Contents

Project Phase 2 - Join Algorithms and Group By **1**
Task 1 - Implementing join algorithms . . . 1
Block-nested Join . . . 1
Partition Hash-Join . . . 2
Example . . . 2
Task 2 - GROUP BY - AGGREGATES . . . 4
Syntax . . . 4
Example . . . 4

Project Phase 2 - Join Algorithms and Group By

Deadline - 11:59 PM, 21st November 2021

This phase consists of two tasks: (1) Join algorithms and (2) Group By aggregate operator.

Task 1 - Implementing join algorithms

You are required to implement two algorithms for the `JOIN` operation. The algorithms assume the following notation: $R \bowtie_{A=B} S$, where A and B are valid join attributes, of R and S respectively.

Block-nested Join

In the simple nested join algorithm, for each record t from R (outer loop), we scan through all the records s from S (inner loop) and check if the join condition ($t[A] = s[B]$) is satisfied by the two records. For this project, you have to implement a block-nested join algorithm, which is an optimization of the simple nested join algorithm that involves reading several blocks from the first relation R at a time, therefore scanning S only once for every group of R tuples.

For implementation, we assume the following syntax to be strictly followed.

```
<new_relation_name> <- JOIN USING NESTED <table1>, <table2> ON <column1
    > <bin_op> <column2> BUFFER <buffer_size>
```

Where is the number of blocks available in main memory that you can use to read blocks of relation R. If nB is the buffer size, $nB - 2$ blocks of the first relation ($R) should be read in and processed.

For example:

```
RS <- JOIN USING NESTED R, S ON A == B BUFFER 5
```

Partition Hash-Join

Each file is first split into M partitions using the same hash function on the join attributes. Then each pair of corresponding partitions is joined.

In the partitioning phase, R and S both are split into M partitions, with these partitions $(R_i^{'s}, S_i^{'s})$ following the property with respect to the join operation that records in R_i only need to be joined to records in S_i and vice versa. During partitioning of a file, M in-memory buffers are allocated to store the records that hash to each partition and one additional buffer is needed to hold one block at a time of the input file. Note that in case of an overflow for an in-memory buffer, the remaining content is appended to a disk subfile that stores the partition.

The two iterations involved in the algorithm are:

1. Partitioning: R and S are split into M partitions.
2. Joining: Each of the i^{th} partitions is joined, thus this goes on for M iterations (number of buckets).

If we use the nested loop join, the smaller of the two partitions, say R_i is loaded into the main memory, and all blocks from the other partition S_i are read one at a time, wherein each record is used to probe partition R_i for matches. These matches are joined and written into a result file. Note to increase the performance of finding matches, it is common to use an in-memory hash table for storing records in partition R_i, by employing a different hash function.

For implementation, we assume the following syntax to be strictly followed.

```
<new_relation_name> <- JOIN USING PARTHASH <table1>, <table2> ON <
    column1> <bin_op> <column2> BUFFER <buffer_size>
```

Where is the number of blocks available in main memory that you can use to read blocks of relation R. If nB is the buffer size, the relations can be split into $nB - 1$ partitions.

For example:

```
RS <- JOIN USING PARTHASH R, S ON A == B BUFFER 5
```

Example

Consider a STUDENT table:

ID	Age
1	18
2	21
3	23
4	17
5	26

Consider a COURSE table:

Course_ID	Roll_Number
1	1
2	1
3	2
6	3
5	4
7	4
2	5

If the query to be run is to list all the students along with the courses they are enrolled in, then the statement would be:

```
Result <- JOIN USING NESTED STUDENT, COURSE ON ID == Roll_Number
```

The Result table in this example would be:

ID	Age	Course_ID	Roll_Number
1	18	1	1
1	18	2	1
2	21	3	2
3	23	6	3
4	17	5	4

ID	Age	Course_ID	Roll_Number
4	17	7	4
5	26	2	5

The Result table must contain all the columns from both the tables. For the sake of this assignment, you can assume that both tables will never contain columns with matching names. The order of the output rows doesn't matter, but the order of the columns must be columns from the first table followed by columns from the second table.

Task 2 - GROUP BY - AGGREGATES

Another query you will have to implement is the `GROUP BY` query along with aggregate operators `MAX`, `MIN`, `SUM`, `AVG` where these aggregate operators take on their usual meaning. For those of you unfamiliar with the `GROUP BY` query, this query clusters the rows of a table based on the attribute value of the grouping attribute, and a query result is returned for each "group" or supposed element in the grouping attribute's domain

Syntax

The syntax of the `GROUP BY` statement is as follows

```
<new_table> <- GROUP BY <grouping_attribute> FROM <table_name> RETURN
   MAX|MIN|SUM|AVG(<attribute>)
```

Example

For example, if you loaded a table `R` with the following contents

```
A, B, C
1, 2, 3
1, 4, 6
1, 6, 12
1, 8, 15
2, 2, 18
2, 4, 21
2, 6, 24
2, 8, 27
```

The expected output of executing `GROUP BY` command is detailed below

```
T1 <- GROUP BY A FROM R RETURN MAX(C)
```

New created table `T1` contains two columns `(A, MAXC)` with the following rows

The output column name will be the aggregate operator concatenated with column name it's operating on

```
MAX|MIN|SUM|AVG(X) -> MAX|MIN|SUM|AVGX
```

So, `MAX(C)-> MAXC`

```
A, MAXC
1, 15
2, 27
```

The following query produces `T2`

```
T2 <- GROUP BY C FROM R RETURN SUM(B)
```

Where `T2` has 2 columns `(C, SUMB)` with the following rows

```
C, SUMB
3, 2
6, 4
12, 6
15, 8
18, 2
21, 4
24, 6
27, 8
```

The following query produces `T3`

```
T3 <- GROUP BY A FROM R RETURN MIN(A)
```

Where `T3` has 2 columns `(A, MINA)` with the following rows

```
A, MINA
1, 1
2, 2
```

For both tasks,

- You have to implement syntactic and semantic checks
- You will be graded on the correctness of your implementation
- In both parts, you are required to output the number of block accesses onto the terminal.
- For Block Nested Join: For each block from the relation R, we would be loading all the blocks from the relation S for the worst case and if the entire Relation S fits in the main memory along with an extra block space to load each block of R (and another to hold/write the result back), this would become our best case scenario.
- For Partition Hash Join: Given in the description.
- For Group By: We load the whole relation in the memory.

This course is intolerant of plagiarism. Any plagiarism will lead to an F in the course.