

**CZ4031 Database System Principles**

**Project 2**

|  |  |  |
| --- | --- | --- |
| Name | Matric No. | Email |
| Keith Tay Tat Leong | U1321940J | ktay014@e.ntu.edu.sg |
| Tan Ser Han | U1321964D | c130140@e.ntu.edu.sg |
| Nguyen Ngoc Tram Anh | U1221056H | tramanh001@e.ntu.edu.sg |

# Merge Sort

# Merge sort is a comparison-based algorithm for sorting. The idea of it is to provide an efficient and effective means to sort a collection of items as quickly as possible with a reduced amount of IO cost by leveraging on the divide and conquer approach.

# There are two phases in the merge sort algorithm; sorting tuples into different sublists and then merging them back together. To perform merge sort, the basic requirement must be met: , where B(R) is number of blocks needed to hold tuples of R, and M is available memory buffer allocated for the join operation. This means the total number of sublist must be less than the total memory size – 1, while 1 buffer is to hold output buffer. The IO cost is computed based on:

# An initial loading of all blocks into memory

# Writing back sublists to the disk after sorting

# Reading the blocks for merging.

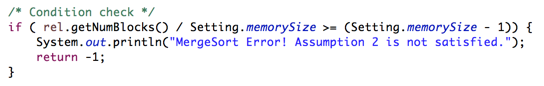
# Therefore, a total of three IOs per block are required.

# 1.1 Code Analysis

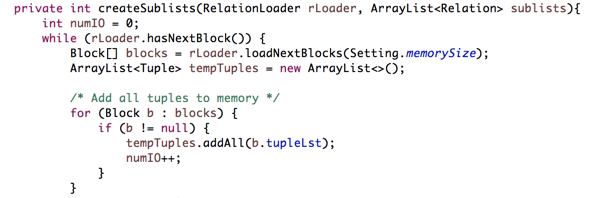
Below are the screenshot snippets and accompanying explanation for each phase:

##### Phase 1 – Generating Sorted sublists:

Step 0: Before the algorithm starts, Merge Sort assumption is checked, i.e. . If the condition does not meet, method will return -1 for numIO.



**Step 1:** As the entire relation is stored in blocks, the first step is to extract tuples from each block using M memory buffers. Every block processed incurs a read IO access

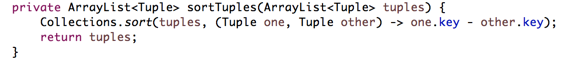


**Step 2:** The tuples then undergo sorting based on the comparison algorithm below

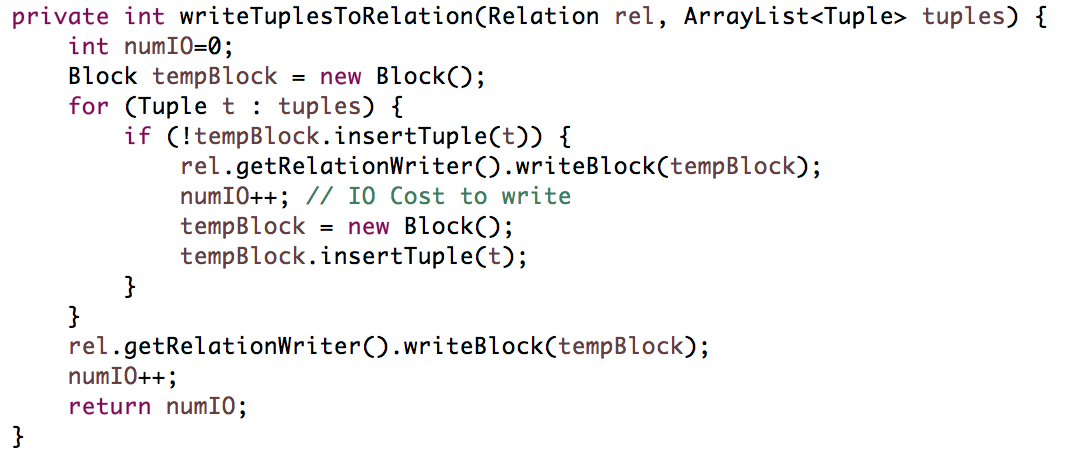
In createSublists(), it calls a method sortTuples()



Detailed Implementation of sortTuples():

