SJSU Students
CS157B-01

Hw 3 Experiments

Main sections: Summary, Purpose, Hypothesis, Procedure before Experiments, Experiments, Overall Results, and Overall Conclusion.

Summary

Here is the summary section of the overall paper because the paper is too long and might be difficult to navigate for grading purposes. We are using MySQL.
This summary section contains the following sections:

1. **Results** (includes the conclusion, overall plan check on all the joins, and table of all experiments' runtimes)
2. **How to create the input tables**
3. **Transcripts of JOIN Query and EXPLAIN operation for each join** (includes join algorithm determination and some write-up)
4. **Timing Tests for each Join w/o Index**
5. **Using Indices Effect**

To find the beginning of the more detailed or full report version, please look up "Purpose".

Results:

This is extracted from the "Overall Conclusion" and "Overall Results" sections. (Warning: this part is long)
Highlights:

- Without the indexes, all joins have almost the same runtime.
- Index on B-value alone makes the runtime much slower for 2 reasons: first, having an index eliminates the use of hash-join (building the hash table in memory if fit). Secondly, the size of the index entry in this case is as big as the size of the data record. Reading from the index table and then following the record pointer to load the data from the original table for C-value would add extra overhead.
- Out of all the indexes, the composite index manages to improve the query time because both B and C in the query conditions can be obtained directly from the index table.

My hypothesis is partially right. (Short version of hypothesis: I thought the runtime will be about the same. I also thought indexing would overall improve the runtime of the queries when used. For more detail, please look up the "Hypothesis" that is bolded and underlined)

Overall, based on the experiments, **the runtime among the three joins is almost the same.** The bushy tree is a little slower than the left-deep join tree and right-deep join tree, but not far off.

However, indexing does not always mean improving query time. It depends on the condition of the query as well as the attribute(s) being used in the index. For instance, indexing on B-value only will reduce the query performance compared to the query performance without any index. Meanwhile composite index on B and C helps improve the query performance.

Overall Plan Check:
Without indexes, the plan for each join query is the same. They all use where for one relation and then use where and the **hash-join** algorithm for the rest.

With indexes, the plan for each join query is also the same. The plan often tends to use **hash-join** for relations without indexes and **index-based join** for those with indexes.

Note: There is a possibility that the reason no nested loop was found throughout the experiments is that the computer's memory (16 GB 2133 MHz LPDDR3 Memory) is large enough to build a hash for the entire table.

Experimentation Result Tables:
Below are tables representing the average runtime (in seconds) for the joins based on the experiments.

Without Indexes:

Type of Join	Average w/o Index	Standard deviation w/o Indexes
Left-deep Join	14.64	0.214009
Bushy	14.67	0.515636
Right-deep Join	14.562	0.223553

With Indexes Applied on all 5 Relations:

Indexing on… →	B-value only		C-value only		B-value then C-value		Composite Index (B,C)	
Type of Join	AVG	SD	AVG	SD	AVG	SD	AVG	SD
Left-deep	82.878	0.690895	14.508	0.244336	45.714	1.945133	7.596	0.287305
Bushy	86.272	4.487273	14.918	0.505149	45.706	1.796804	8.04	0.560393
Right-deep	91.666	4.880986	15.718	0.455956	46.146	0.528870	7.814	0.292

With Indexes Applied on 2 of 5 Relations:

Indexing on… →	B-value only		C-value only		B-value then C-value		Composite Index (B,C)	
Type of Join	AVG	SD	AVG	SD	AVG	SD	AVG	SD
Left-deep	15.794	0.645804	13.882	0.692168	44.58	3.279134	8.424	0.863101
Bushy	15.596	0.547087	13.684	0.860153	47.186	1.701686	8.238	0.777725
Right-deep	15.3	0.283267	14.376	1.130568	47.13	1.378826	7.742	0.256312

How to create the input tables:

For a clear and detailed step-by-step procedure with pictures, please look (or control/command f) for the "Procedure before Experiments" section.

Generate 5 data files:

```
java DataGenerator 0 1000 5 10 R1data.txt
java DataGenerator 1000 1000 5 10 R2data.txt
java DataGenerator 2000 1000 5 10 R3data.txt
java DataGenerator 3000 1000 10 5 R4data.txt
java DataGenerator 4000 1000 10 5 R5data.txt
```

Create the 5 tables in MySQL database: R1(A,B,C), R2(D,B,C), R3(E,B,C), R4(F,B,C), R5(G,B,C) Queries:

```
CREATE TABLE R1 (A INTEGER, B INTEGER, C INTEGER);
CREATE TABLE R2 (D INTEGER, B INTEGER, C INTEGER);
CREATE TABLE R3 (E INTEGER, B INTEGER, C INTEGER);
CREATE TABLE R4 (F INTEGER, B INTEGER, C INTEGER);
CREATE TABLE R5 (G INTEGER, B INTEGER, C INTEGER);
```

Bulk Load Queries:

```
SHOW GLOBAL VARIABLES LIKE 'local_infile';
SET GLOBAL local_infile=1;
//exit mysql and enter again with:
mysql --local_infile=1 -u <your username here> -p <your password here>

LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R1data.txt' INTO TABLE R1
FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';
```

```
LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R2data.txt' INTO TABLE R2
FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';
LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R3data.txt' INTO TABLE R3
FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';
LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R4data.txt' INTO TABLE R4
FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';
LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R5data.txt' INTO TABLE R5
FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';
```

Transcripts of JOIN Query and EXPLAIN operation for each join:

- **Left-deep:**
  SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

  EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND
 R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | R1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                  |
|  1 | SIMPLE      | R2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
5 rows in set, 1 warning (0.01 sec)
```

  // Determination of algorithm:
  From the output of EXPLAIN, we can see in the “type” column that the DBMS marks it “ALL.” We can also see in the “Extra” column that the DBMS just uses where in R1 and then uses where and **hash-join algorithm** for the rest.
  The “ALL” means that MySQL scans the whole table using no indexes to access/join the table, so the DBMS access and scans through R1 filtering with the where condition. It then accesses R2 with the join buffer and so forth.

- **Bushy Tree:**
  SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B

AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

```
mysql>
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
(R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.
B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | R1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                  |
|  1 | SIMPLE      | R2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

// Determination of algorithm:
From the output of EXPLAIN, we can see in the “type” column that the DBMS marks it “ALL.” We can also see in the “Extra” column that the DBMS just uses where in R1 and then uses where and **hash-join algorithm** for the rest.
The “ALL” means that MySQL scans the whole table using no indexes to access/join the table, so the DBMS access and scans through R1 filtering with the where condition. It then accesses R2 with the join buffer and so forth.

This is the same as the left-deep join tree without indexes.

- **Right-deep:**
  SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

  EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C)

ON R1.B = R2.B AND R1.C = R2.C);

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
 (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON
R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+---------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+---------------------------------------------+
|  1 | SIMPLE      | R1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                 |
|  1 | SIMPLE      | R2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+---------------------------------------------+
5 rows in set, 1 warning (0.00 sec)

mysql>
```

//Determination of algorithm:

From the output of EXPLAIN, we can see in the “type” column that the DBMS marks it “ALL.” We can also see in the “Extra” column that the DBMS just uses where in R1 and then uses where and **hash-join algorithm** for the rest.

The “ALL” means that MySQL scans the whole table using no indexes to access/join the table, so the DBMS access and scans through R1 filtering with the where condition. It then accesses R2 with the join buffer and so forth.

This is the same plan as the left-deep join tree and the bushy tree join without any index.

<u>Timing Tests for each Join w/o Index:</u>

(For a more detailed look, please look up “Execute Query 5 times. Compute the Average and Standard Deviation Result:”. There should be 4 search results including this section. The second search result is for Left-deep. The third search result is Bushy tree and the fourth search result is Right-deep.)

We are going to execute each join query 5 times. Then, we will compute the average and standard deviation for each.

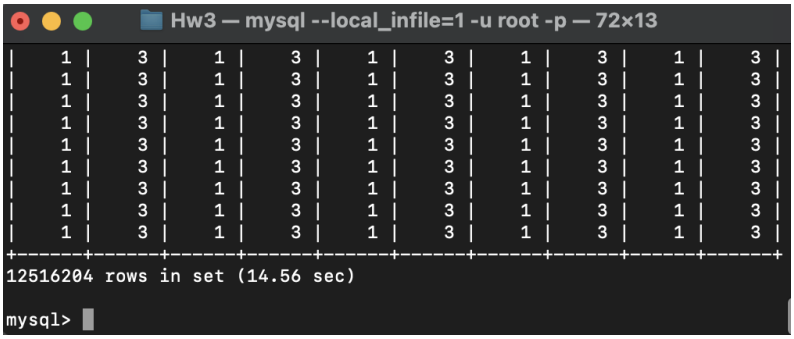
The 10 columns shown after the execution are as follows: R1B, R1C, R2B, R2C, R3B, R3C, R4B, R4C, R5B, R5C.

- **Left-deep**
    1. Execution 1 Time: **14.30 seconds**

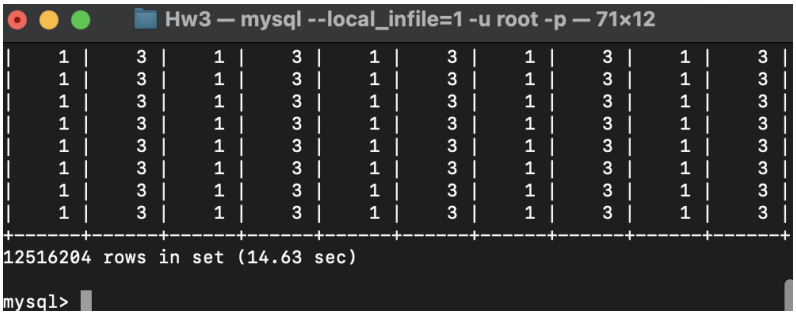
```
Hw3 — mysql --local_infile=1 -u root -p — 74×13
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (14.30 sec)

mysql>
```

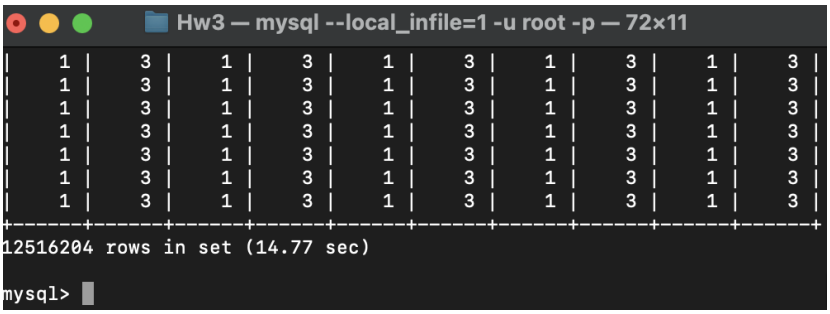
2. Execution 2 Time: **14.56 seconds**



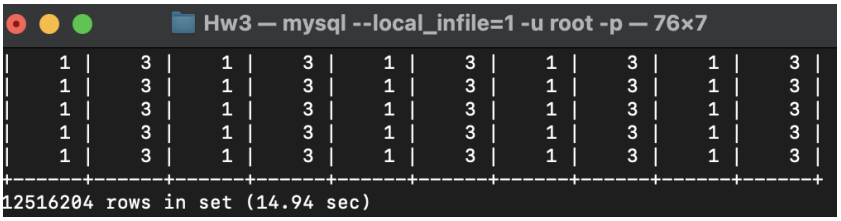
3. Execution 3 Time: **14.63 seconds**



4. Execution 4 Time: **14.77 seconds**



5. Execution 5 Time: **14.94 seconds**



At this point, we have the 5 times from the 5 executions of the join query. We will compute the average and standard deviation.

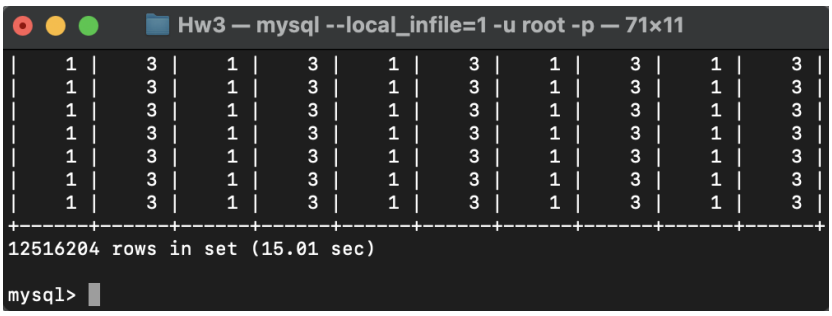
Average = (14.30 + 14.56 + 14.63 + 14.77 + 14.94) / 5 = **14.64 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

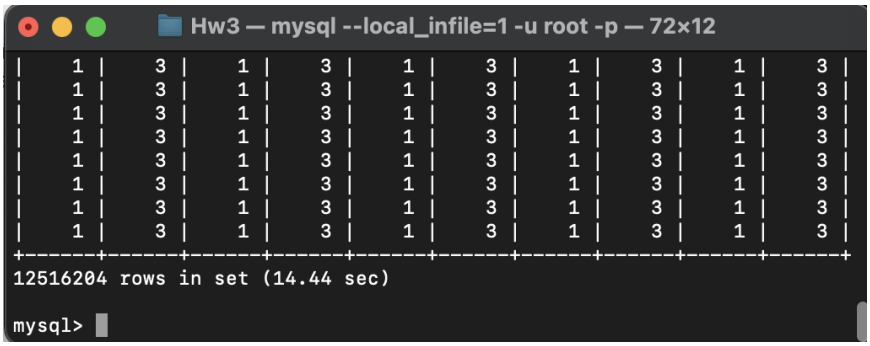
$= \sqrt{\frac{\mid 14.30 - 14.64 \mid^2 + \mid 14.56 - 14.64 \mid^2 + \mid 14.63 - 14.64 \mid^2 + \mid 14.77 - 14.64 \mid^2 + \mid 14.94 - 14.64 \mid^2}{5}}$

$$= \sqrt{\frac{0.1156 + 0.0064 + 0.0001 + 0.0169 + 0.09}{5}} = \sqrt{\frac{0.229}{5}} = \mathbf{0.214009}$$

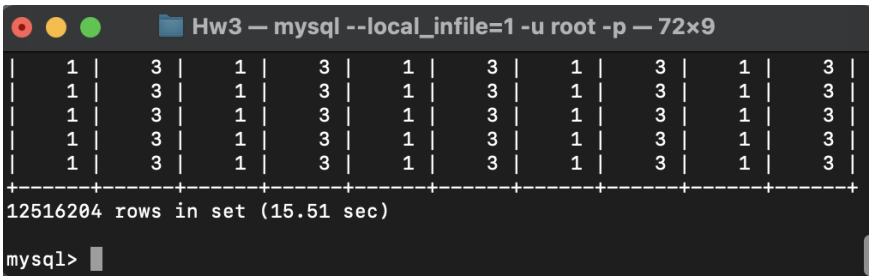
- **Bushy tree**
  1. Execution Time 1: **15.01 seconds**



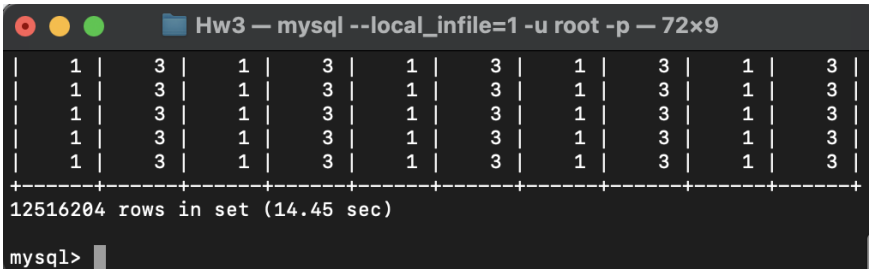
  2. Execution Time 2: **14.44 seconds**



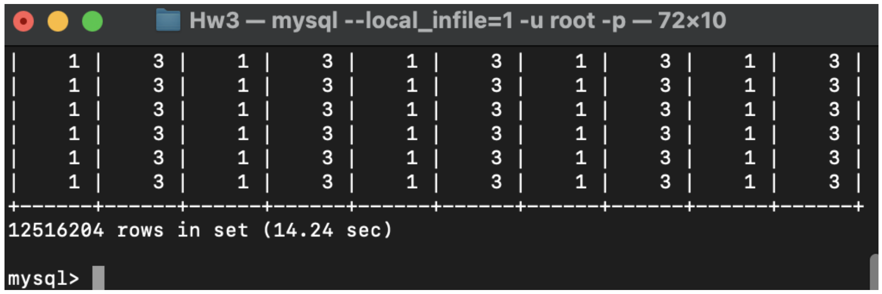
  3. Execution Time 3: **15.51 seconds**



  4. Execution Time 4: **14.15 seconds**



5. Execution Time 5: **14.24 seconds**



At this point, we have the 5 times from the 5 executions of the join query. We will compute the average and standard deviation.

Average = (15.01 + 14.44 + 15.51 + 14.15 + 14.24) / 5= **14.67 seconds**

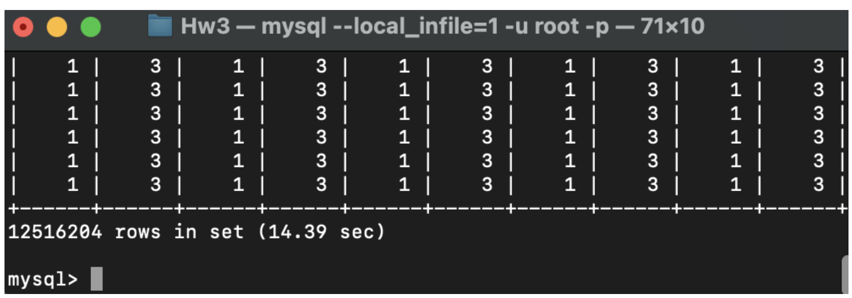
Standard deviation $= \sqrt{\frac{\Sigma\,|\,x - \mu\,|^2}{N}}$

$$= \sqrt{\frac{|\,15.01 - 14.67\,|^2 + |\,14.44 - 14.67\,|^2 + |\,15.51 - 14.67\,|^2 + |\,14.15 - 14.67\,|^2 + |\,14.24 - 14.67\,|^2}{5}}$$

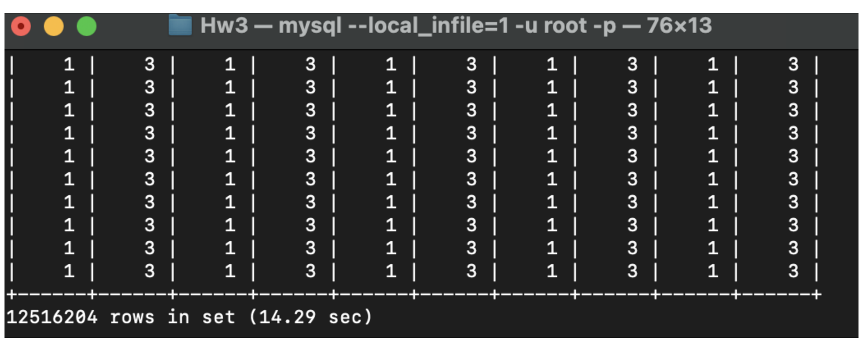
$$= \sqrt{\frac{0.1156 + 0.0529 + 0.7056 + 0.2704 + 0.1849}{5}} = \sqrt{\frac{1.3294}{5}} = \mathbf{0.515636}$$

- **Right-deep**

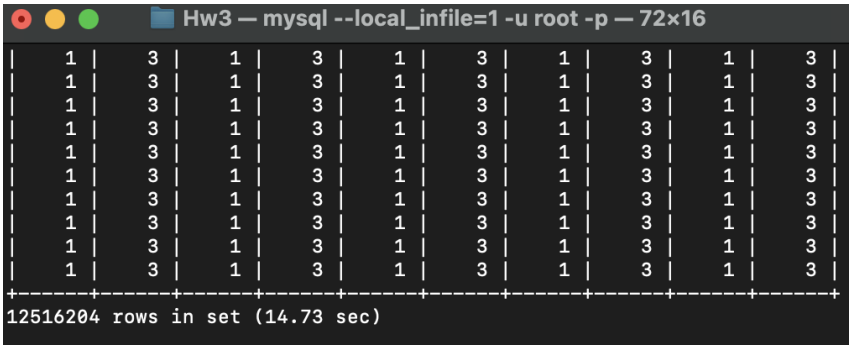
1. Execution 1 Time: **14.39 seconds**



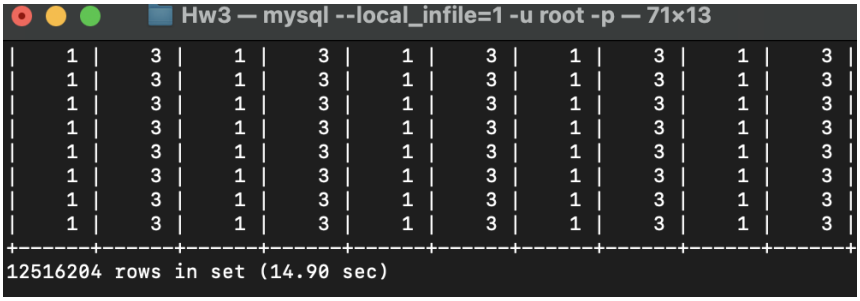
2. Execution 2 Time: **14.29 seconds**



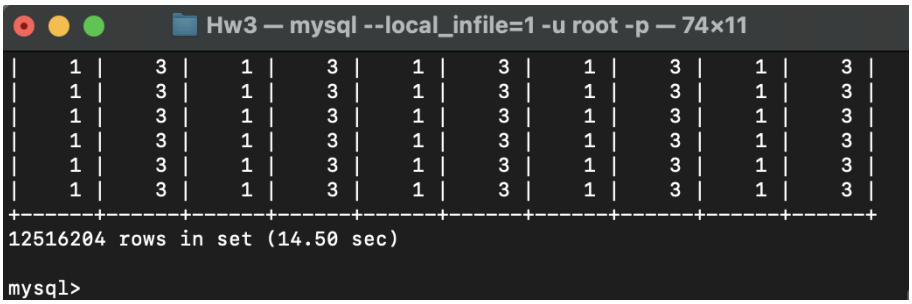
3. Execution 3 Time: **14.73 seconds**



4. Execution 4 Time: **14.90 seconds**



5. Execution 5 Time: **14.50 seconds**



At this point, we have the 5 times from the 5 executions of the join query. We will compute the average and standard deviation.

Average = (14.39 + 14.29 + 14.73 + 14.90 + 14.50) / 5 = **14.562 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x-\mu \mid^2}{N}}$

=

$$\sqrt{\frac{\mid 14.39-14.562 \mid^2+\mid 14.29\ -14.562 \mid^2+\mid 14.73-14.562 \mid^2+\mid 14.90-14.562 \mid^2+\mid 14.50-14.562 \mid^2}{5}}$$

$$=\sqrt{\frac{0.029584+0.073984+0.028224+0.114244+0.003844}{5}}=\sqrt{\frac{1.05908}{5}}=\mathbf{0.223553}$$

Using Indices Effect:

We will test whether using indexes will improve our timing in these join trees.

We will create indexes for attributes that are part of "JOIN ON" conditions. In this case, that will be the attributes B and C.

There are 4 index experiments to try on the 5 relations:

1. Create index on B-value only
2. Create index on C-value only
3. Create index on B. Then create index on C
4. Create a composite index (B, C)

To view the executions and calculations done for each case for each join query, please look up "**Effect of Indices**" where the 2nd search result is for Left-deep, 3rd search result is for Bushy tree, and 4th search result is for Right-deep.

The effect results can be seen in the tables at the top of the paper under "Experimentation Result Tables."

Overall, adding indexes to relations does not always help the query run faster. Adding the correct indexes (e.g. composite index of B and C) would help the query while indexes on B or C alone may have a negative impact on query time. As seen in the B-value index-only section, that attribute only made time slower. Having an index sometimes eliminates the use of hash-join which may slow things down. In this case, table-scan with hash-join would perform better.

Without an index, DBMS just uses hash-join.

==========
==========

Purpose

To determine if any join operation (left-tree, right-tree, and bushy tree) is faster than the other and if putting indexes improves the timing.

Hypothesis

Among the left-deep join tree, right-deep join tree, and bushy join tree, I think they are all about the same speed. If there were any differences, they should be minimal. Indexing should greatly improve timing.

We will do some experiments with 5 tables and the three join operations.

The laptop used for this has 16 GB 2133 MHz LPDDR3 Memory.

Procedure before Experiments

In order to do the experiments, we must have 5 tables (R1, R2, R3, R4, R5) to use. Each table will need generated data. We will be using MySQL to create the tables.

1. Use DataGenerator.java to generate 5 text files of 3 columns of data.

