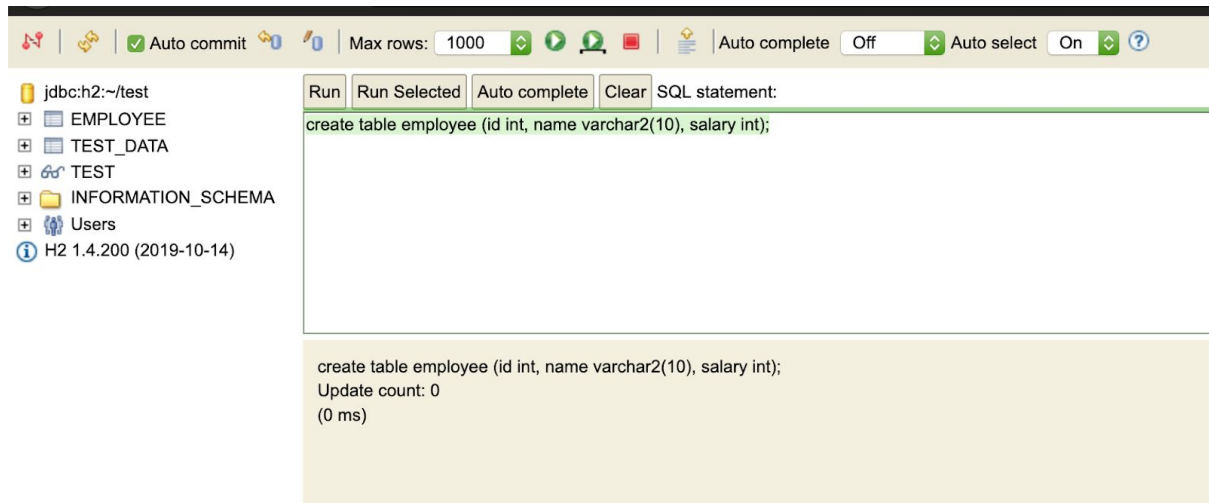


Table employee is successfully created with attributes ID INT, name VARCHAR2(10) and salary INT.

2. Displaying the column information for the created table

The screenshot shows the H2 database console interface. The left sidebar displays the database structure: jdbc:h2:~/test, EMPLOYEE, TEST_DATA, TEST, INFORMATION_SCHEMA, Users, and H2 1.4.200 (2019-10-14). The top toolbar includes buttons for Run, Run Selected, Auto complete, and Clear, along with settings for Max rows (1000), Auto complete (Off), and Auto select (On). The SQL statement entered is "show columns from employee;". The result is displayed as a table with 5 columns: FIELD, TYPE, NULL, KEY, and DEFAULT. The table contains 3 rows of data for the EMPLOYEE table. Below the table, it indicates "(3 rows, 15 ms)".

| FIELD | TYPE | NULL | KEY | DEFAULT |
|--------|-------------|------|-----|---------|
| ID | INTEGER(10) | YES | | NULL |
| NAME | VARCHAR(10) | YES | | NULL |
| SALARY | INTEGER(10) | YES | | NULL |

(3 rows, 15 ms)

3. Inserting new records in the user info table

The screenshot shows the H2 database console interface with the same sidebar and toolbar as the previous image. The SQL statement entered is a series of 12 insert statements for the EMPLOYEE table, each with a unique ID and name, and a salary value. The results are displayed below the statements, showing the update count and execution time for each insert operation.

```
insert into employee values(101,'Akash', 1000);
insert into employee values(102,'Viraj', 3000);
insert into employee values(103,'Nihal', 6500);
insert into employee values(105,'Sarang', 6000);
insert into employee values(110,'Vaibhav', 7000);
insert into employee values(115,'Karan', 8000);
insert into employee values(116,'Sayan', 5500);
insert into employee values(120,'Rohit', 7000);
insert into employee values(123,'Vivek', 1500);
insert into employee values(125,'Akshay', 7500);
```

insert into employee values(101,'Akash', 1000);
Update count: 1
(0 ms)

insert into employee values(102,'Viraj', 3000);
Update count: 1
(1 ms)

insert into employee values(103,'Nihal', 6500);
Update count: 1
(0 ms)

insert into employee values(105,'Sarang', 6000);
Update count: 1
(0 ms)

insert into employee values(110,'Vaibhav', 7000);
Update count: 1
(1 ms)

4. Displaying records of the user info table

The screenshot shows a database client interface with a sidebar on the left containing a tree view of the database structure. The main area displays the results of a SQL query. The query is `select * from employee;`. The results are shown in a table with columns `ID`, `NAME`, and `SALARY`. The table contains 10 rows of data. Below the table, it indicates `(10 rows, 4 ms)`.

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

SQL statement: `select * from employee;`

| ID | NAME | SALARY |
|-----|---------|--------|
| 101 | Akash | 1000 |
| 102 | Viraj | 3000 |
| 103 | Nihal | 6500 |
| 105 | Sarang | 6000 |
| 110 | Vaibhav | 7000 |
| 115 | Karshit | 8000 |
| 116 | Savan | 5500 |
| 120 | Rohit | 7000 |
| 123 | Vivek | 1500 |
| 125 | Akshay | 7500 |

(10 rows, 4 ms)

The screenshot shows the same database client interface. The query is `select AVG(salary) from employee`. The results are shown in a table with a single column `AVG(SALARY)` and a single row containing the value `5300`. Below the table, it indicates `(1 row, 0 ms)`.

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

SQL statement: `select AVG(salary) from employee`

| AVG(SALARY) |
|-------------|
| 5300 |

(1 row, 0 ms)

5. Displaying the aggregate function results

The screenshot shows the same database client interface. The query is `select LT_AVG(salary) from employee;`. The results are shown in a table with a single column `LT_AVG(SALARY)` and a single row containing the value `3`. Below the table, it indicates `(1 row, 1 ms)`.

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

SQL statement: `select LT_AVG(salary) from employee;`

| LT_AVG(SALARY) |
|----------------|
| 3 |

(1 row, 1 ms)

jdbc:h2:~/test | ☒ Auto commit | Max rows: 1000 | Auto complete: Off | Auto select: On

Run Run Selected Auto complete Clear SQL statement:

select count(*) from employee where salary <= (select avg(salary) from employee);

select count(*) from employee where salary <= (select avg(salary) from employee);

COUNT(*)

3

(1 row, 1 ms)

jdbc:h2:~/test | ☒ Auto commit | Max rows: 1000 | Auto complete: Off | Auto select: On

Run Run Selected Auto complete Clear SQL statement:

select GT_AVG(salary) from employee;

select GT_AVG(salary) from employee;

GT_AVG(SALARY)

7

(1 row, 2 ms)

jdbc:h2:~/test | ☒ Auto commit | Max rows: 1000 | Auto complete: Off | Auto select: On

Run Run Selected Auto complete Clear SQL statement:

select count(*) from employee where salary > (select avg(salary) from employee);

select count(*) from employee where salary > (select avg(salary) from employee);

COUNT(*)

7

(1 row, 0 ms)

REFERENCES:

1. <https://github.com/h2database/h2database>
2. <https://github.com/h2database/h2database/blob/master/h2/src/main/org/h2/api/AggregateFunction.java>
3. <http://h2-database.66688.n3.nabble.com/user-defined-aggregate-function-FIRST-and-LAST-t4031866.html>