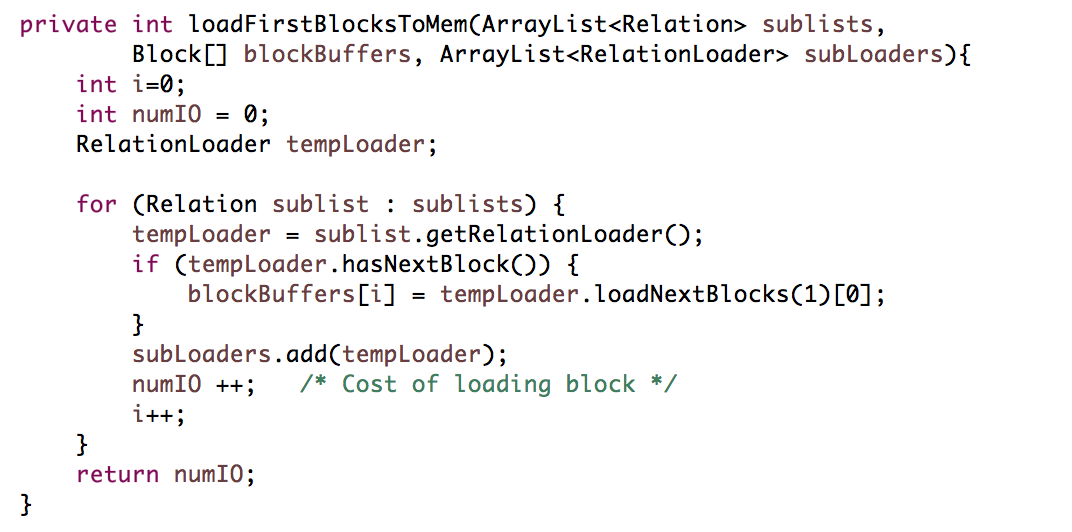


**Step 3:** The sorted tuples are then filled into a buffer and written back into sublists (which data structure is Relation in the code) after the block buffer is completely filled. A write IO access is required on every new block written back. We write this in a separate function *writeTuplesToRelation()* for future code reuse.