```
(base) Mints-MacBook-Pro:Hw3 Chicken$ java DataGenerator 0 1000 5 10 R1data.txt
Finished writing generated data into fileName R1data.txt
(base) Mints-MacBook-Pro:Hw3 Chicken$ java DataGenerator 1000 1000 5 10 R2data.txt
Finished writing generated data into fileName R2data.txt
(base) Mints-MacBook-Pro:Hw3 Chicken$ java DataGenerator 2000 1000 5 10 R3data.txt
Finished writing generated data into fileName R3data.txt
(base) Mints-MacBook-Pro:Hw3 Chicken$ java DataGenerator 3000 1000 10 5 R4data.txt
Finished writing generated data into fileName R4data.txt
(base) Mints-MacBook-Pro:Hw3 Chicken$ java DataGenerator 4000 1000 10 5 R5data.txt
Finished writing generated data into fileName R5data.txt
(base) Mints-MacBook-Pro:Hw3 Chicken$
```

   Query used:
   java DataGenerator 0 1000 5 10 R1data.txt
   java DataGenerator 1000 1000 5 10 R2data.txt
   java DataGenerator 2000 1000 5 10 R3data.txt
   java DataGenerator 3000 1000 10 5 R4data.txt
   java DataGenerator 4000 1000 10 5 R5data.txt

2. Create the 5 tables in MySQL database:
   R1(A,B,C), R2(D,B,C), R3(E,B,C), R4(F,B,C), R5(G,B,C)

```
mysql>
mysql> CREATE TABLE R1 (A INTEGER, B INTEGER, C INTEGER);
Query OK, 0 rows affected (0.04 sec)

mysql> CREATE TABLE R2 (D INTEGER, B INTEGER, C INTEGER);
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE R3 (E INTEGER, B INTEGER, C INTEGER);
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE R4 (F INTEGER, B INTEGER, C INTEGER);
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE R5 (G INTEGER, B INTEGER, C INTEGER);
Query OK, 0 rows affected (0.00 sec)

mysql>
```

   Query used:
   CREATE TABLE R1 (A INTEGER, B INTEGER, C INTEGER);
   CREATE TABLE R2 (D INTEGER, B INTEGER, C INTEGER);
   CREATE TABLE R3 (E INTEGER, B INTEGER, C INTEGER);

CREATE TABLE R4 (F INTEGER, B INTEGER, C INTEGER);
CREATE TABLE R5 (G INTEGER, B INTEGER, C INTEGER);

Outcome:

```
mysql> show tables;
+----------------------+
| Tables_in_cs157b_hw3 |
+----------------------+
| R1                   |
| R2                   |
| R3                   |
| R4                   |
| R5                   |
+----------------------+
5 rows in set (0.00 sec)
```

3. **Load the text files into the corresponding table. (Bulk Load)**

To do that in MySQL, first, make sure the global variable 'local_infile' is "ON" not "OFF."

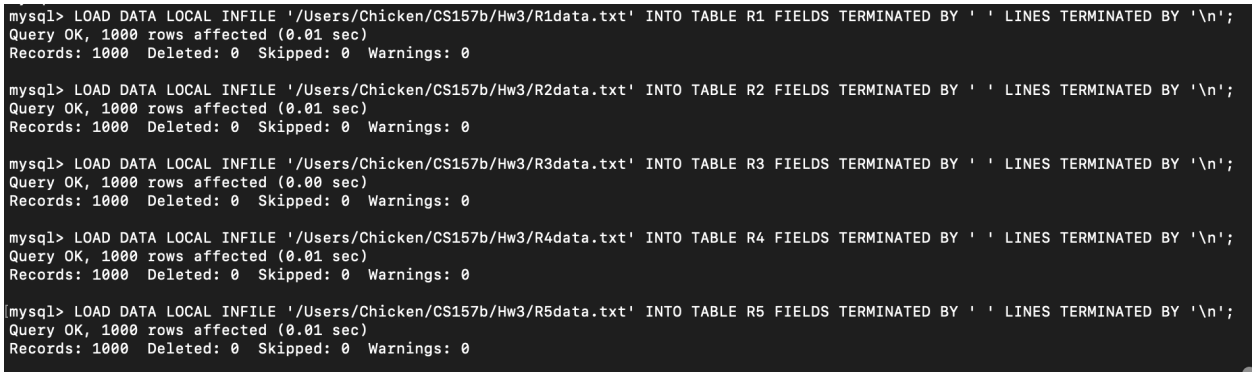
```
mysql> show global variables like 'local_infile';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| local_infile  | ON    |
+---------------+-------+
1 row in set (0.00 sec)
```

Query used:
show global variables like 'local_infile';

To turn it on, set the global variable to 1 with the following query in MySQL:
SET GLOBAL local_infile=1;

After that, exit MySQL. Open MySQL again with "--local_infile=1" after the keyword "mysql."
Query Used:
mysql --local_infile=1 -u <your username here> -p <your password here>

Once completed, execute the "load data" queries as seen below:



Query used:
LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R1data.txt' INTO TABLE R1 FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';

LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R2data.txt' INTO TABLE R2 FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';

LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R3data.txt' INTO TABLE R3 FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';

LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R4data.txt' INTO TABLE R4 FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';

LOAD DATA LOCAL INFILE '/Users/Chicken/CS157b/Hw3/R5data.txt' INTO TABLE R5 FIELDS TERMINATED BY ' ' LINES TERMINATED BY '\n';

Now, we have 5 tables with data in them. It is time to proceed to Experiments.

Experiments

With the 5 tables, we will be considering a join with the following condition:
R1.B=R2.B and R1.C=R2.C and R2.B=R3.B and R2.C=R3.C and R3.B=R4.B and R3.C=R4.C and R4.B=R5.B and R4.C=R5.C and R5.B=1 followed by a projection onto the columns B, C.

We will be doing an experiment for each of the 3 types of joins: left-deep join tree, bushy tree, and right-deep join tree. For each, we will:

- Express the join with the condition above as a SQL query
- Use DBMS “EXPLAIN” statement with the query and determine which join algorithms were used
- Execute the join query 5 times. Then, compute the average and standard deviation
- Test and compare the times when using indexes

- A brief conclusion/observation

==========

Experiment 1: Left-Deep Join Tree

Join SQL Query:

SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

Using MySQL DBMS "EXPLAIN" statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND
 R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | R1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                  |
|  1 | SIMPLE      | R2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
5 rows in set, 1 warning (0.01 sec)
```

Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

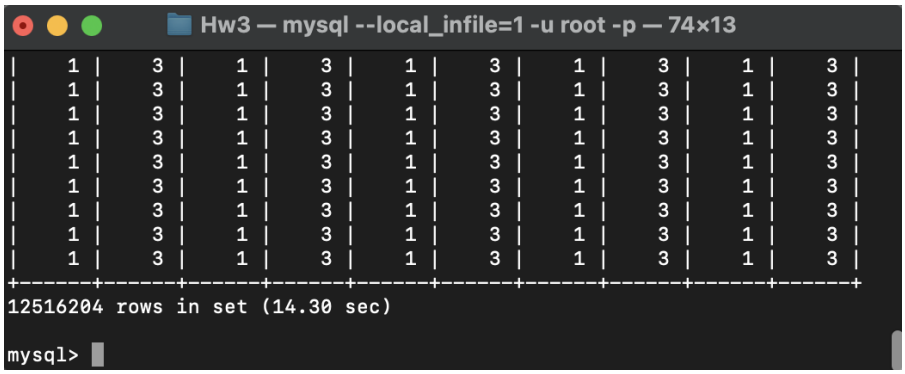
From the output of EXPLAIN, we can see in the "type" column that the DBMS marks it "ALL." We can also see in the "Extra" column that the DBMS just uses where in R1 and then uses where and **hash-join algorithm** for the rest.

The "ALL" means that MySQL scans the whole table using no indexes to access/join the table, so the DBMS access and scans through R1 filtering with the where condition. It then accesses R2 with the join buffer and so forth.

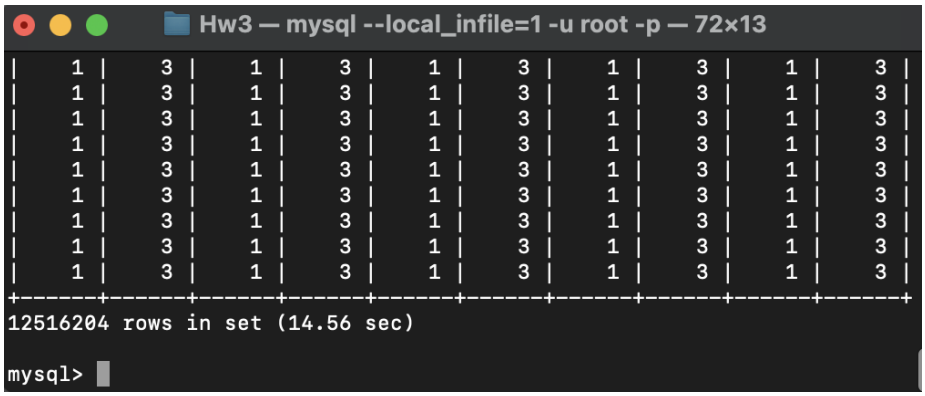
Execute Query 5 times. Compute the Average and Standard Deviation Result:

The 10 columns shown after the execution are as follows: R1B, R1C, R2B, R2C, R3B, R3C, R4B, R4C, R5B, R5C.

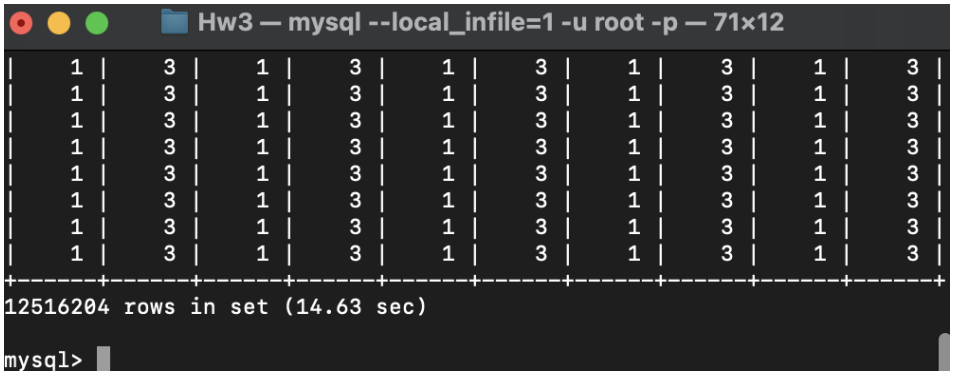
1. Execution 1 Time: **14.30 seconds**



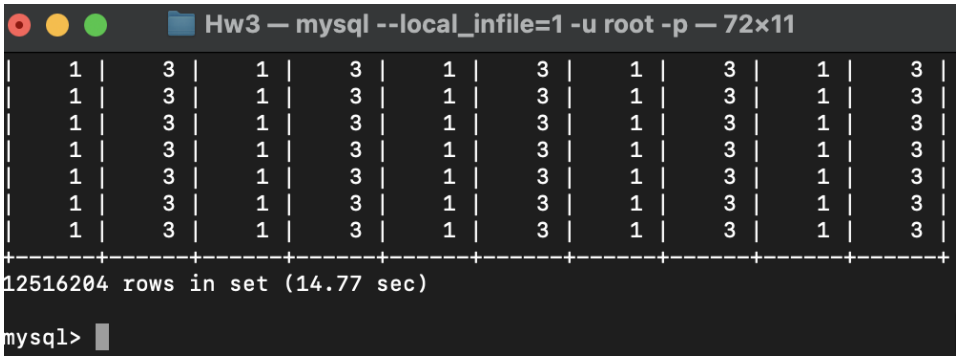
2. Execution 2 Time: **14.56 seconds**



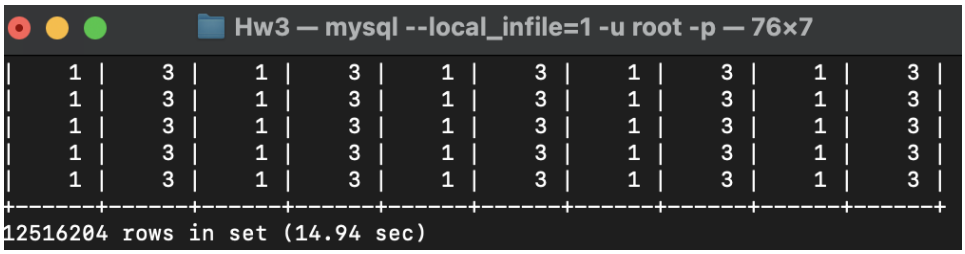
3. Execution 3 Time: **14.63 seconds**



4. Execution 4 Time: **14.77 seconds**



5. Execution 5 Time: **14.94 seconds**



At this point, we have the 5 times from the 5 executions of the join query. We will compute the average and standard deviation.

Average = (14.30 + 14.56 + 14.63 + 14.77 + 14.94) / 5 = **14.64 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x-\mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 14.30-14.64 \mid^2 + \mid 14.56-14.64 \mid^2 + \mid 14.63-14.64 \mid^2 + \mid 14.77-14.64 \mid^2 + \mid 14.94-14.64 \mid^2}{5}}$$

$$= \sqrt{\frac{0.1156+0.0064+0.0001+0.0169+0.09}{5}} = \sqrt{\frac{0.229}{5}} = \mathbf{0.214009}$$

Effect of Indices: Use Indexes and Compare:

We will test whether using indexes will improve our timing in left-deep join tree.
We will create indexes for attributes that are part of "JOIN ON" conditions. In this case, that will be the attributes B and C.