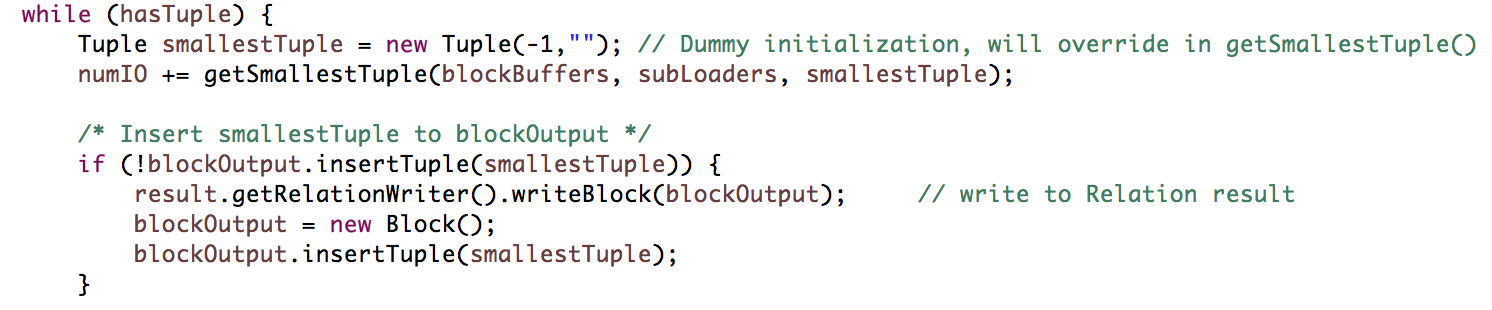


##### Phase 2 – Sorting and Merging Two Relations:

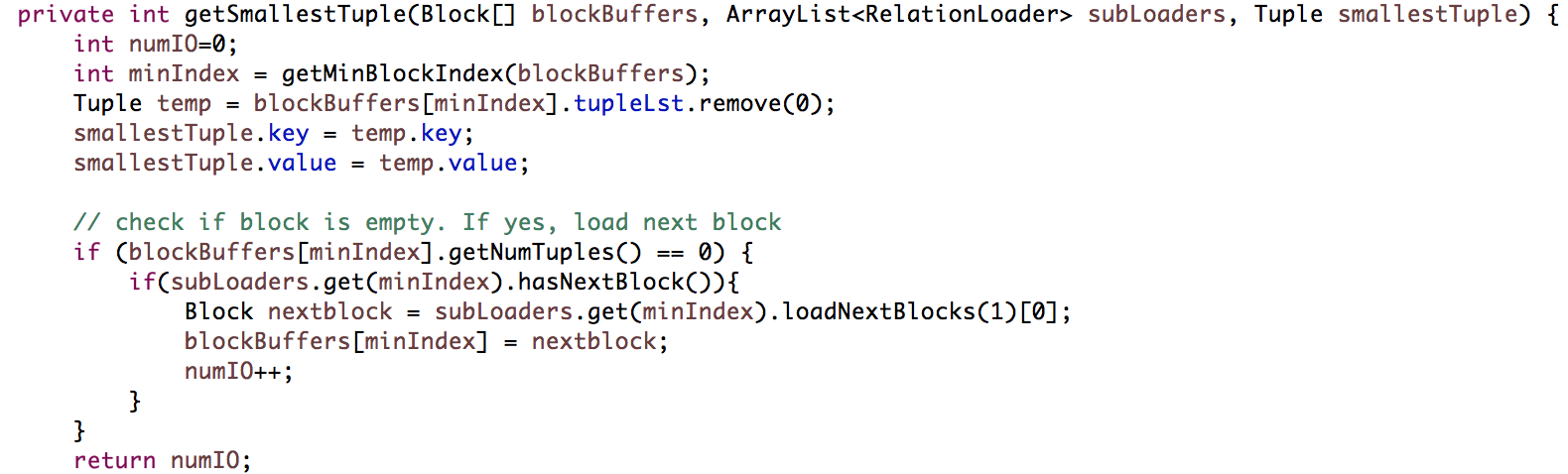
**Step 4:** For the second phase, first we need to load the first block for each sublists into memory. This is done via *loadFirstBlocksToMem()* method. For each sublist, we load 1 block, and add that block to an array called blockBuffers, that mimics the block buffers in memory. We also add the current state of each sublist’s RelationLoader and return them to the main method, so we know where we last read the block.



**Step 5:** After loading the first block from each sublist to the memory, we commence merging process. Below is the main steps:



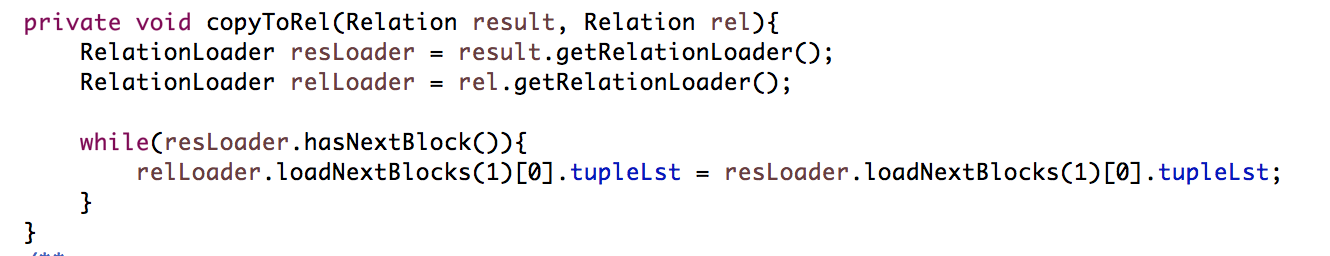
Note that we call a method *getSmallestTuple()* which is supposed to get the smallest tuple across all blockBuffers in memory, and also delete that tuple from the memory. Below is the code for *getSmallestTuple()*:



**Step 6:** Since we store all the sorted tuples into a temporary Relation called result, after we exit the while loop, the last *blockOutput* is written to result.



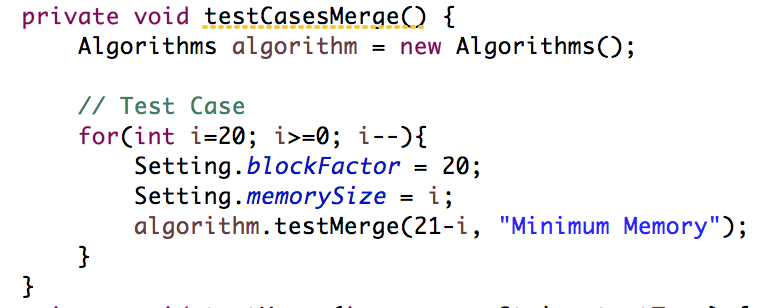
**Step 7:** Final step is to copy Relation result to Relation rel. The trick is to copy each block’s tupleLst to rel’s block tupleLst.