There are 4 index experiments to try on the 5 relations:

1. Create index on B-value only
2. Create index on C-value only
3. Create index on B. Then create index on C
4. Create a composite index (B, C)

1. **Experiment 1.1: Index on B-value only (no index on C-value)**

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R3 ON R3 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R4 ON R4 (B);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R5 ON R5 (B);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);
CREATE INDEX index_B_R3 ON R3 (B);
CREATE INDEX index_B_R4 ON R4 (B);
CREATE INDEX index_B_R5 ON R5 (B);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B
AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B
= R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key        | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_B_R5    | index_B_R5 | 5       | const |   88 |   100.00 | NULL        |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_B_R4    | index_B_R4 | 5       | const |  118 |    10.00 | Using where |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_B_R3    | index_B_R3 | 5       | const |  174 |    10.00 | Using where |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2    | index_B_R2 | 5       | const |  208 |    10.00 | Using where |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1    | index_B_R1 | 5       | const |  209 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-------------+
5 rows in set, 1 warning (0.01 sec)
```

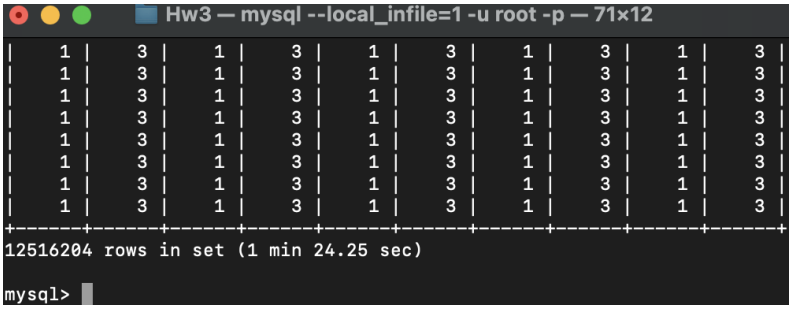
Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

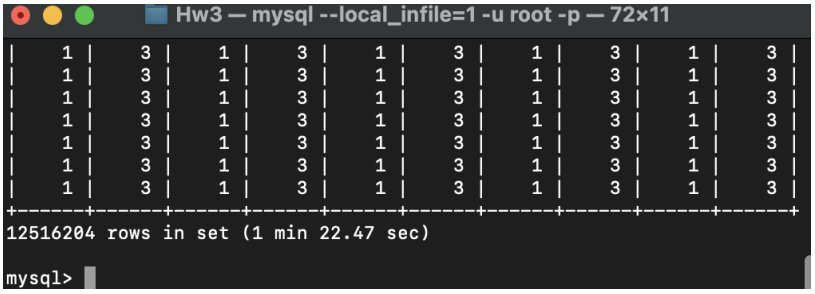
From the output of EXPLAIN, we can see the index we just made under “possible_keys.” Under “type,” it says “ref” and under the “ref” column it says “const” because our query condition B = 1 is indeed a constant. We can see the “key” column actually being used. We can also see in the “Extra” column that the DBMS uses the index to find B-value in R5 and then uses where for the rest. DBMS seems to be using an **index-join algorithm.**

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

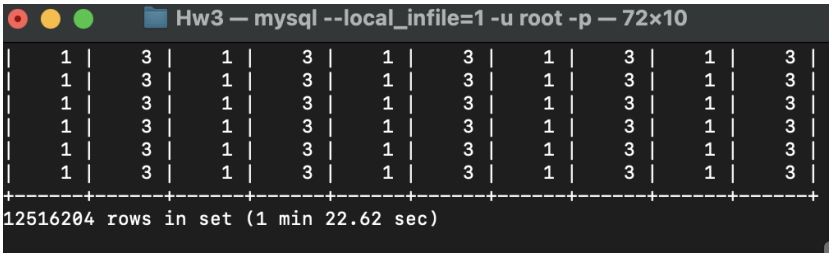
1. Execution Time 1: **1 minute and 24.25 seconds (84.25 seconds)**



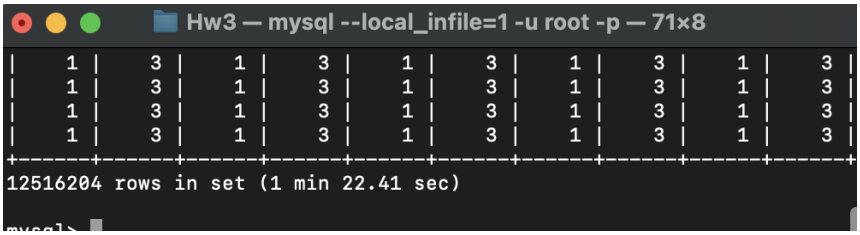
2. Execution Time 2: **1 minute and 22.47 seconds (82.47 seconds)**



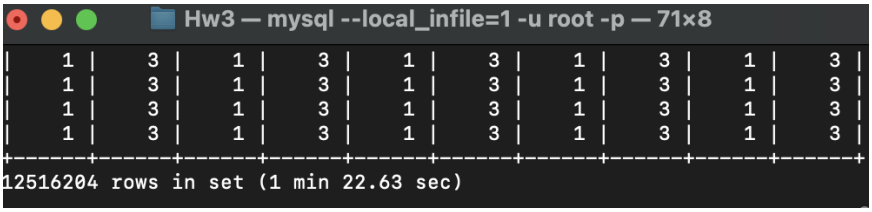
3. Execution Time 3: **1 minute and 22.62 seconds (82.62 seconds)**



4. Execution Time 4: **1 minute and 22.42 seconds (82.42 seconds)**



5. Execution Time 5: **1 minute and 22.63 seconds (82.63 seconds)**



Average = (84.25 + 82.47 + 82.62 + 82.42 + 82.63) / 5 = **82.878 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 84.25 - 82.878 \mid^2 + \mid 82.47 - 82.878 \mid^2 + \mid 82.62 - 82.878 \mid^2 + \mid 82.42 - 82.878 \mid^2 + \mid 82.63 - 82.878 \mid^2}{5}}$$

$$= \sqrt{\frac{1.882384 + 0.166464 + 0.066564 + 0.209764 + 0.061504}{5}} = \sqrt{\frac{2.38668}{5}} = \mathbf{0.690895}$$

Case 2: Create the index on 2 relations only

Create Index:

```
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

CREATE INDEX index_B_R1 ON R1 (B);

CREATE INDEX index_B_R2 ON R2 (B);

Using DBMS EXPLAIN statement:

```
mysql>
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1
 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C
 AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key        | key_len | ref   | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2    | index_B_R2 | 5       | const |  208 |   100.00 | NULL                                                |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1    | index_B_R1 | 5       | const |  209 |    10.00 | Using where                                         |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL  | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL  | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL  | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:

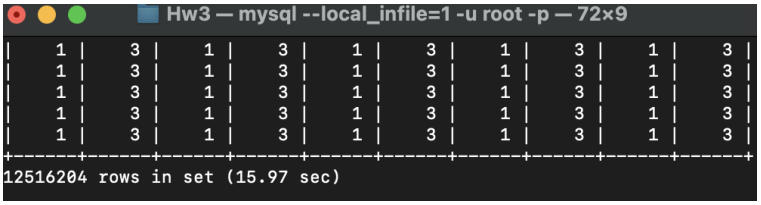
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND

R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

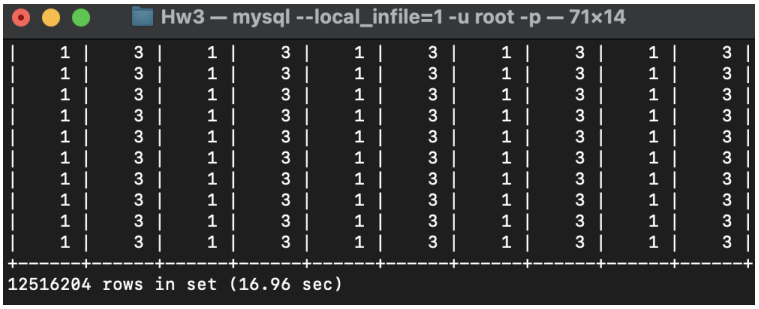
From the output of EXPLAIN, we can see the index we just made under “possible_keys.” Under “type,” it says “ref” for and under the “ref” column it says “const” for the relations with the index, but those without the index have type “ALL” and ref “NULL”. We can also see in the “Extra” column that the DBMS uses the index to find B-value in R1 and where for R2. DBMS seems to use where and **index-join algorithm** for R1 and R2, but uses where and **hash-join algorithm** for those who do not have an index.

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

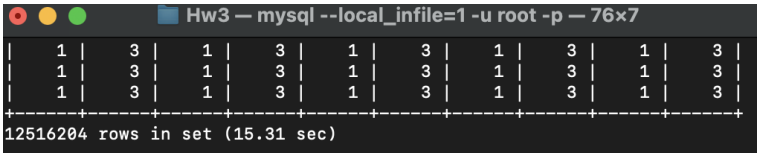
1. Execution Time 1: **15.97 seconds**



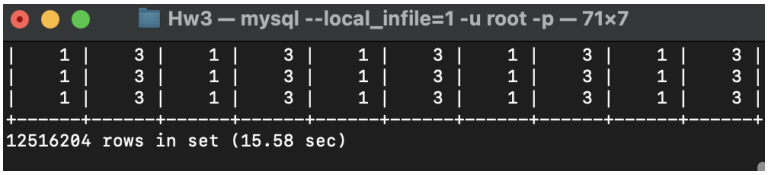
2. Execution Time 2: **16.96 seconds**



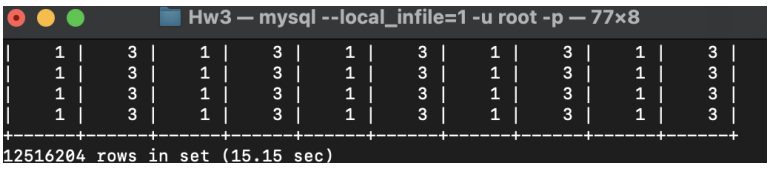
3. Execution Time 3: **15.31 seconds**



4. Execution Time 4: **15.58 seconds**



5. Execution Time 5: **15.15 seconds**



Average = (15.97 + 16.96 + 15.31 + 15.58 + 15.15) / 5 = **15.794 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 15.97 - 15.794 \mid^2 + \mid 16.96 - 15.794 \mid^2 + \mid 15.31 - 15.794 \mid^2 + \mid 15.58 - 15.794 \mid^2 + \mid 15.15 - 15.794 \mid^2}{5}}$$

$$= \sqrt{\frac{0.030976 + 1.359556 + 0.234256 + 0.045796 + 0.414736}{5}} = \sqrt{\frac{2.08532}{5}} = \mathbf{0.645804}$$

2. Experiment 1.2: Index on C-value only

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R3 ON R3 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R4 ON R4 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R5 ON R5 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);
CREATE INDEX index_C_R3 ON R3 (C);
CREATE INDEX index_C_R4 ON R4 (C);
CREATE INDEX index_C_R5 ON R5 (C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
  ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B A
ND R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                   |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------+
|  1 | SIMPLE      | R4    | NULL       | ALL  | index_C_R4    | NULL | NULL    | NULL | 1000 |    10.00 | Using where                             |
|  1 | SIMPLE      | R1    | NULL       | ALL  | index_C_R1    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R2    | NULL       | ALL  | index_C_R2    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R3    | NULL       | ALL  | index_C_R3    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R5    | NULL       | ALL  | index_C_R5    | NULL | NULL    | NULL | 1000 |     2.00 | Using where; Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

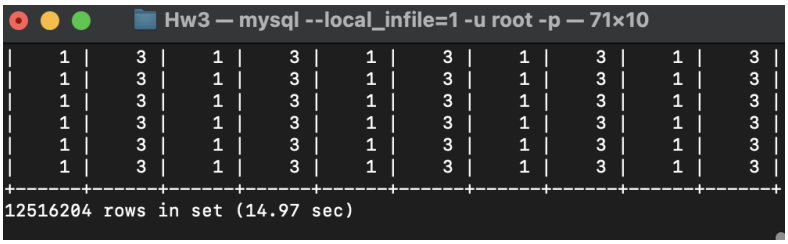
Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

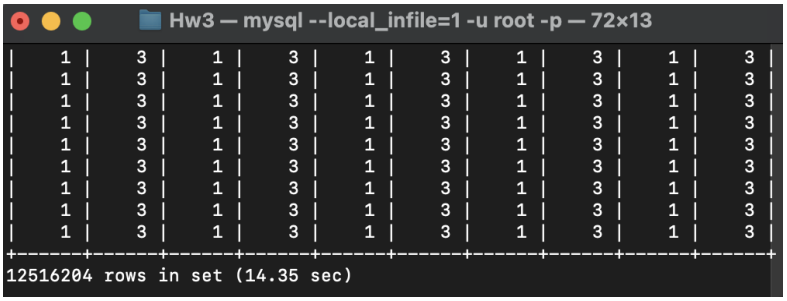
From the output of EXPLAIN, we can see the index we just made under "possible_keys." Under "type," it says "ALL". It is also clear it is not using this index because everything in the "key" column is NULL. We can also see in the "Extra" column that the DBMS uses where for the first relation R4 (not R1 or R5). Then, it uses where and hash-join for the rest. The order the DBMS access the relations is somewhat different too. DBMS is using a **hash-join algorithm.**

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

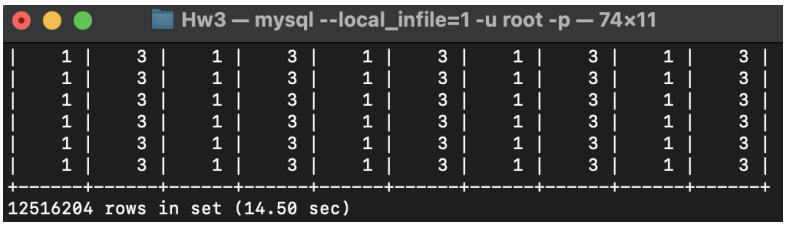
1. Execution Time 1: **14.97 seconds**



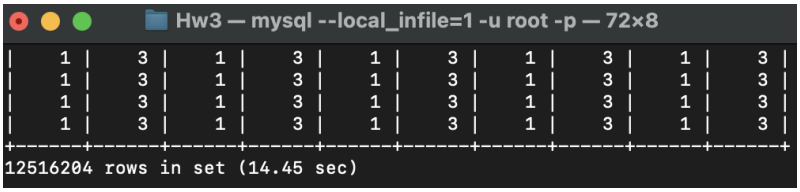
2. Execution Time 2: **14.35 seconds**



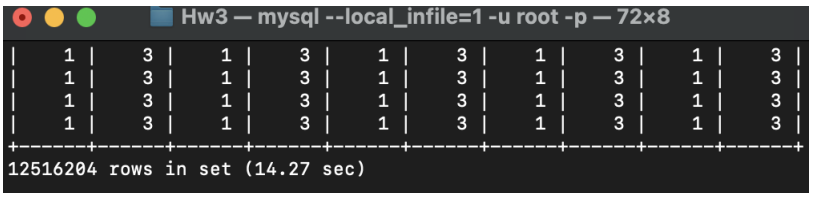
3. Execution Time 3: **14.50 seconds**



4. Execution Time 4: **14.45 seconds**



5. Execution Time 5: **14.27 seconds**



Average = (14.97 + 14.35 + 14.50 + 14.45 + 14.27) / 5 = **14.508 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 14.97 - 14.508 \mid^2 + \mid 14.35 - 14.508 \mid^2 + \mid 14.50 - 14.508 \mid^2 + \mid 14.45 - 14.508 \mid^2 + \mid 14.27 - 14.508 \mid^2}{5}}$$

$$= \sqrt{\frac{0.213444 + 0.024964 + 0.000064 + 0.003364 + 0.0566644}{5}} = \sqrt{\frac{0.2985004}{5}} = \mathbf{0.244336}$$

Case 2: Create the index on 2 relations only

Create Index:

```
mysql>
mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FRO
M ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B
 AND R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                         |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R1    | NULL       | ALL  | index_C_R1    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R2    | NULL       | ALL  | index_C_R2    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

From the output of EXPLAIN, we can see the 2 indexes we made under “possible_keys.” Under “type,” it says “ALL”. Again, it is also clear it is not using these indexes because everything in the “key” column is NULL. We can also see in the “Extra” column that the DBMS uses where for the first relation R3 (not R4). Then use where and hash-join for the rest. There is a different order of access too. DBMS is still using a **hash-join algorithm** regardless of our index.

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

1. Execution 1 Time: **14.39 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 72×8
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (14.39 sec)
```

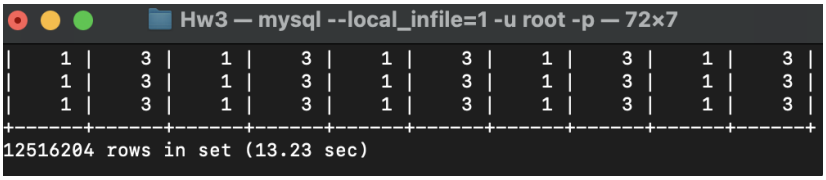
2. Execution 2 Time: **14.99 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 72×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (14.99 sec)
```

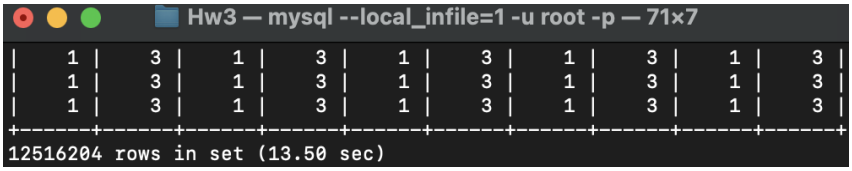
3. Execution 3 Time: **13.30 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 72×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (13.30 sec)
```

4. Execution 4 Time: 13.23 seconds



5. Execution 5 Time: 13.50



Average = (14.39 + 14.99 + 13.30 + 13.23 + 13.50) / 5 = **13.882 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$= \sqrt{\frac{\mid 14.39 - 13.882 \mid^2 + \mid 14.99 - 13.882 \mid^2 + \mid 13.30 - 13.882 \mid^2 + \mid 13.23 - 13.882 \mid^2 + \mid 13.50 - 13.882 \mid^2}{5}}$

$= \sqrt{\frac{0.258064 + 1.227664 + 0.338724 + 0.425104 + 0.145924}{5}} = \sqrt{\frac{2.39548}{5}} =$ **0.692168**

Observe how indexing on two relations with C-value is faster than indexing on all 5 relations with C-value. This is also faster than performing the join with no indexes.

3. Experiment 1.3: Index on B. Then index on C.

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_B_R3 ON R3 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R4 ON R4 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R5 ON R5 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R3 ON R3 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R4 ON R4 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R5 ON R5 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

```
CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);
CREATE INDEX index_B_R3 ON R3 (B);
CREATE INDEX index_B_R4 ON R4 (B);
CREATE INDEX index_B_R5 ON R5 (B);

CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);
CREATE INDEX index_C_R3 ON R3 (C);
CREATE INDEX index_C_R4 ON R4 (C);
CREATE INDEX index_C_R5 ON R5 (C);
```

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R
5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON
 R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------------+------------+---------+-----------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys       | key        | key_len | ref             | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------------+------------+---------+-----------------+------+----------+-------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_B_R5,index_C_R5 | index_B_R5 | 5       | const           |   88 |   100.00 | Using where |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_B_R3,index_C_R3 | index_C_R3 | 5       | cs157b_hw3.R5.C |  100 |    17.40 | Using where |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2,index_C_R2 | index_C_R2 | 5       | cs157b_hw3.R5.C |  100 |    20.80 | Using where |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_B_R4,index_C_R4 | index_B_R4 | 5       | const           |  118 |    20.00 | Using where |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1,index_C_R1 | index_C_R1 | 5       | cs157b_hw3.R5.C |  100 |    20.90 | Using where |
+----+-------------+-------+------------+------+---------------------+------------+---------+-----------------+------+----------+-------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:

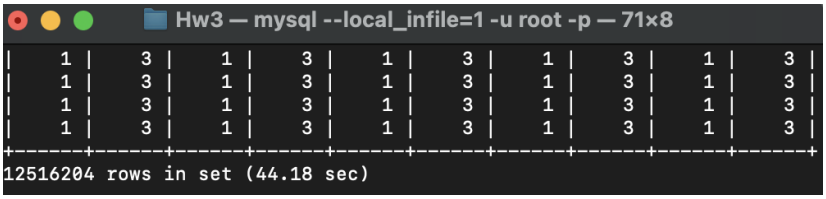
```
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS
R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
```

((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

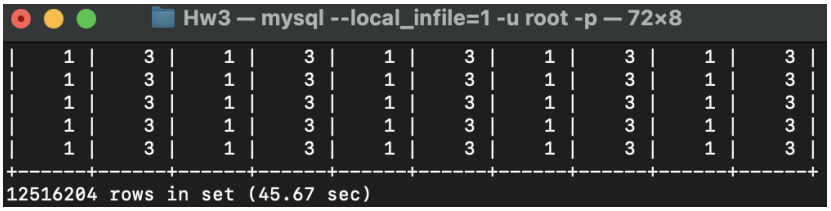
From the output of EXPLAIN, we can see the index we just made under “possible_keys.” Under “type,” it says “ref”. In the “key” column, DBMS is using the indexes we put out and using it at certain columns. We can also see in the “Extra” column that the DBMS uses where for all the relations. DBMS is using an **index-join algorithm** in this case.

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

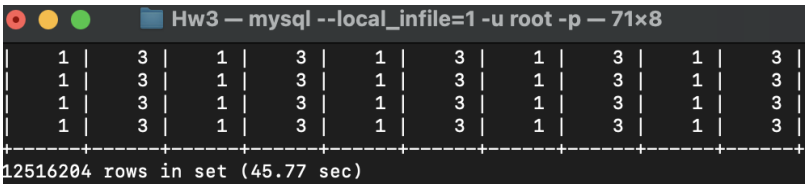
1. Execution Time 1: **44.18 seconds**



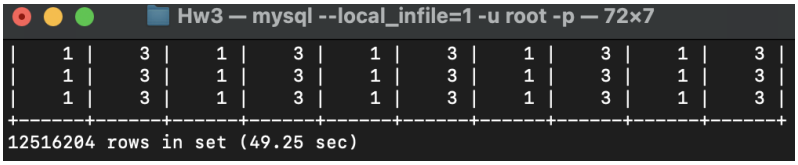
2. Execution Time 2: **45.67 seconds**



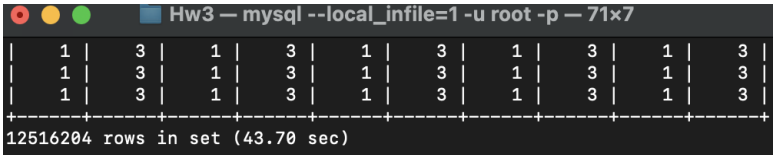
3. Execution Time 3: **45.77 seconds**



4. Execution Time 4: **49.25 seconds**



5. Execution Time 5: **43.70 seconds**



Average = (44.18 + 45.67 + 45.77 + 49.25 + 43.70) / 5 = **45.714 seconds**

Standard deviation $= \sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 44.18 - 45.714 \mid^2 + \mid 45.67 - 45.714 \mid^2 + \mid 45.77 - 45.714 \mid^2 + \mid 49.25 - 45.714 \mid^2 + \mid 43.70 - 45.714 \mid^2}{5}}$$

$$= \sqrt{\frac{2.353156 + 0.001936 + 0.003136 + 12.503296 + 4.056196}{5}} = \sqrt{\frac{18.91772}{5}} = \mathbf{1.945133}$$

The runtime of index B and then index C is still worse than the original runtime without indexes. It is also worse than indexing on only C value. Compared to the runtime with an index only on B-value, it is still much faster than indexing on B. 45.714 seconds is less than 82.878.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql>
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);

CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B =
 R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------------+------------+---------+-----------------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys       | key        | key_len | ref             | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------------+------------+---------+-----------------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL                | NULL       | NULL    | NULL            | 1000 |    10.00 | Using where                                  |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL                | NULL       | NULL    | NULL            | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL                | NULL       | NULL    | NULL            | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2,index_C_R2 | index_C_R2 | 5       | cs157b_hw3.R3.C |  100 |    20.80 | Using where                                  |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1,index_C_R1 | index_C_R1 | 5       | cs157b_hw3.R3.C |  100 |    20.90 | Using where                                  |
+----+-------------+-------+------------+------+---------------------+------------+---------+-----------------+------+----------+----------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

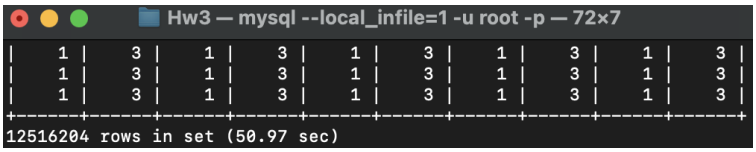
Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

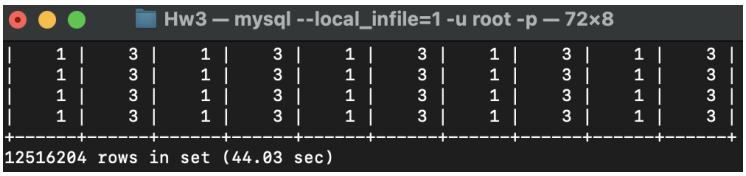
From the output of EXPLAIN, we can see the 2 indexes we made under "possible_keys" and "key" columns, so that means they are being used. Under "type," it says "ref" and "key" column is not null for those with the index, but those without the index have type "ALL" and ref "NULL". We can also see in the "Extra" column the usage of where and hash-join. DBMS seems to be using an index-based **join algorithm** and **hash-join algorithm**.

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

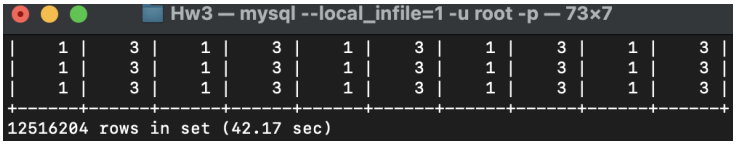
1. Execution Time 1: **50.97 seconds**



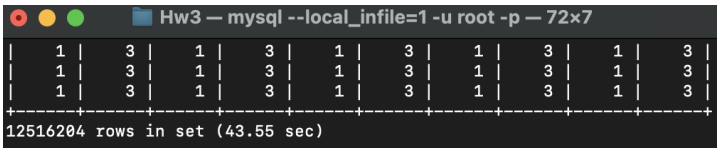
2. Execution Time 2: **44.03 seconds**



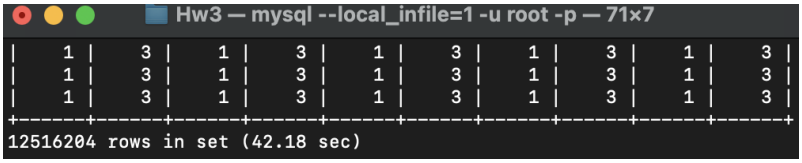
3. Execution Time 3: **42.17 seconds**



4. Execution Time 4: **43.55 seconds**



5. Execution Time 5: **42.18 seconds**



Average = (50.97 + 44.03 + 42.17 + 43.55 + 42.18) / 5 = **44.58 seconds**

Standard deviation $= \sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 50.97 - 44.58 \mid^2 + \mid 44.03 - 44.58 \mid^2 + \mid 42.17 - 44.58 \mid^2 + \mid 43.55 - 44.58 \mid^2 + \mid 42.18 - 44.58 \mid^2}{5}}$$

$$= \sqrt{\frac{40.8321 + 0.3025 + 5.8081 + 1.0609 + 5.76}{5}} = \sqrt{\frac{53.7637}{5}} = \mathbf{3.279134}$$

The runtime of index B and then index C is, again, still worse than the original runtime without indexes. It is also still worse than indexing on only C value. Compared to the runtime with an index only on B-value, it is still much faster than indexing on B. 44.58 seconds is less than 82.878. Partially applying indexes on two relations is faster than applying indexes on all the relations. 44.58 seconds < 45.714.

4. **Experiment 1.4: Index on (B, C) i.e. composite index of B-value and C-value**

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_BC_R1 ON R1 (B,C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R2 ON R2 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R3 ON R3 (B,C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R4 ON R4 (B,C);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R5 ON R5 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

```
CREATE INDEX index_BC_R1 ON R1 (B,C);
CREATE INDEX index_BC_R2 ON R2 (B,C);
CREATE INDEX index_BC_R3 ON R3 (B,C);
CREATE INDEX index_BC_R4 ON R4 (B,C);
CREATE INDEX index_BC_R5 ON R5 (B,C);
```

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R
1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5
.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+-------------+---------+---------------------+------+----------+-------------------------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref                 | rows | filtered | Extra                   |
+----+-------------+-------+------------+------+---------------+-------------+---------+---------------------+------+----------+-------------------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_BC_R5   | index_BC_R5 | 5       | const               |   88 |   100.00 | Using where; Using index |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_BC_R4   | index_BC_R4 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_BC_R3   | index_BC_R3 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_BC_R2   | index_BC_R2 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_BC_R1   | index_BC_R1 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
+----+-------------+-------+------------+------+---------------+-------------+---------+---------------------+------+----------+-------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

From the output of EXPLAIN, we can see the index we just made under “possible_keys” and “key” columns. Under “type,” it says “ref”. We can also see in the “Extra” column that the DBMS uses Where and index for the first relation R5 and keeps using the index for the rest of the relations. DBMS is using an **index-join algorithm**.

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

1. Execution Time 1: **7.33 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.33 sec)
```

2. Execution Time 2: **7.82 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.82 sec)
```

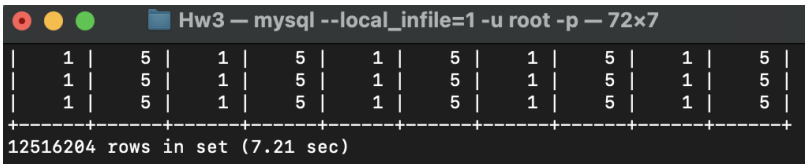
3. Execution Time 3: **7.65 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.65 sec)
```

4. Execution Time 4: **7.97 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.97 sec)
```

5. Execution Time 5: **7.21** seconds



Average = (7.33 + 7.82 + 7.65 + 7.97 + 7.21) / 5 = **7.596 seconds**

Standard deviation = $\sqrt{\frac{\Sigma\,|\,x-\mu\,|^2}{N}}$

$$= \sqrt{\frac{|\,7.33-7.596\,|^2+|\,7.82-7.596\,|^2+|\,7.65-7.596\,|^2+|\,7.97-7.596\,|^2+|\,7.21-7.596\,|^2}{5}}$$

$$= \sqrt{\frac{0.070756+0.050176+0.002916+0.139876+0.148996}{5}} = \sqrt{\frac{0.41272}{5}} = \mathbf{0.287305}$$

The runtime of the index (B, C) on all relations had a good time speed. It is faster than the original runtime, the runtime of B-values only index, the runtime of C-values only index, and the runtime with index B and then index C.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql> CREATE INDEX index_BC_R1 ON R1 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R2 ON R2 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

CREATE INDEX index_BC_R1 ON R1 (B,C);
CREATE INDEX index_BC_R2 ON R2 (B,C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B
 = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);
+----+-------------+-------+------------+------+---------------+-------------+---------+-----------------------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref                   | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+-------------+---------+-----------------------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL        | NULL    | NULL                  | 1000 |    10.00 | Using where                                        |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL        | NULL    | NULL                  | 1000 |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL        | NULL    | NULL                  | 1000 |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_BC_R2   | index_BC_R2 | 10      | const,cs157b_hw3.R3.C |   20 |   100.00 | Using index                                        |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_BC_R1   | index_BC_R1 | 10      | const,cs157b_hw3.R3.C |   20 |   100.00 | Using index                                        |
+----+-------------+-------+------------+------+---------------+-------------+---------+-----------------------+------+----------+----------------------------------------------------+
5 rows in set, 1 warning (0.01 sec)
```

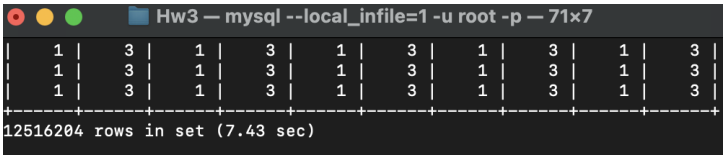
Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM ((((R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN R3 ON R2.B = R3.B AND R2.C = R3.C) JOIN R4 ON R3.B = R4.B AND R3.C = R4.C) JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1);

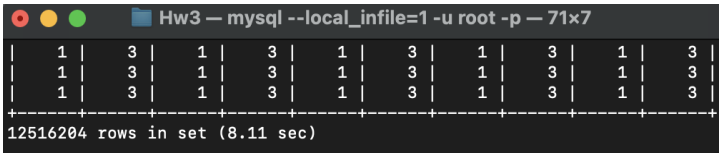
From the output of EXPLAIN, we can see the 2 indexes we made under "possible_keys" and "key," so that means they are being used. Under "type," it says "ref" and "ALL". We can also see in the "Extra" column that the DBMS uses where for the first relation R3. Then it uses a hash-join for relations that do not have an index. For those with indexes, they are using indexes. DBMS is using **hash-join** and **index-based join**.

Execute Left-deep Join Query 5 times. Compute the Average and Standard Deviation:

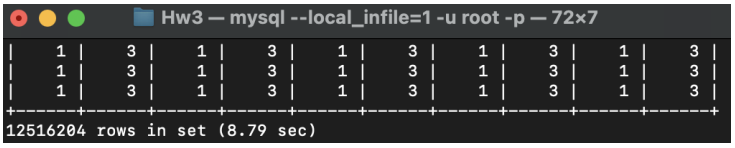
1. Execution Time 1: **7.43 seconds**



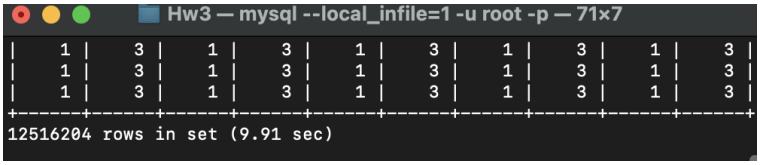
2. Execution Time 2: **8.11 seconds**



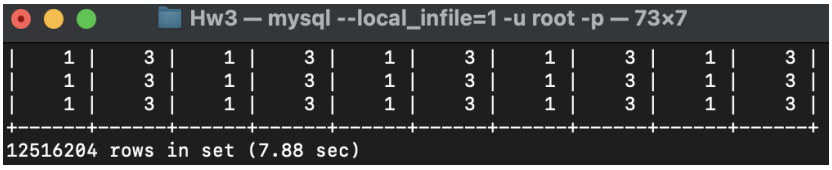
3. Execution Time 3: **8.79 seconds**



4. Execution Time 4: **9.91 seconds**



5. Execution Time 5: **7.88 seconds**



Average = (7.43 + 8.11 + 8.79 + 9.91 + 7.88) / 5 = **8.424 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$= \sqrt{\frac{\mid 7.43 - 8.424 \mid^2 + \mid 8.11 - 8.424 \mid^2 + \mid 8.79 - 8.424 \mid^2 + \mid 9.91 - 8.424 \mid^2 + \mid 7.88 - 8.424 \mid^2}{5}}$

$= \sqrt{\frac{0.988036 + 0.098596 + 0.133956 + 2.208196 + 0.295936}{5}} = \sqrt{\frac{3.72472}{5}} =$ **0.863101**

The runtime of the index (B, C) on two relations had a good time speed. It is faster than the original runtime, the runtime of B-values only index, the runtime of C-values only index, and the runtime with index B and then index C. However, it is not as fast as the runtime for with (B, C) on all relations.

Experiment 1 Conclusion:

Adding indexes to relations does not always help the query run faster. Adding the correct indexes (e.g. composite index of B and C) would help the query while indexes on B or C alone may have a negative impact on query time. As seen in the B-value index-only section, that attribute only made time slower. Having an index sometimes eliminates the use of hash-join which may slow things down. In this case, table-scan with hash-join would perform better.

Without an index, DBMS just uses hash-join.

Average without Index: 14.64 seconds
SD: 0.214009
Average with Composite Index B,C): 7.596 seconds
SD: 0.287305

==========

Experiment 2: Bushy Join Tree

Join SQL Query:

SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

Using MySQL DBMS "EXPLAIN" statement:

```
mysql>
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
(R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.
B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | R1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                  |
|  1 | SIMPLE      | R2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:

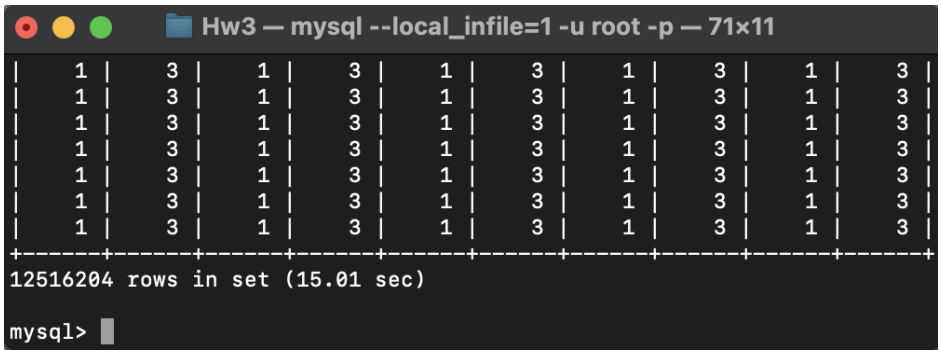
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND

R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

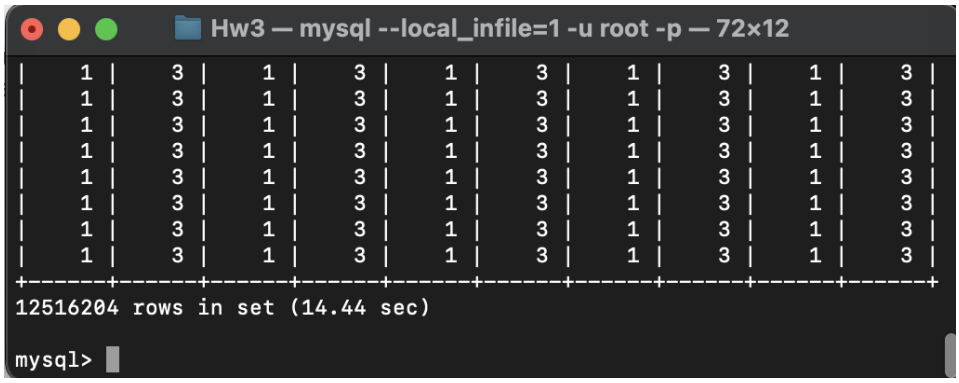
From the output of EXPLAIN, we can see in the “type” column that the DBMS marks it “ALL.” We can also see in the “Extra” column that the DBMS just uses where in R1 and then uses where and **hash-join algorithm** for the rest.
The “ALL” means that MySQL scans the whole table using no indexes to access/join the table, so the DBMS access and scans through R1 filtering with the where condition. It then accesses R2 with the join buffer and so forth.
This is the same as the left-deep join tree without indexes.

Execute Query 5 times. Compute the Average and Standard Deviation Result:
The 10 columns shown after the execution are as follows: R1B, R1C, R2B, R2C, R3B, R3C, R4B, R4C, R5B, R5C.

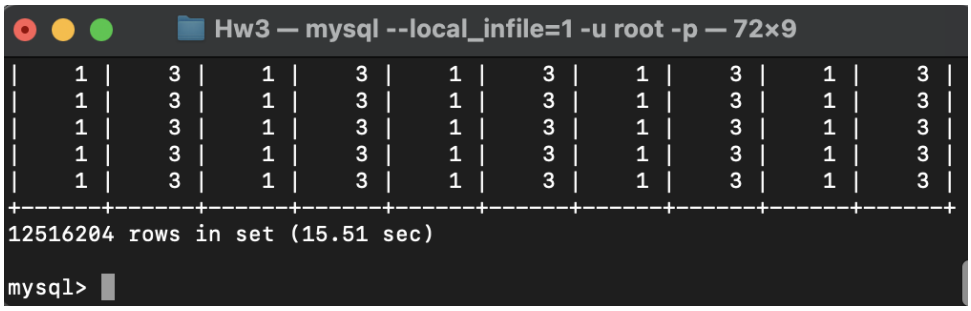
1. Execution Time 1: **15.01 seconds**



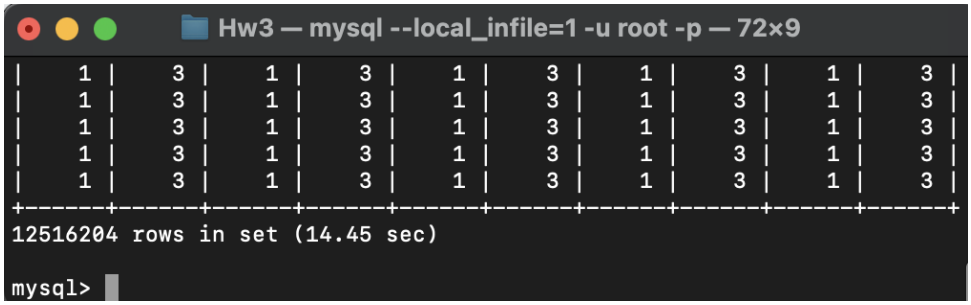
2. Execution Time 2: **14.44 seconds**



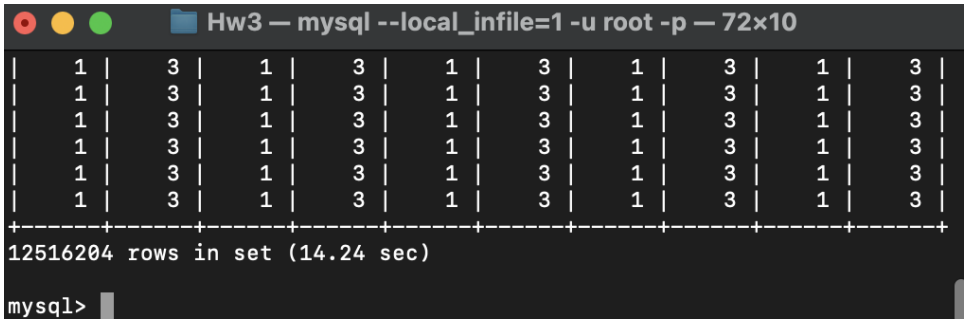
3. Execution Time 3: **15.51 seconds**



4. Execution Time 4: **14.15 seconds**



5. Execution Time 5: **14.24 seconds**



At this point, we have the 5 times from the 5 executions of the join query. We will compute the average and standard deviation.

Average = (15.01 + 14.44 + 15.51 + 14.15 + 14.24) / 5= **14.67 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x-\mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 15.01-14.67 \mid^2 + \mid 14.44 -14.67 \mid^2 + \mid 15.51-14.67 \mid^2 + \mid 14.15-14.67 \mid^2 + \mid 14.24-14.67 \mid^2}{5}}$$

$$= \sqrt{\frac{0.1156+0.0529+0.7056+0.2704+0.1849}{5}} = \sqrt{\frac{1.3294}{5}} = \mathbf{0.515636}$$

Effect of Indices: Use Indexes and Compare:
We will test whether using indexes will improve our timing in a bushy tree join tree.
We will create indexes for attributes that are part of "JOIN ON" conditions. In this case, that will be the attributes B and C.

There are 4 index experiments to try on the 5 relations:

1. Create index on B-value only
2. Create index on C-value only
3. Create index on B. Then create index on C
4. Create a composite index (B, C)

1. **Experiment 1.1: Index on B-value only (no index on C-value)**

   Case 1: Create the index on all 5 relations
   Create Index:

```
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R3 ON R3 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R4 ON R4 (B);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R5 ON R5 (B);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

   Query used:
   CREATE INDEX index_B_R1 ON R1 (B);
   CREATE INDEX index_B_R2 ON R2 (B);
   CREATE INDEX index_B_R3 ON R3 (B);
   CREATE INDEX index_B_R4 ON R4 (B);
   CREATE INDEX index_B_R5 ON R5 (B);

   Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B
 AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C
AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key        | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_B_R5    | index_B_R5 | 5       | const |   88 |   100.00 | NULL        |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_B_R4    | index_B_R4 | 5       | const |  118 |    10.00 | Using where |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_B_R3    | index_B_R3 | 5       | const |  174 |    10.00 | Using where |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2    | index_B_R2 | 5       | const |  208 |    10.00 | Using where |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1    | index_B_R1 | 5       | const |  209 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-------------+
5 rows in set, 1 warning (0.01 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

From the output of EXPLAIN, we can see the index we made under “possible_keys” and “key,” so that means they are being used. Under “type,” it says “ref”. We can also see in the “Extra” column that the DBMS uses where for the relations. DBMS is using **index-based join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

1. Execution Time 1: **1 minute and 34.10 seconds (94.10 seconds)**

```
Hw3 — mysql --local_infile=1 -u root -p — 72×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (1 min 34.10 sec)
```

2. Execution Time 2: **1 minute and 28.33 seconds (88.33 seconds)**

```
Hw3 — mysql --local_infile=1 -u root -p — 72×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (1 min 28.33 sec)
```

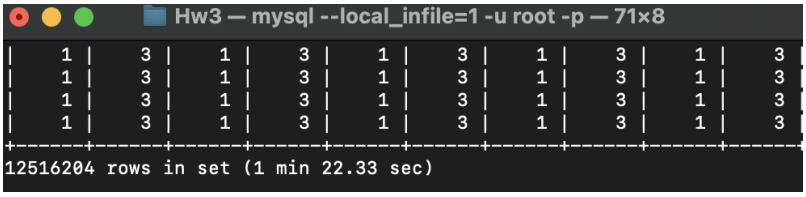
3. Execution Time 3: **1 minute and 24.29 seconds (84.29 seconds**)

```
Hw3 — mysql --local_infile=1 -u root -p — 72×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (1 min 24.29 sec)
```

4. Execution Time 4: **1 minute and 22.31 seconds (82.31 seconds)**

```
Hw3 — mysql --local_infile=1 -u root -p — 72×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (1 min 22.31 sec)
```

5. Execution Time 5: **1 minute and 22.33 seconds (82.33 seconds)**



Average = (94.10 + 88.33 + 84.29 + 82.31 + 82.33) / 5 = **86.272 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 94.10 - 86.272 \mid^2 + \mid 88.33 - 86.272 \mid^2 + \mid 84.29 - 86.272 \mid^2 + \mid 82.31 - 86.272 \mid^2 + \mid 82.33 - 86.272 \mid^2}{5}}$$

$$= \sqrt{\frac{61.277584 + 4.235364 + 3.928324 + 15.697444 + 15.539364}{5}} = \sqrt{\frac{100.67808}{5}} = \mathbf{4.487273}$$

The time bushy tree average runtime is 86.272 seconds which is much slower than the average runtime without indexes (14.67 seconds). Let's try to see if lesser indices are faster.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOI
N R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R
2.C = R3.C;
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key        | key_len | ref   | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2    | index_B_R2 | 5       | const |  208 |   100.00 | NULL                                               |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1    | index_B_R1 | 5       | const |  209 |    10.00 | Using where                                        |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL  | 1000 |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL  | 1000 |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL  | 1000 |     1.00 | Using where; Using join buffer (hash join)         |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

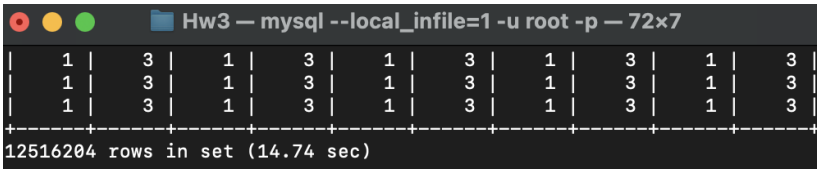
Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM

(R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

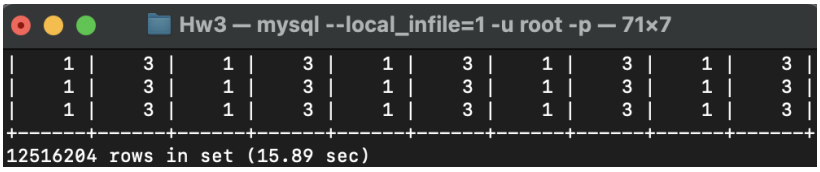
From the output of EXPLAIN, we can see the 2 indexes we made under "possible_keys" and "key," so that means they are being used. Under "type," it says "ref" and "ALL". We can also see in the "Extra" column that the DBMS uses where for the first relation R2 and then uses hash-join for those without indexes. DBMS is using **hash-join** and **index-based join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

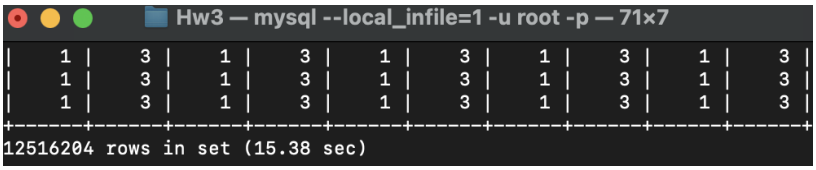
1. Execution Time 1: **14.74 seconds**



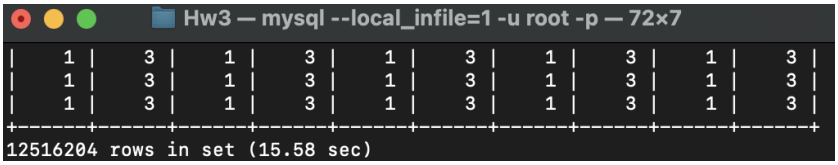
2. Execution Time 2: **15.89 seconds**



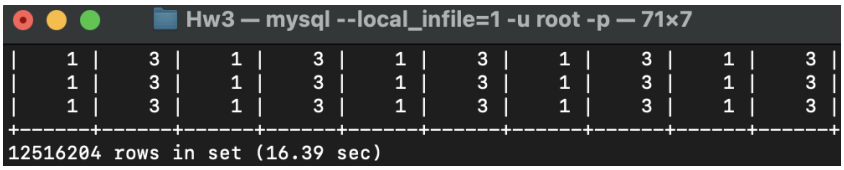
3. Execution Time 3: **15.38 seconds**



4. Execution Time 4: **15.58 seconds**



5. Execution Time 5: **16.39 seconds**



Average = (14.74 + 15.89 + 15.38 + 15.58 + 16.39) / 5 = **15.596 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{|14.74-15.596|^2+|15.89-15.596|^2+|15.38-15.596|^2+|15.58-15.596|^2+|16.39-15.596|^2}{5}}$$

$$= \sqrt{\frac{0.732736+0.086436+0.046656+0.000256+0.630436}{5}} = \sqrt{\frac{1.49652}{5}} = \mathbf{0.547087}$$

Notice how the time the average runtime (15.596 seconds) is faster than the one with the B value index on all relations (86.272 seconds). However, it is not as fast as the one without indexes.
Indexes are not useful here because of the overhead.

2. **Experiment 1.2: Index on C-value only**

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R3 ON R3 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R4 ON R4 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R5 ON R5 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);
CREATE INDEX index_C_R3 ON R3 (C);
CREATE INDEX index_C_R4 ON R4 (C);
CREATE INDEX index_C_R5 ON R5 (C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FRO
M (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON
 R2.B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                   |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------+
|  1 | SIMPLE      | R4    | NULL       | ALL  | index_C_R4    | NULL | NULL    | NULL | 1000 |    10.00 | Using where                             |
|  1 | SIMPLE      | R1    | NULL       | ALL  | index_C_R1    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R2    | NULL       | ALL  | index_C_R2    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R3    | NULL       | ALL  | index_C_R3    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R5    | NULL       | ALL  | index_C_R5    | NULL | NULL    | NULL | 1000 |     2.00 | Using where; Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------+
5 rows in set, 1 warning (0.01 sec)
```

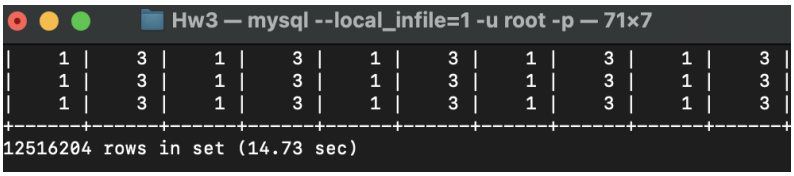
Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM

(R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

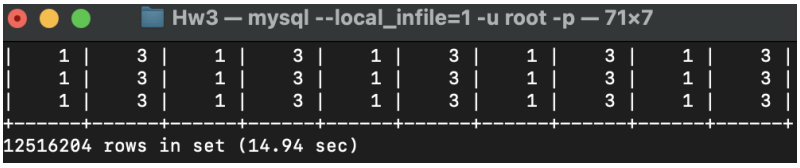
From the output of EXPLAIN, we can see the indexes we made under "possible_keys," but "key" shows all NULL, so that means they are NOT being used. We can also see in the "Extra" column that the DBMS uses where for the first relation R4. Then it uses a hash-join for the rest. DBMS is using **hash-join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

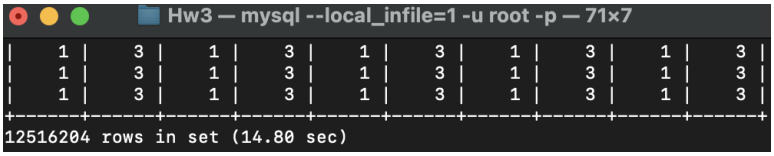
1. Execution Time 1: **14.73 seconds**



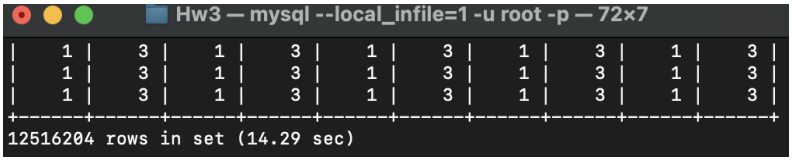
2. Execution Time 2: **14.94 seconds**



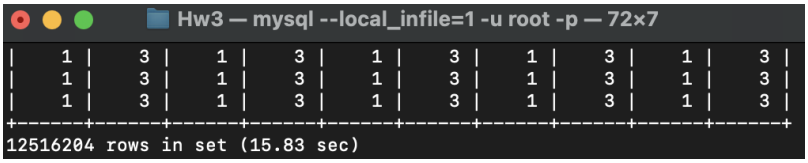
3. Execution Time 3: **14.80 seconds**



4. Execution Time 4: **14.29 seconds**



5. Execution Time 5: **15.83 seconds**



Average = (14.73 + 14.94 + 14.80 + 14.29 + 15.83) / 5 = **14.918 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{|14.73-14.918|^2+|14.94-14.918|^2+|14.80-14.918|^2+|14.29-14.918|^2+|15.83-14.918|^2}{5}}$$

$$= \sqrt{\frac{0.035344+0.000484+0.013924+0.394384+0.831744}{5}} = \sqrt{\frac{1.27588}{5}} = \mathbf{0.505149}$$

Observe how indexing on all relations with C-value is still slower than performing the join with no indexes by about 0.3 seconds.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql>
mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);

Using DBMS EXPLAIN statement:

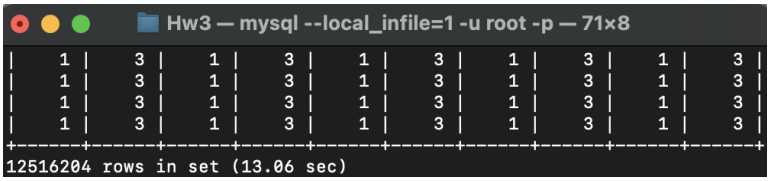
```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FRO
M (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON
 R2.B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                  |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R1    | NULL       | ALL  | index_C_R1    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
|  1 | SIMPLE      | R2    | NULL       | ALL  | index_C_R2    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)   |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

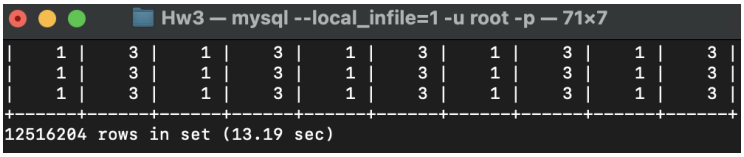
From the output of EXPLAIN, we can see the 2 indexes we made under “possible_keys.” The indexes are not being used because “key,” doesn’t show them. We can also see in the “Extra” column that the DBMS uses where for the first relation R3. Then it uses a hash-join for relations that do not have an index. DBMS is using **hash-join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

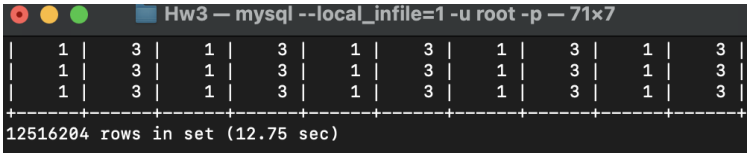
1. Execution Time 1: **13.06 seconds**



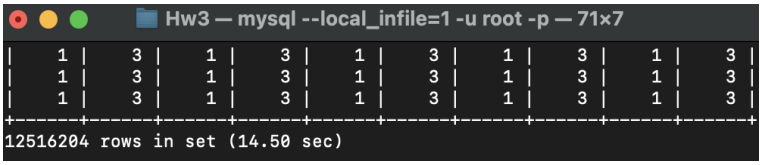
2. Execution Time 2: **13.19 seconds**



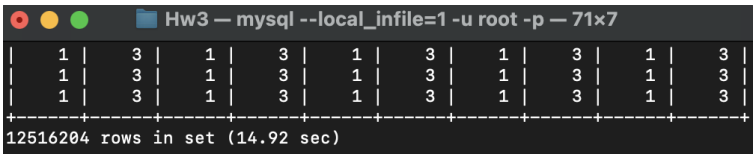
3. Execution Time 3: **12.75 seconds**



4. Execution Time 4: **14.50 seconds**



5. Execution Time 5: **14.92 seconds**



Average = (13.06 + 13.19 + 12.75 + 14.50 + 14.92) / 5 = **13.684 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x-\mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 13.06-13.684 \mid^2 + \mid 13.19-13.684 \mid^2 + \mid 12.75-13.684 \mid^2 + \mid 14.50-13.684 \mid^2 + \mid 14.92-13.684 \mid^2}{5}}$$

$$= \sqrt{\frac{0.389376 + 0.244036 + 0.872356 + 0.665856 + 1.527696}{5}} = \sqrt{\frac{3.69932}{5}} = \mathbf{0.860153}$$

Notice how this time is much faster than if we make all the relations have a C-value index. This may indicate there is some overhead when we apply an attribute as the index to all relations.