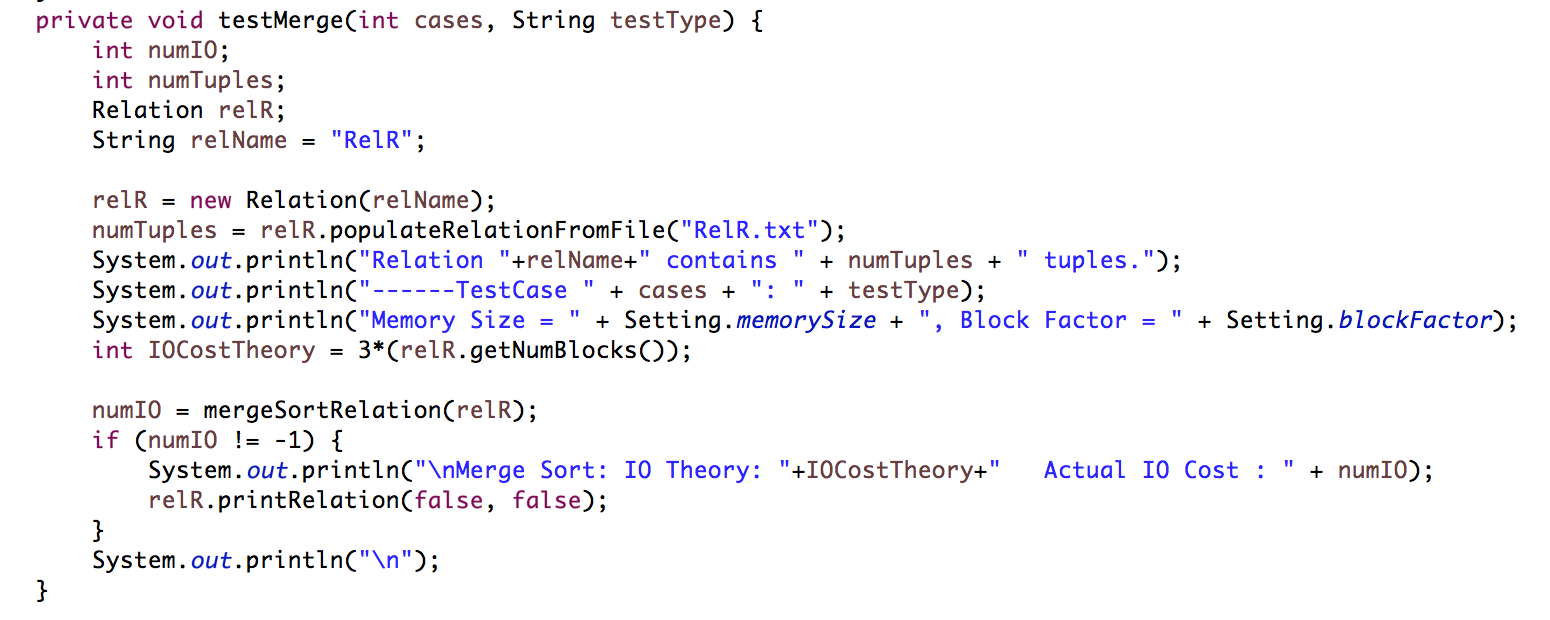
Note that the act of writing blocks to rel does not need to count for IO Cost because the algorithm assumes that the output is printed on the fly, as it’s generated.

## 1.3 Test Case Implementation

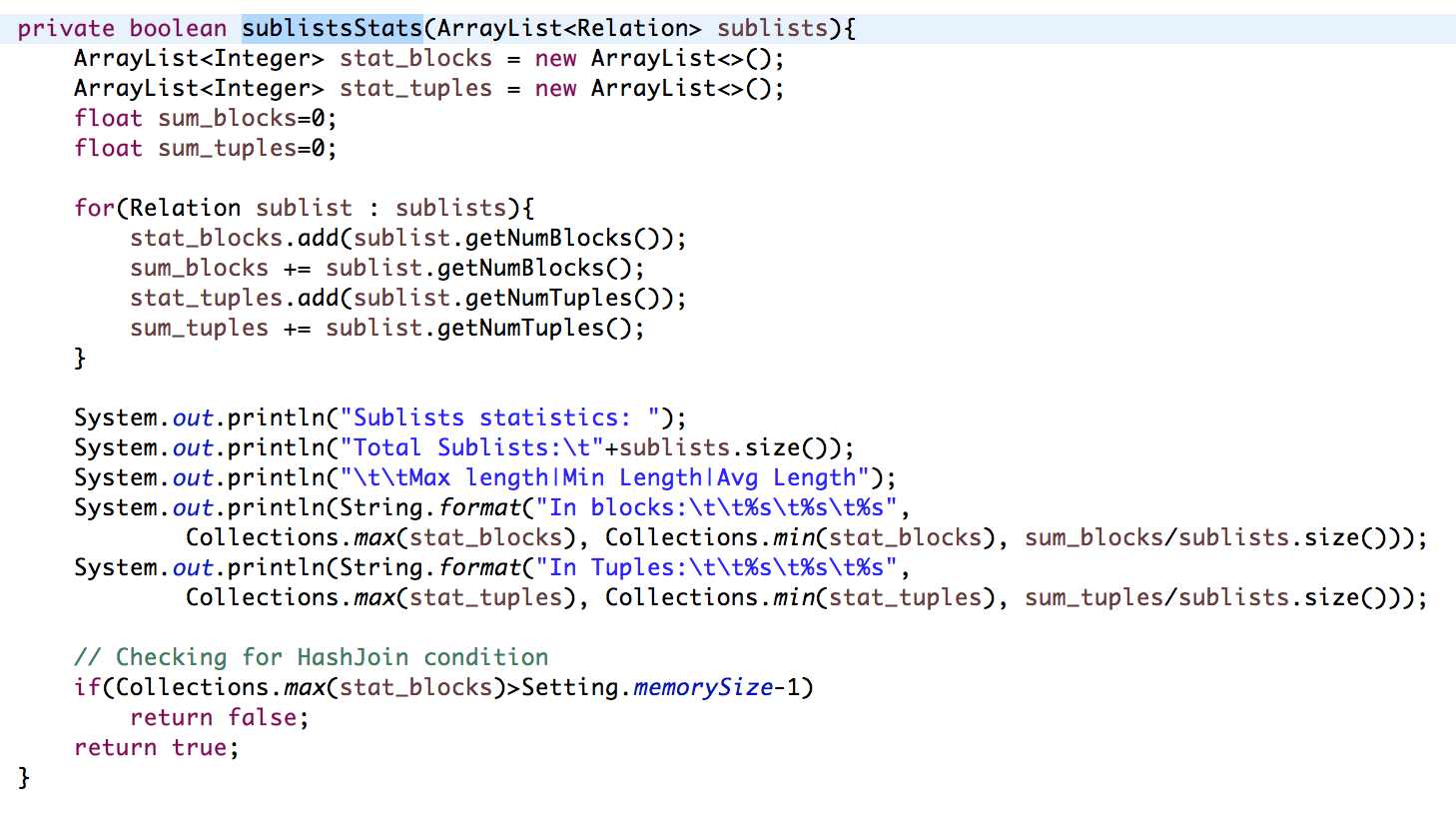
To test for varying sizes of memory and block factor, the general approach is to fix one variable and vary the other. To repeatedly test for different setting, we run a for loop that calls our test function. The code snippet is shown below:



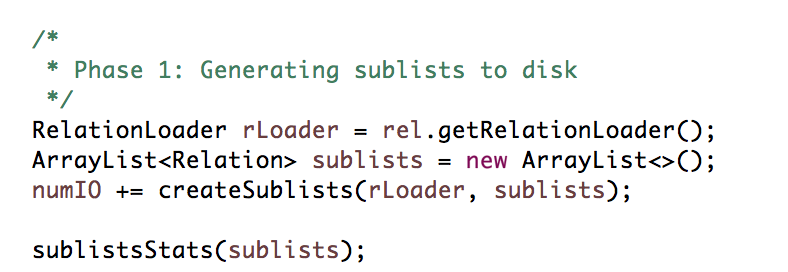
The test function is just the normal implementation of Merge Sort.