3. **Experiment 1.3: Index on B. Then index on C.**

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_B_R3 ON R3 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R4 ON R4 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R5 ON R5 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R3 ON R3 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R4 ON R4 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R5 ON R5 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

```
CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);
CREATE INDEX index_B_R3 ON R3 (B);
CREATE INDEX index_B_R4 ON R4 (B);
CREATE INDEX index_B_R5 ON R5 (B);

CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);
CREATE INDEX index_C_R3 ON R3 (C);
CREATE INDEX index_C_R4 ON R4 (C);
CREATE INDEX index_C_R5 ON R5 (C);
```

Using DBMS EXPLAIN statement:

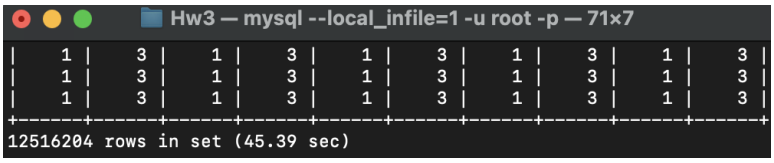
```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R
5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.
C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+-----------------------+------------+---------+-----------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys         | key        | key_len | ref             | rows | filtered | Extra       |
+----+-------------+-------+------------+------+-----------------------+------------+---------+-----------------+------+----------+-------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_B_R5,index_C_R5 | index_B_R5 | 5       | const           |   88 |   100.00 | Using where |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_B_R3,index_C_R3 | index_C_R3 | 5       | cs157b_hw3.R5.C |  100 |    17.40 | Using where |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2,index_C_R2 | index_C_R2 | 5       | cs157b_hw3.R5.C |  100 |    20.80 | Using where |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_B_R4,index_C_R4 | index_B_R4 | 5       | const           |  118 |    20.00 | Using where |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1,index_C_R1 | index_C_R1 | 5       | cs157b_hw3.R5.C |  100 |    20.90 | Using where |
+----+-------------+-------+------------+------+-----------------------+------------+---------+-----------------+------+----------+-------------+
5 rows in set, 1 warning (0.01 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

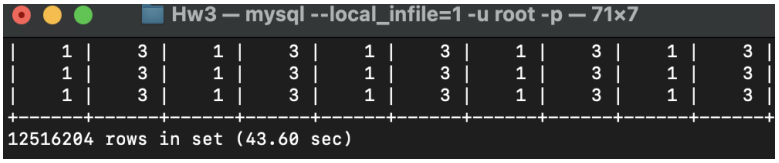
From the output of EXPLAIN, we can see the index we made under “possible_keys” and “key,” so that means they are being used. Under “type,” it says “ref.” We can also see in the “Extra” column that the DBMS uses where for all the relations. DBMS is using **index-based join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

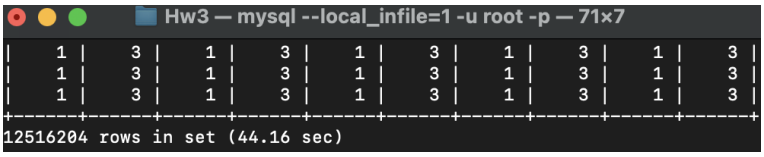
1. Execution Time 1: **45.39 seconds**



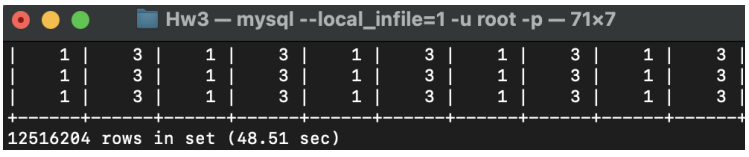
2. Execution Time 2: **43.60 seconds**



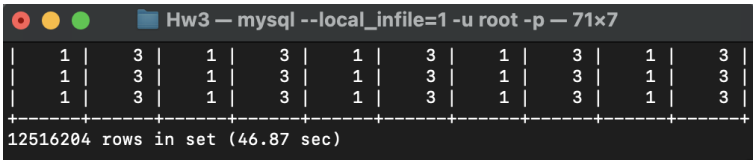
3. Execution Time 3: **44.16 seconds**



4. Execution Time 4: **48.51 seconds**



5. Execution Time 5: **46.87 seconds**



Average = (45.39 + 43.60 + 44.16 + 48.51 + 46.87) / 5 = **45.706 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 45.39 - 45.706 \mid^2 + \mid 43.60 - 45.706 \mid^2 + \mid 44.16 - 45.706 \mid^2 + \mid 48.51 - 45.706 \mid^2 + \mid 46.87 - 45.706 \mid^2}{5}}$$

$$= \sqrt{\frac{0.099856 + 4.435236 + 2.390116 + 7.862416 + 1.354896}{5}} = \sqrt{\frac{16.14252}{5}} = \mathbf{1.796804}$$

The runtime of index B and then index C is still much worse than the original runtime without indexes. It is also worse than indexing on only C value. Compared to the runtime with an index only on B-value, it is still much faster than indexing on B.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql>
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);

CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R
2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+-----------------------+-----------+---------+-------------------+------+----------+-----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys         | key       | key_len | ref               | rows | filtered | Extra                                         |
+----+-------------+-------+------------+------+-----------------------+-----------+---------+-------------------+------+----------+-----------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL                  | NULL      | NULL    | NULL              | 1000 |    10.00 | Using where                                   |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL                  | NULL      | NULL    | NULL              | 1000 |     1.00 | Using where; Using join buffer (hash join)    |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL                  | NULL      | NULL    | NULL              | 1000 |     1.00 | Using where; Using join buffer (hash join)    |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2,index_C_R2 | index_C_R2 | 5       | cs157b_hw3.R3.C |  100 |    20.80 | Using where                                   |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1,index_C_R1 | index_C_R1 | 5       | cs157b_hw3.R3.C |  100 |    20.90 | Using where                                   |
+----+-------------+-------+------------+------+-----------------------+-----------+---------+-------------------+------+----------+-----------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

From the output of EXPLAIN, we can see the 2 indexes we made under “possible_keys” and “key,” so that means they are being used. Under “type,” it says “ref” and “ALL”. We can also see in the “Extra” column that the DBMS uses where for the first relation R3. Then it uses a hash-join for relations that do not have an index. For those with indexes, they are using indexes. DBMS is using **hash-join** and **index-based join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

1. Execution Time 1: **45.54 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (45.54 sec)
```

2. Execution Time 2: **46.31 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (46.31 sec)
```

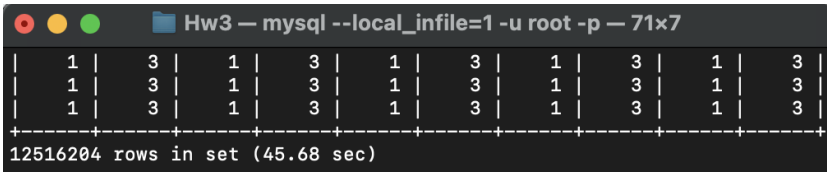
3. Execution Time 3: **49.57 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 72×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (49.57 sec)
```

4. Execution Time 4: **48.83 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (48.83 sec)
```

5. Execution Time 5: **45.68 seconds**



Average = (45.54 + 46.31 + 49.57 + 48.83 + 45.68) / 5 = **47.186 seconds**

Standard deviation = $\sqrt{\frac{\Sigma\,|\,x-\mu\,|^2}{N}}$

$$= \sqrt{\frac{|\,45.54-47.186\,|^2+|\,46.31-47.186\,|^2+|\,49.57-47.186\,|^2+|\,48.83-47.186\,|^2+|\,45.68-47.186\,|^2}{5}}$$

$$= \sqrt{\frac{2.709316+1.115136+5.683456+2.702736+2.268036}{5}} = \sqrt{\frac{14.47868}{5}} = \mathbf{1.701686}$$

The runtime of index B and then index C is, again, still worse than the original runtime without indexes. It is also still worse than indexing only C-value. Compared to the runtime with an index only on B-value, it is still much faster than indexing on B. Interestingly, applying this method of indexing on only 2 relations is slower than applying it to all 5 relations.

4. **Experiment 1.4: Index on (B, C) i.e. composite index of B-value and C-value**

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_BC_R1 ON R1 (B,C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R2 ON R2 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R3 ON R3 (B,C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R4 ON R4 (B,C);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R5 ON R5 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

```
CREATE INDEX index_BC_R1 ON R1 (B,C);
CREATE INDEX index_BC_R2 ON R2 (B,C);
CREATE INDEX index_BC_R3 ON R3 (B,C);
CREATE INDEX index_BC_R4 ON R4 (B,C);
CREATE INDEX index_BC_R5 ON R5 (B,C);
```

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1
JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B
 AND R2.C = R3.C;
+----+-------------+-------+------------+------+---------------+-------------+---------+---------------------+------+----------+--------------------------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref                 | rows | filtered | Extra                    |
+----+-------------+-------+------------+------+---------------+-------------+---------+---------------------+------+----------+--------------------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_BC_R5   | index_BC_R5 | 5       | const               |   88 |   100.00 | Using where; Using index |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_BC_R4   | index_BC_R4 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index              |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_BC_R3   | index_BC_R3 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index              |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_BC_R2   | index_BC_R2 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index              |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_BC_R1   | index_BC_R1 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index              |
+----+-------------+-------+------------+------+---------------+-------------+---------+---------------------+------+----------+--------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

From the output of EXPLAIN, we can see the index we made under “possible_keys” and “key,” so that means they are being used. We can also see in the “Extra” column that the DBMS uses indexes for all and where for just the first relation R5. DBMS is using **index-based join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

1. Execution Time 1: **7.55 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.55 sec)
```

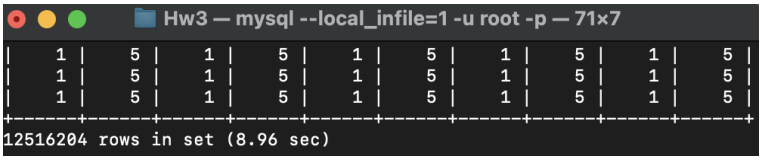
2. Execution Time 2: **7.71 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.71 sec)
```

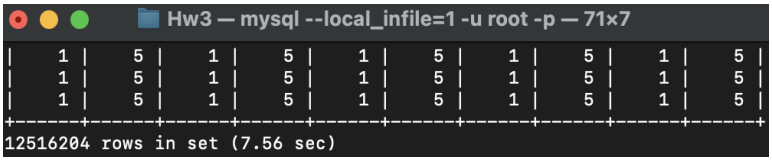
3. Execution Time 3: **8.42 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (8.42 sec)
```

4. Execution Time 4: **8.96 seconds**



5. Execution Time 5: **7.56 seconds**



Average = (7.55 + 7.71 + 8.42 + 8.96 + 7.56) / 5 = **8.04 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 7.55 - 8.04 \mid^2 + \mid 7.71 - 8.04 \mid^2 + \mid 8.42 - 8.04 \mid^2 + \mid 8.96 - 8.04 \mid^2 + \mid 7.56 - 8.04 \mid^2}{5}}$$

$$= \sqrt{\frac{0.2401 + 0.1089 + 0.1444 + 0.8464 + 0.2304}{5}} = \sqrt{\frac{1.5702}{5}} = \mathbf{0.560393}$$

The runtime of the index (B, C) on all relations had a quick runtime speed. It is faster than the original runtime, the runtime of B-values only index, the runtime of C-values only index, and the runtime with index B and then index C.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql> CREATE INDEX index_BC_R1 ON R1 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R2 ON R2 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_BC_R1 ON R1 (B,C);
CREATE INDEX index_BC_R2 ON R2 (B,C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B =
R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;
+----+-------------+-------+------------+------+---------------+-------------+---------+-------------------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref               | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------------------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL        | NULL    | NULL              | 1000 |    10.00 | Using where                                         |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL        | NULL    | NULL              | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL        | NULL    | NULL              | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_BC_R2   | index_BC_R2 | 10      | const,cs157b_hw3.R3.C |   20 |   100.00 | Using index                                         |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_BC_R1   | index_BC_R1 | 10      | const,cs157b_hw3.R3.C |   20 |   100.00 | Using index                                         |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------------------+------+----------+-----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

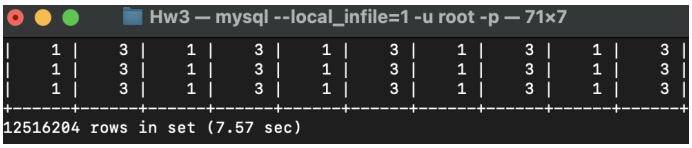
Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN R2 ON R1.B = R2.B AND R1.C = R2.C) JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C;

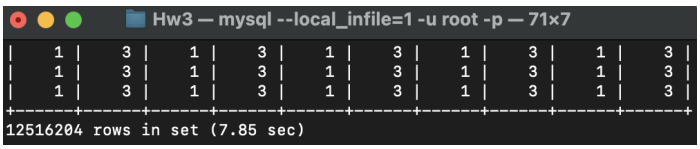
From the output of EXPLAIN, we can see the 2 indexes we made under "possible_keys" and "key," so that means they are being used. Under "type," it says "ref" and "ALL". We can also see in the "Extra" column that the DBMS uses where for the first relation R3. Then it uses a hash-join for relations that do not have an index. For those with indexes, they are using indexes. DBMS is using **hash-join** and **index-based join**.

Execute Bushy Tree Join Query 5 times. Compute the Average and Standard Deviation:

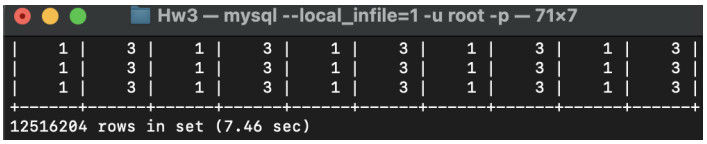
1. Execution Time 1: **7.57 seconds**



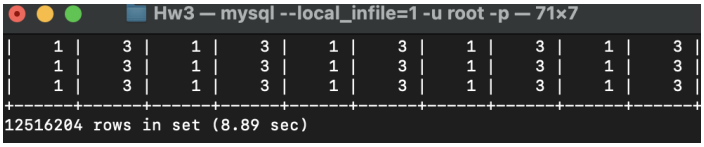
2. Execution Time 2: **7.85 seconds**



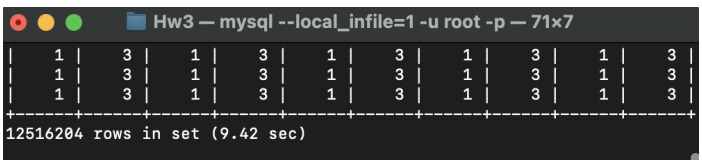
3. Execution Time 3: **7.46 seconds**



4. Execution Time 4: **8.89 seconds**



5. Execution Time 5: **9.42 seconds**



Average = (7.57 + 7.85 + 7.46 + 8.89 + 9.42) / 5 = **8.238 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{|7.57 - 8.238|^2 + |7.85 - 8.238|^2 + |7.46 - 8.238|^2 + |8.89 - 8.238|^2 + |9.42 - 8.238|^2}{5}}$$

$$= \sqrt{\frac{0.446224 + 0.150544 + 0.605284 + 0.425104 + 1.397124}{5}} = \sqrt{\frac{3.02428}{5}} = \mathbf{0.777725}$$

The runtime of the index (B, C) on two relations still has a fast runtime speed in general. It is faster than the original runtime, the runtime of B-values only index, the runtime of C-values only index, and the runtime with index B and then index C. It seems that in this case, it is a bit faster than the runtime for applying index (B,C) on all 5 relations.

Experiment 2 Conclusion:

Similar to Experiment 1, having indexes does not necessarily mean it will run faster than relations that do not have indexes. Indexing on a composite of B and C is faster.

Without an index, DBMS just uses hash-join.

Average without Index: 14.67 seconds
SD: 0.515636
Average with Composite Index B,C): 8.04 seconds
SD: 0.560393

==========

Experiment 3: Right-Deep Join Tree

Join SQL Query:

SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

Using MySQL DBMS "EXPLAIN" STATEMENT:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM
  (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON
R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | R1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                  |
|  1 | SIMPLE      | R2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
5 rows in set, 1 warning (0.00 sec)

mysql>
```

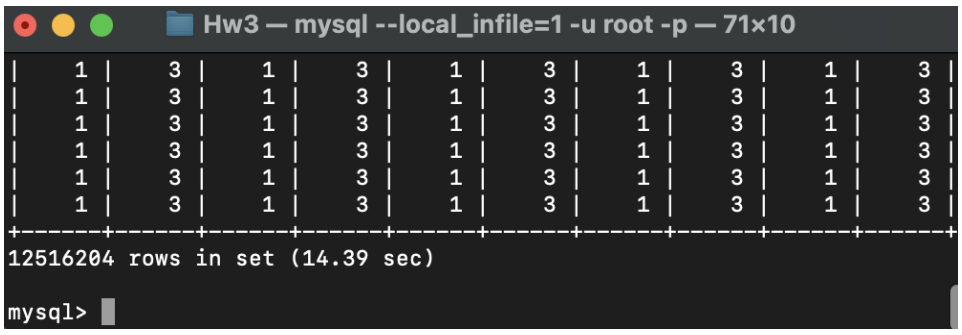
Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

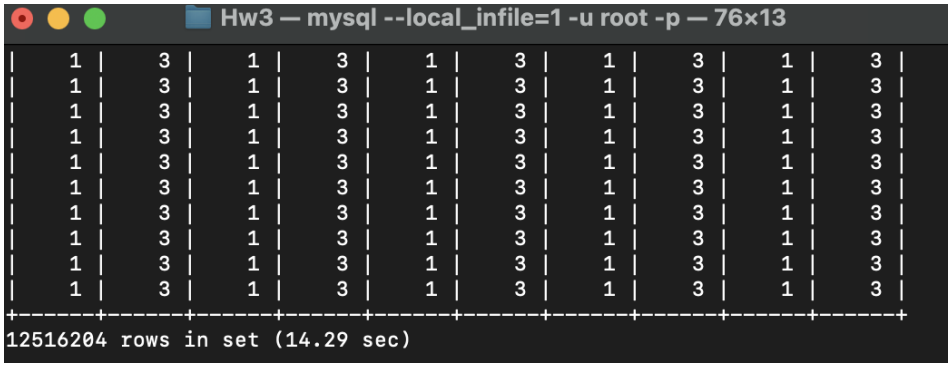
From the output of EXPLAIN, we can see in the "type" column that the DBMS marks it "ALL." We can also see in the "Extra" column that the DBMS just uses where in R1 and then uses where and **hash-join algorithm** for the rest.
The "ALL" means that MySQL scans the whole table using no indexes to access/join the table, so the DBMS access and scans through R1 filtering with the where condition. It then accesses R2 with the join buffer and so forth.
This is the same plan as the left-deep join tree and the bushy tree join without any index.

Execute Query 5 times. Compute the Average and Standard Deviation Result:
The 10 columns shown after the execution are as follows: R1B, R1C, R2B, R2C, R3B, R3C, R4B, R4C, R5B, R5C.

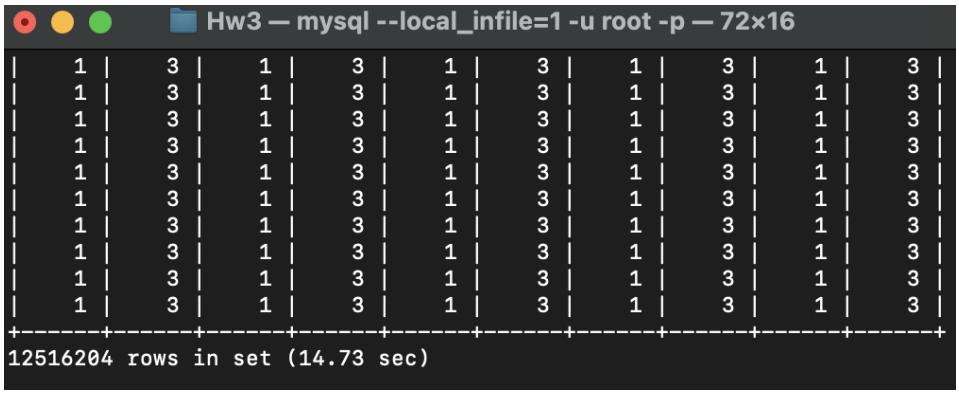
1. Execution 1 Time: **14.39 seconds**



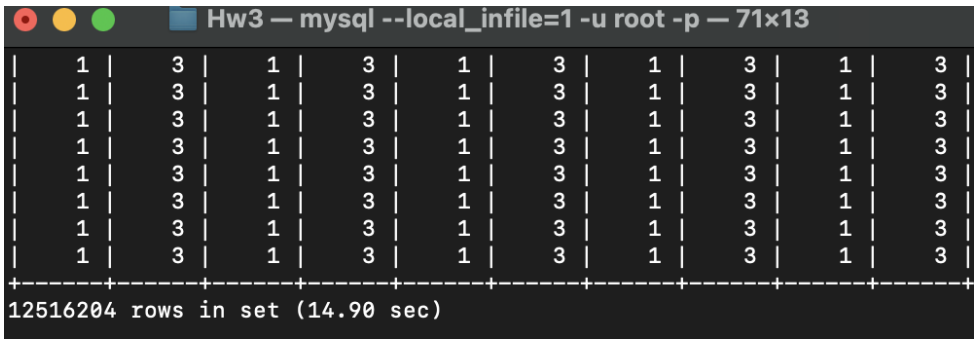
2. Execution 2 Time: **14.29 seconds**



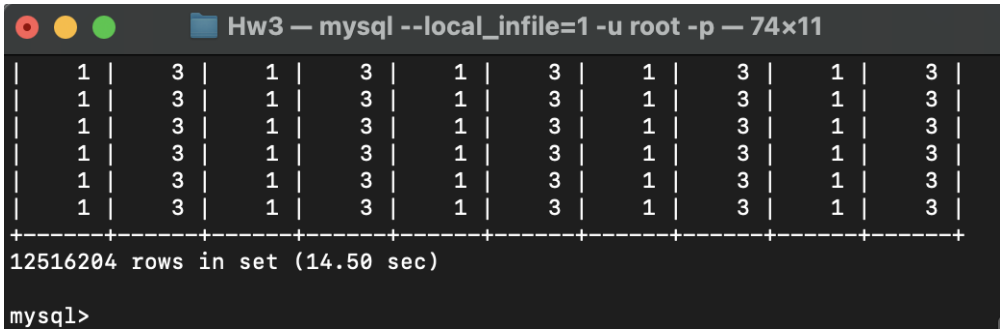
3. Execution 3 Time: **14.73 seconds**



4. Execution 4 Time: **14.90 seconds**



5. Execution 5 Time: **14.50 seconds**



At this point, we have the 5 times from the 5 executions of the join query. We will compute the average and standard deviation.

Average = (14.39 + 14.29 + 14.73 + 14.90 + 14.50) / 5 = **14.562 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$= \sqrt{\frac{\mid 14.39 - 14.562 \mid^2 + \mid 14.29 - 14.562 \mid^2 + \mid 14.73 - 14.562 \mid^2 + \mid 14.90 - 14.562 \mid^2 + \mid 14.50 - 14.562 \mid^2}{5}}$

$$= \sqrt{\frac{0.029584 + 0.073984 + 0.028224 + 0.114244 + 0.003844}{5}} = \sqrt{\frac{1.05908}{5}} = \mathbf{0.223553}$$

Effect of Indices: Use Indexes and Compare:

We will test whether using indexes will improve our timing in right-deep join tree.
We will create indexes for attributes that are part of "JOIN ON" conditions. In this case, that will be the attributes B and C.

There are 4 index experiments to try on the 5 relations:

1. Create index on B-value only
2. Create index on C-value only
3. Create index on B. Then create index on C
4. Create a composite index (B, C)

1. **Experiment 1.1: Index on B-value only (no index on C-value)**

   Case 1: Create the index on all 5 relations

   Create Index:

```
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R3 ON R3 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R4 ON R4 (B);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R5 ON R5 (B);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

   Query used:

```
CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);
CREATE INDEX index_B_R3 ON R3 (B);
CREATE INDEX index_B_R4 ON R4 (B);
CREATE INDEX index_B_R5 ON R5 (B);
```

Using DBMS EXPLAIN statement:

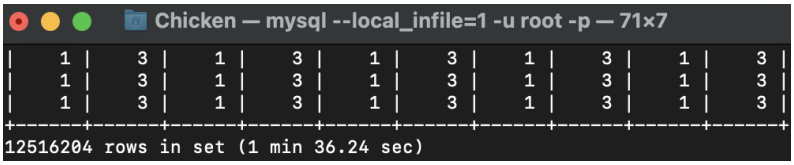
```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B
 AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AN
D R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+-----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key       | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+-----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_B_R5    | index_B_R5 | 5       | const |   88 |   100.00 | NULL        |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_B_R4    | index_B_R4 | 5       | const |  118 |    10.00 | Using where |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_B_R3    | index_B_R3 | 5       | const |  174 |    10.00 | Using where |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2    | index_B_R2 | 5       | const |  208 |    10.00 | Using where |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1    | index_B_R1 | 5       | const |  209 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+-----------+---------+-------+------+----------+-------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

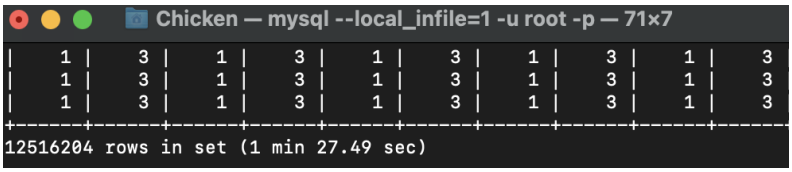
From the output of EXPLAIN, we can see the indexes we made under “possible_keys” and “key,” so that means they are being used. Under “type,” it says “ref”. We can also see in the “Extra” column that the DBMS mostly used where. DBMS uses **index-based join**.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

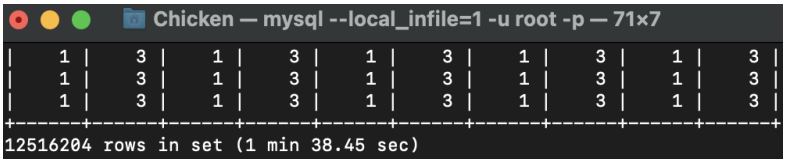
1. Execution Time 1: **1 minute and 36.24 seconds (96.24 seconds)**



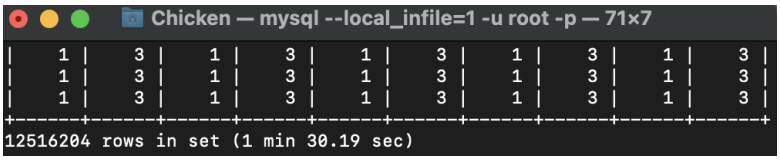
2. Execution Time 2: **1 minute and 27.49 seconds (87.49 seconds)**



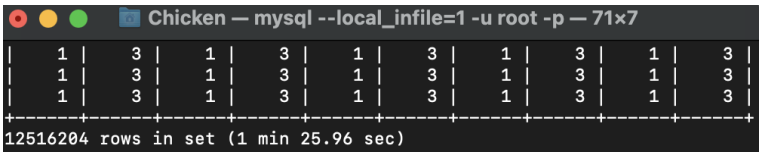
3. Execution Time 3: **1 minute and 38.45 seconds (98.45 seconds)**



4. Execution Time 4: **1 minute and 30.19 seconds (90.19 seconds)**



5. Execution Time 5: **1 minute and 25.96 seconds (85.96 seconds)**



Average = (96.24 + 87.49 + 98.45 + 90.19 + 85.96) / 5 = **91.666 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 96.24 - 91.666 \mid^2 + \mid 87.49 - 91.666 \mid^2 + \mid 98.45 - 91.666 \mid^2 + \mid 90.19 - 91.666 \mid^2 + \mid 85.96 - 91.666 \mid^2}{5}}$$

$$= \sqrt{\frac{20.921476 + 17.438976 + 46.022656 + 2.178576 + 32.558436}{5}} = \sqrt{\frac{119.12012}{5}} = \mathbf{4.880986}$$

The average runtime is 91.666 seconds which is much slower than the average time without indexes (14.562 seconds). Let's try to see if lesser indices are faster.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);

Using DBMS EXPLAIN statement:

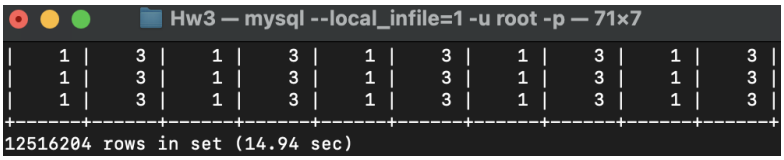
```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key        | key_len | ref   | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2    | index_B_R2 | 5       | const |  208 |   100.00 | NULL                                                |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1    | index_B_R1 | 5       | const |  209 |    10.00 | Using where                                         |
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL       | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
+----+-------------+-------+------------+------+---------------+------------+---------+-------+------+----------+-----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

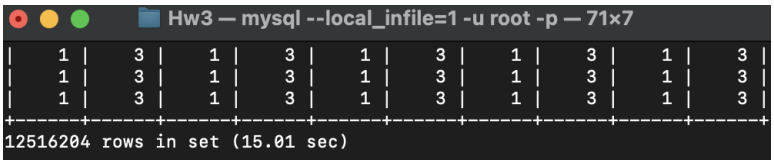
From the output of EXPLAIN, we can see the 2 indexes we made under "possible_keys" and "key," so that means they are being used. Under "type," it says "ref" and "ALL". We can also see in the "Extra" column that the DBMS uses where for the second relation it reached. Then it uses a hash-join for relations that do not have an index. For those with indexes, they are using indexes. DBMS is using **hash-join** and **index-based join**.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

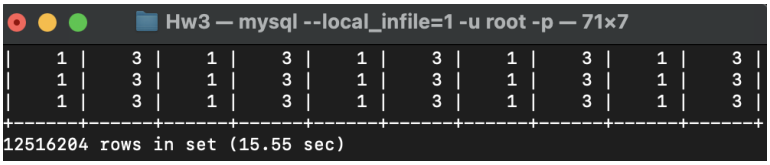
1. Execution Time 1: **14.94 seconds**



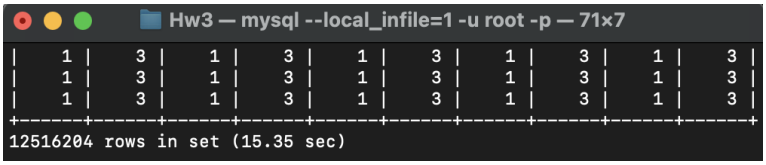
2. Execution Time 2: **15.01 seconds**



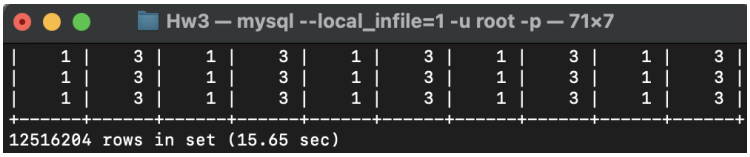
3. Execution Time 3: **15.55 seconds**



4. Execution Time 4: **15.35 seconds**



5. Execution Time 5: **15.65 seconds**



Average = (14.94 + 15.01 + 15.55 + 15.35 + 15.65) / 5 = **15.3 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 14.94 - 15.3 \mid^2 + \mid 15.01 - 15.3 \mid^2 + \mid 15.55 - 15.3 \mid^2 + \mid 15.35 - 15.3 \mid^2 + \mid 15.65 - 15.3 \mid^2}{5}}$$

$$= \sqrt{\frac{0.1296 + 0.0841 + 0.0625 + 0.0025 + 0.1225}{5}} = \sqrt{\frac{0.4012}{5}} = \mathbf{0.283267}$$

Notice how the average runtime (15.3 seconds) is faster than the one with the B value index on all relations. However, it is still not as fast as the one without indexes. Indexes are not useful here because of the overhead.

2. **Experiment 1.2: Index on C-value only**
   Case 1: Create the index on all 5 relations
   Create Index:

```
mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R3 ON R3 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R4 ON R4 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R5 ON R5 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

   Query used:
   CREATE INDEX index_C_R1 ON R1 (C);
   CREATE INDEX index_C_R2 ON R2 (C);
   CREATE INDEX index_C_R3 ON R3 (C);
   CREATE INDEX index_C_R4 ON R4 (C);
   CREATE INDEX index_C_R5 ON R5 (C);

   Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FRO
M (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) O
N R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------------+
|  1 | SIMPLE      | R4    | NULL       | ALL  | index_C_R4    | NULL | NULL    | NULL | 1000  |    10.00 | Using where                                        |
|  1 | SIMPLE      | R1    | NULL       | ALL  | index_C_R1    | NULL | NULL    | NULL | 1000  |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R2    | NULL       | ALL  | index_C_R2    | NULL | NULL    | NULL | 1000  |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R3    | NULL       | ALL  | index_C_R3    | NULL | NULL    | NULL | 1000  |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R5    | NULL       | ALL  | index_C_R5    | NULL | NULL    | NULL | 1000  |     2.00 | Using where; Using join buffer (hash join)         |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

From the output of EXPLAIN, we can see the index we made under “possible_keys.” They are NOT being used because “key” column is all NULL. We can also see in the “Extra” column that the DBMS uses where for the first relation R4. Then it uses a hash-join for relations that do not have an index. For those with indexes, they are using indexes. DBMS is using **hash-join** only.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

1. Execution Time 1: **15.18 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (15.18 sec)
```

2. Execution Time 2: **15.14 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (15.14 sec)
```

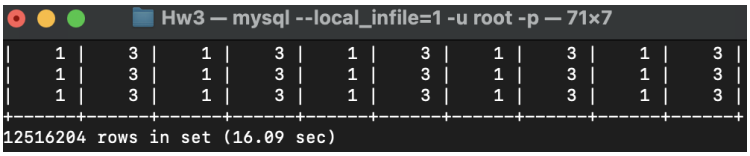
3. Execution Time 3: **16.07 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (16.07 sec)
```

4. Execution Time 4: **16.11 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
|    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |    1 |    3 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (16.11 sec)
```

5. Execution Time 5: **16.09 seconds**



Average = (15.18 + 15.14 + 16.07 + 16.11 + 16.09) / 5 = **15.718 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x-\mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 15.18-15.718 \mid^2 + \mid 15.14-15.718 \mid^2 + \mid 16.07-15.718 \mid^2 + \mid 16.11-15.718 \mid^2 + \mid 16.09-15.718 \mid^2}{5}}$$

$$= \sqrt{\frac{0.289444 + 0.334084 + 0.123904 + 0.153664 + 0.138384}{5}} = \sqrt{\frac{1.03948}{5}} = \mathbf{0.455956}$$

Notice how having a C index for all 5 relations is different from having a B index for all 5 relations in a right-deep join tree. It is much faster than indexing on B-value.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql>
mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FRO
M (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) O
N R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |    10.00 | Using where                                         |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R1    | NULL       | ALL  | index_C_R1    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
|  1 | SIMPLE      | R2    | NULL       | ALL  | index_C_R2    | NULL | NULL    | NULL | 1000 |     1.00 | Using where; Using join buffer (hash join)          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

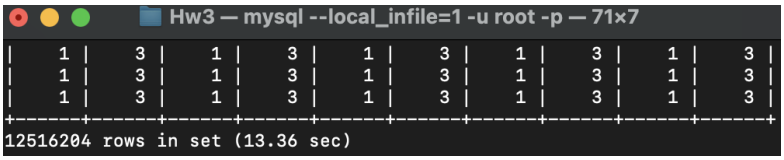
Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND

R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

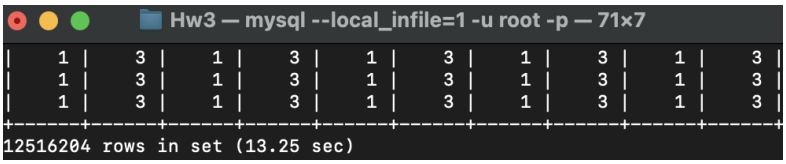
From the output of EXPLAIN, we can see the 2 indexes we made under "possible_keys." They are not shown under "key," so that means they are NOT being used. We can also see in the "Extra" column that the DBMS uses where for the first relation it accessed. Then it uses a hash-join for the rest of the relations. DBMS is using **hash-join** and **index-based join**.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

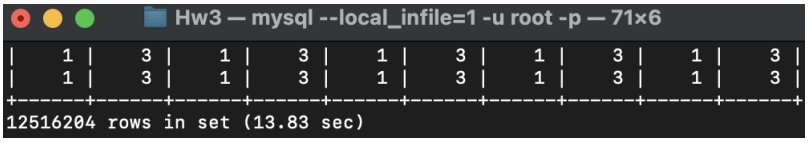
1. Execution Time 1: **13.36 seconds**



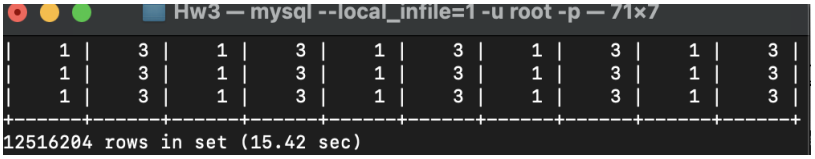
2. Execution Time 2: **13.25 seconds**



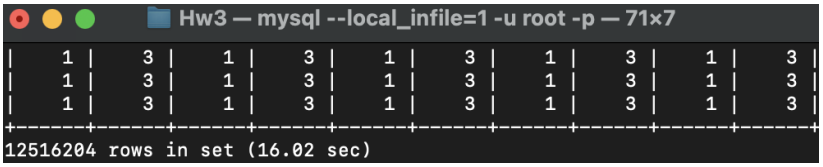
3. Execution Time 3: **13.83 seconds**



4. Execution Time 4: **15.42 seconds**



5. Execution Time 5: **16.02 seconds**



Average = (13.36 + 13.25 + 13.83 + 15.42 + 16.02) / 5 = **14.376 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{|13.36-14.376|^2+|13.25-14.376|^2+|13.83-14.376|^2+|15.42-14.376|^2+|16.02-14.376|^2}{5}}$$

$$= \sqrt{\frac{1.032256+1.267876+0.298116+1.089936+2.702736}{5}} = \sqrt{\frac{6.39092}{5}} = \mathbf{1.130568}$$

Observe how indexing on two relations with C-value is faster than indexing on all 5 relations with C-value. This is also faster than performing the join with no indexes.

3. **Experiment 1.3: Index on B. Then index on C.**

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_B_R3 ON R3 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R4 ON R4 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R5 ON R5 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R3 ON R3 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R4 ON R4 (C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R5 ON R5 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

```
CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);
CREATE INDEX index_B_R3 ON R3 (B);
CREATE INDEX index_B_R4 ON R4 (B);
CREATE INDEX index_B_R5 ON R5 (B);

CREATE INDEX index_C_R1 ON R1 (C);
CREATE INDEX index_C_R2 ON R2 (C);
CREATE INDEX index_C_R3 ON R3 (C);
CREATE INDEX index_C_R4 ON R4 (C);
```

CREATE INDEX index_C_R5 ON R5 (C);

Using DBMS EXPLAIN statement:

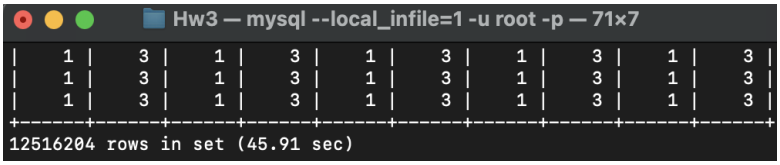
```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R
5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2
.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+-----------------------+-----------+---------+-----------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys         | key       | key_len | ref             | rows | filtered | Extra       |
+----+-------------+-------+------------+------+-----------------------+-----------+---------+-----------------+------+----------+-------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_B_R5,index_C_R5 | index_B_R5 | 5       | const           |   88 |   100.00 | Using where |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_B_R3,index_C_R3 | index_C_R3 | 5       | cs157b_hw3.R5.C |  100 |    17.40 | Using where |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2,index_C_R2 | index_C_R2 | 5       | cs157b_hw3.R5.C |  100 |    20.80 | Using where |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_B_R4,index_C_R4 | index_B_R4 | 5       | const           |  118 |    20.00 | Using where |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1,index_C_R1 | index_C_R1 | 5       | cs157b_hw3.R5.C |  100 |    20.90 | Using where |
+----+-------------+-------+------------+------+-----------------------+-----------+---------+-----------------+------+----------+-------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

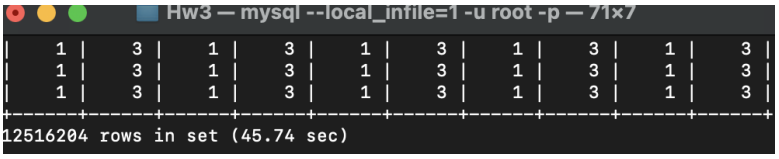
From the output of EXPLAIN, we can see the index we made under “possible_keys” and “key,” so that means they are being used. Under “type,” it says “ref”. We can also see in the “Extra” column that the DBMS uses where for the all relation it accessed. DBMS is using **index-based join**.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

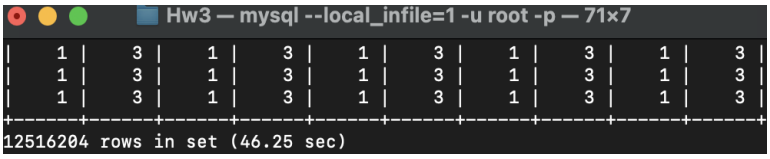
1. Execution Time 1: **45.91 seconds**



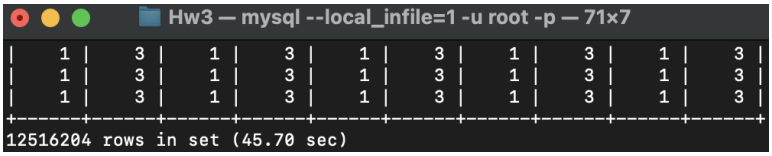
2. Execution Time 2: **45.74 seconds**



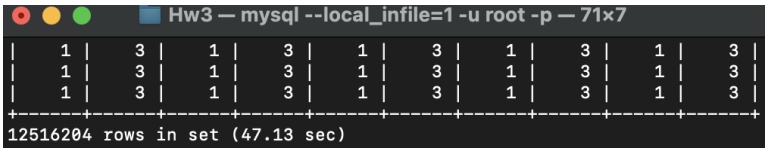
3. Execution Time 3: **46.25 seconds**



4. Execution Time 4: **45.70 seconds**



5. Execution Time 5: **47.13 seconds**



Average = (45.91 + 45.74 + 46.25 + 45.70 + 47.13) / 5 = **46.146 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 45.91 - 46.146 \mid^2 + \mid 45.74 - 46.146 \mid^2 + \mid 46.25 - 46.146 \mid^2 + \mid 45.70 - 46.146 \mid^2 + \mid 47.13 - 46.146 \mid^2}{5}}$$

$$= \sqrt{\frac{0.055696 + 0.164836 + 0.010816 + 0.198916 + 0.968256}{5}} = \sqrt{\frac{1.39852}{5}} = \mathbf{0.528870}$$

The runtime of index B and then index C is still worse than the original runtime without indexes. It is still faster than the runtime with B-value index only.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql>
mysql> CREATE INDEX index_B_R1 ON R1 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_B_R2 ON R2 (B);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R1 ON R1 (C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_C_R2 ON R2 (C);
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

CREATE INDEX index_B_R1 ON R1 (B);
CREATE INDEX index_B_R2 ON R2 (B);

CREATE INDEX index_C_R1 ON R1 (C);

CREATE INDEX index_C_R2 ON R2 (C);

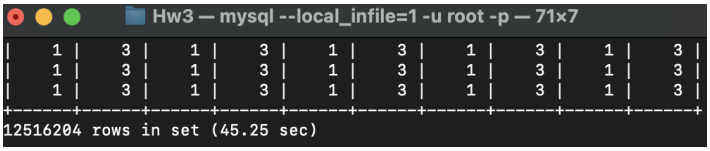
Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 J
OIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+-----------------------+-----------+---------+--------------------+------+----------+-------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys         | key       | key_len | ref                | rows | filtered | Extra                                           |
+----+-------------+-------+------------+------+-----------------------+-----------+---------+--------------------+------+----------+-------------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL                  | NULL      | NULL    | NULL               | 1000 |    10.00 | Using where                                     |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL                  | NULL      | NULL    | NULL               | 1000 |     1.00 | Using where; Using join buffer (hash join)      |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL                  | NULL      | NULL    | NULL               | 1000 |     1.00 | Using where; Using join buffer (hash join)      |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_B_R2,index_C_R2 | index_C_R2 | 5       | cs157b_hw3.R3.C    |  100 |    20.80 | Using where                                     |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_B_R1,index_C_R1 | index_C_R1 | 5       | cs157b_hw3.R3.C    |  100 |    20.90 | Using where                                     |
+----+-------------+-------+------------+------+-----------------------+-----------+---------+--------------------+------+----------+-------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

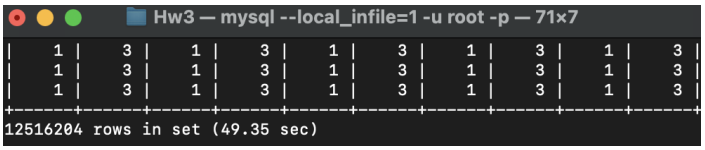
Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);
From the output of EXPLAIN, we can see the 2 indexes we made under “possible_keys” and “key,” so that means they are being used. Under “type,” it says “ref” and “ALL”. We can also see in the “Extra” column that the DBMS uses where for the first relation R3. Then it uses a hash-join for relations that do not have an index. For those with indexes, they are using indexes. DBMS is using **hash-join** and **index-based join**.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

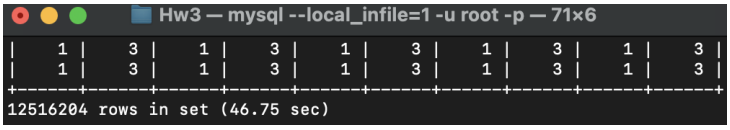
1. Execution Time 1: **45.25 seconds**



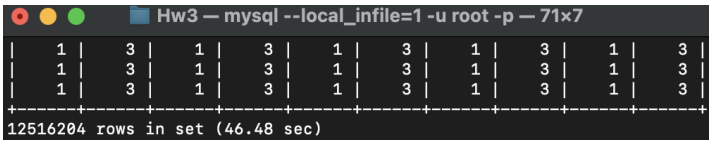
2. Execution Time 2: **49.35 seconds**



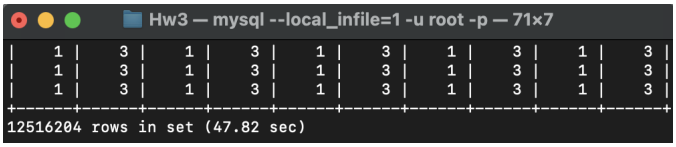
3. Execution Time 3: **46.75 seconds**



4. Execution Time 4: **46.48 seconds**



5. Execution Time 5: **47.82 seconds**



Average = (45.25 + 49.35 + 46.75 + 46.48 + 47.82) / 5 = **47.13 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 45.25 - 47.13 \mid^2 + \mid 49.35 - 47.13 \mid^2 + \mid 46.75 - 47.13 \mid^2 + \mid 46.48 - 47.13 \mid^2 + \mid 47.82 - 47.13 \mid^2}{5}}$$

$$= \sqrt{\frac{3.5344 + 4.9284 + 0.1444 + 0.4225 + 0.4761}{5}} = \sqrt{\frac{9.5058}{5}} = \mathbf{1.378826}$$

The runtime of index B and then index C is still worse than the original runtime without indexes. It is also still worse than indexing on only C value. Compared to the runtime with an index only on B-value, it is still much faster than indexing on B.
Partially applying this index case (B and then C) on two relations is slower than applying indexes on all the relations in this case.

4. Experiment 1.4: Index on (B, C) i.e. composite index of B-value and C-value

Case 1: Create the index on all 5 relations

Create Index:

```
mysql> CREATE INDEX index_BC_R1 ON R1 (B,C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R2 ON R2 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R3 ON R3 (B,C);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R4 ON R4 (B,C);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R5 ON R5 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:

```
CREATE INDEX index_BC_R1 ON R1 (B,C);
CREATE INDEX index_BC_R2 ON R2 (B,C);
CREATE INDEX index_BC_R3 ON R3 (B,C);
CREATE INDEX index_BC_R4 ON R4 (B,C);
```

CREATE INDEX index_BC_R5 ON R5 (B,C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1
JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.
B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+------------+---------+-----------------------+------+----------+-------------------------+
| id | select_type | table | partitions | type | possible_keys | key        | key_len | ref                   | rows | filtered | Extra                   |
+----+-------------+-------+------------+------+---------------+------------+---------+-----------------------+------+----------+-------------------------+
|  1 | SIMPLE      | R5    | NULL       | ref  | index_BC_R5   | index_BC_R5 | 5       | const                 |   88 |   100.00 | Using where; Using index |
|  1 | SIMPLE      | R4    | NULL       | ref  | index_BC_R4   | index_BC_R4 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
|  1 | SIMPLE      | R3    | NULL       | ref  | index_BC_R3   | index_BC_R3 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_BC_R2   | index_BC_R2 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_BC_R1   | index_BC_R1 | 10      | const,cs157b_hw3.R5.C |   20 |   100.00 | Using index             |
+----+-------------+-------+------------+------+---------------+------------+---------+-----------------------+------+----------+-------------------------+
5 rows in set, 1 warning (0.00 sec)
```

Query used:
EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

From the output of EXPLAIN, we can see the index we made under “possible_keys” and “key,” so that means they are being used. Under “type,” it says “ref.” We can also see in the “Extra” column that the DBMS uses where and index for the first relation and then used index for the rest of the relations. DBMS is using **index-based join**.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

1. Execution Time 1: **7.51 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.51 sec)
```

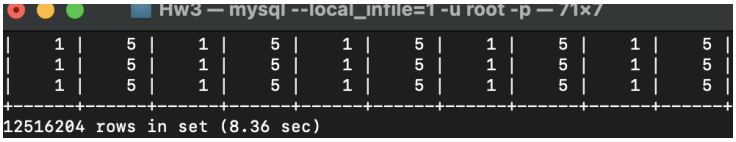
2. Execution Time 2: **7.82 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×7
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.82 sec)
```

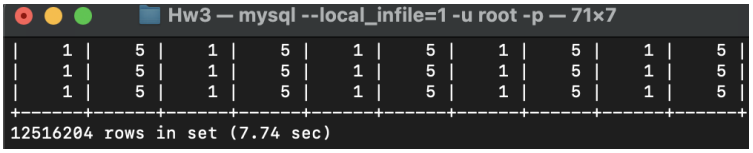
3. Execution Time 3: **7.64 seconds**

```
Hw3 — mysql --local_infile=1 -u root -p — 71×5
|    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |    1 |    5 |
+------+------+------+------+------+------+------+------+------+------+
12516204 rows in set (7.64 sec)
```

4. Execution Time 4: **8.36 seconds**



5. Execution Time 5: **7.74 seconds**



Average = (7.51 + 7.82 + 7.64 + 8.36 + 7.74) / 5 = **7.814 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{\mid 7.51 - 7.814 \mid^2 + \mid 7.82 - 7.814 \mid^2 + \mid 7.64 - 7.814 \mid^2 + \mid 8.36 - 7.814 \mid^2 + \mid 7.74 - 7.814 \mid^2}{5}}$$

$$= \sqrt{\frac{0.092416 + 0.000036 + 0.030276 + 0.298116 + 0.005476}{5}} = \sqrt{\frac{0.42632}{5}} = \mathbf{0.292}$$

The runtime of the index (B, C) on all relations had a good time speed. It is faster than the original runtime, the runtime of B-values only index, the runtime of C-values only index, and the runtime with index B and then index C.

Case 2: Create the index on 2 relations only

Create Index:

```
mysql> CREATE INDEX index_BC_R1 ON R1 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX index_BC_R2 ON R2 (B,C);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query used:
CREATE INDEX index_BC_R1 ON R1 (B,C);
CREATE INDEX index_BC_R2 ON R2 (B,C);

Using DBMS EXPLAIN statement:

```
mysql> EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3
JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);
+----+-------------+-------+------------+------+---------------+---------------+---------+-------------------------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key           | key_len | ref                     | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+---------------+---------+-------------------------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | R3    | NULL       | ALL  | NULL          | NULL          | NULL    | NULL                    | 1000 |    10.00 | Using where                                        |
|  1 | SIMPLE      | R4    | NULL       | ALL  | NULL          | NULL          | NULL    | NULL                    | 1000 |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R5    | NULL       | ALL  | NULL          | NULL          | NULL    | NULL                    | 1000 |     1.00 | Using where; Using join buffer (hash join)         |
|  1 | SIMPLE      | R2    | NULL       | ref  | index_BC_R2   | index_BC_R2   | 10      | const,cs157b_hw3.R3.C   |   20 |   100.00 | Using index                                        |
|  1 | SIMPLE      | R1    | NULL       | ref  | index_BC_R1   | index_BC_R1   | 10      | const,cs157b_hw3.R3.C   |   20 |   100.00 | Using index                                        |
+----+-------------+-------+------------+------+---------------+---------------+---------+-------------------------+------+----------+----------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

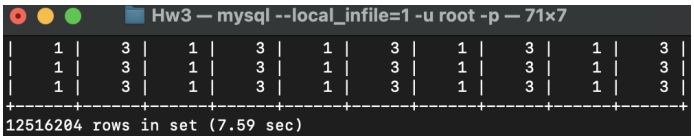
Query used:

EXPLAIN SELECT R1.B AS R1B, R1.C R1C, R2.B AS R2B, R2.C AS R2C, R3.B AS R3B, R3.C AS R3C, R4.B AS R4B, R4.C AS R4C, R5.B AS R5B, R5.C AS R5C FROM (R1 JOIN (R2 JOIN (R3 JOIN (R4 JOIN R5 ON R4.B = R5.B AND R4.C = R5.C AND R5.B = 1) ON R3.B = R4.B AND R3.C = R4.C) ON R2.B = R3.B AND R2.C = R3.C) ON R1.B = R2.B AND R1.C = R2.C);

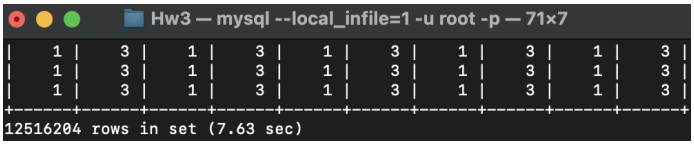
From the output of EXPLAIN, we can see the 2 indexes we made under "possible_keys" and "key," so that means they are being used. We can also see in the "Extra" column that the DBMS uses where for the first relation R3. Then it uses a hash-join for relations that do not have an index. For those with indexes, they are using indexes. DBMS is using **hash-join** and **index-based join**.

Execute Right-deep Join Query 5 times. Compute the Average and Standard Deviation:

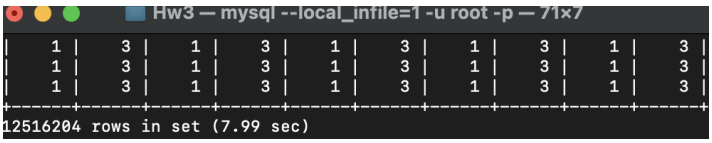
1. Execution Time 1: **7.59 seconds**



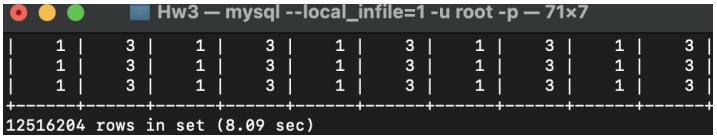
2. Execution Time 2: **7.63 seconds**



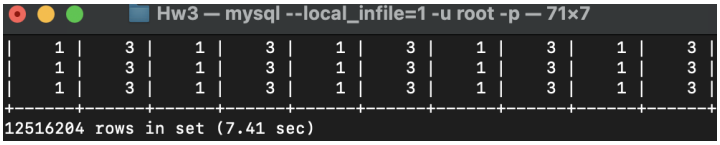
3. Execution Time 3: **7.99 seconds**



4. Execution Time 4: **8.09 seconds**



5. Execution Time 5: **7.41 seconds**



Average = (7.59 + 7.63 + 7.99 + 8.09 + 7.41) / 5 = **7.742 seconds**

Standard deviation = $\sqrt{\frac{\Sigma \mid x - \mu \mid^2}{N}}$

$$= \sqrt{\frac{|7.59 - 7.742|^2 + |7.63 - 7.742|^2 + |7.99 - 7.742|^2 + |8.09 - 7.742|^2 + |7.41 - 7.742|^2}{5}}$$

$$= \sqrt{\frac{0.023104 + 0.012544 + 0.061504 + 0.121104 + 0.110224}{5}} = \sqrt{\frac{0.32848}{5}} = \mathbf{0.256312}$$

The runtime of the index (B, C) on two relations also has a good time speed. It is faster than the original runtime, the runtime of B-values only index, the runtime of C-values only index, and the runtime with index B and then index C. It is a little faster than the runtime for with (B, C) on all relations.

Experiment 3 Conclusion:

Similar to Experiments 1 and 2, having indexes does not necessarily mean it will run faster than relations that do not have indexes. Indexing on a composite of B and C is faster.

Without an index, DBMS just uses hash-join.

Average without Index: 14.562 seconds
SD: 0.223553
Average with Composite Index B,C): 7.814 seconds
SD: 0.292

==========

Overall Results

Below are tables representing the average runtime (in seconds) for the joins based on the experiments.

Without Indexes:

Type of Join	Average w/o Index	Standard deviation w/o Indexes
Left-deep Join	14.64	0.214009
Bushy	14.67	0.515636
Right-deep Join	14.562	0.223553

With Indexes Applied on all 5 Relations:

Indexing on… →	B-value only		C-value only		B-value then C-value		Composite Index (B,C)	
Type of Join	AVG	SD	AVG	SD	AVG	SD	AVG	SD
Left-deep	82.878	0.690895	14.508	0.244336	45.714	1.945133	7.596	0.287305
Bushy	86.272	4.487273	14.918	0.505149	45.706	1.796804	8.04	0.560393
Right-deep	91.666	4.880986	15.718	0.455956	46.146	0.528870	7.814	0.292

With Indexes Applied on 2 of 5 Relations:

Indexing on… →	B-value only		C-value only		B-value then C-value		Composite Index (B,C)	
Type of Join	AVG	SD	AVG	SD	AVG	SD	AVG	SD
Left-deep	15.794	0.645804	13.882	0.692168	44.58	3.279134	8.424	0.863101
Bushy	15.596	0.547087	13.684	0.860153	47.186	1.701686	8.238	0.777725
Right-deep	15.3	0.283267	14.376	1.130568	47.13	1.378826	7.742	0.256312

Overall Conclusion

Highlights:

- Without the indexes, all joins have almost the same runtime.
- Index on B-value alone makes the runtime much slower for 2 reasons: first, having an index eliminates the use of hash-join (building the hash table in memory if fit). Secondly, the size of the index entry in this case is as big as the size of the data record. Reading from the index table and then following the record pointer to load the data from the original table for C-value would add extra overhead.
- Out of all the indexes, the composite index manages to improve the query time because both B and C in the query conditions can be obtained directly from the index table.

My hypothesis is partially right. (Short version of hypothesis: I thought the runtime will be about the same. I also thought indexing would, overall, improve the runtime of the queries.
For more detail, please look up the “Hypothesis” that is bolded and underlined)

Overall, based on the experiments, the runtime among the three joins is almost the same. The bushy tree is a little slower than the left-deep join tree and right-deep join tree, but not far off.

However, indexing does not always mean improving query time. It depends on the condition of the query as well as the attribute(s) being used in the index. For instance, indexing on B-value only will reduce the query performance compared to the query performance without any index. Meanwhile composite index on B and C helps improve the query performance.

Overall Plan Check:
Without indexes, the plan for each join query is the same. They all use where for one relation and then use where and the **hash-join** algorithm for the rest.

With indexes, the plan for each join query is also the same. The plan often tends to use **hash-join** for relations without indexes and **index-based join** for those with indexes.

Note: There is a possibility that the reason no nested loop was found throughout the experiments is that the computer's memory (16 GB 2133 MHz LPDDR3 Memory) is large enough to build a hash for the entire table.