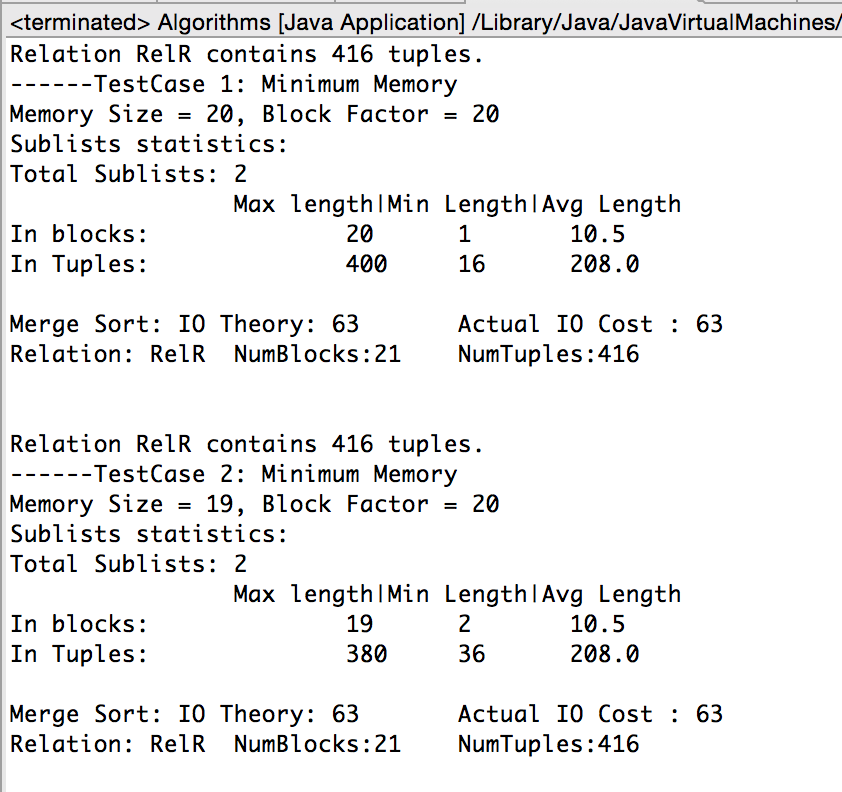
As part of the requirement for testing, statistics for sublists and buckets are also printed during the test. Below is the function to print out the statistics:



The function takes in an *ArrayList* of sublists, then generate statistics info from them. This function is reused for all 3 algorithms, right after sublists are created. In *MergeSortRelation()*, this is how *sublistsStats()* is used:



This is the output of the test cases where we fix blockFactor to be 20.



Since the implementation for testing of RefinedSortMergeJoin and HashJoin is similar, please refer to our code in soft submission for RefinedSortMergeJoin and HashJoin.

## 1.4 Test Case Analysis

All tests on the merge sort algorithm were performed on only Relation R. SQL output on dataset with merge sort algorithm output is shown in the Appendix.

The merge sort algorithm minimally requires at least two memory buffers for the operation as one buffer would be allocated for the merge process. A few tests were conducted to analyze the correlation between the block factor and the number of memory buffers required for each process. The block factor determines how many tuples can be fitted into a block and thus, would influence the number of IO count used.

###### Finding the Minimum Block Factor Based On Fixed Memory

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Min Block Factor Based on Memory | | | | | | |
| Memory Size | Block Factor | Actual IO | Number of sublists | Sublists Average Length (blocks) | Sublists Average length (Tuples) |
| 20 | 20 | 63 | 2 | 10.5 | 208 |
| 20 | 19 | 66 | 2 | 11 | 208 |
| 20 | 18 | 72 | 2 | 12 | 208 |
| 20 | 17 | 75 | 2 | 12.5 | 208 |
| 20 | 16 | 78 | 2 | 13 | 208 |
| 20 | 15 | 84 | 2 | 14 | 208 |
| 20 | 14 | 90 | 2 | 15 | 208 |
| 20 | 13 | 96 | 2 | 16 | 208 |
| 20 | 12 | 105 | 2 | 17.5 | 208 |
| 20 | 11 | 114 | 2 | 19 | 208 |
| 20 | 10 | 126 | 3 | 14 | 138.67 |
| 20 | 9 | 141 | 3 | 15.67 | 138.67 |
| 20 | 8 | 156 | 3 | 17.34 | 138.67 |
| 20 | 7 | 180 | 3 | 20 | 138.67 |
| 20 | 6 | 210 | 4 | 17.5 | 104 |
| 20 | 5 | 252 | 5 | 16.8 | 83.2 |
| 20 | 4 | 312 | 6 | 17.34 | 69.33 |
| 20 | 3 | 417 | 7 | 19.86 | 59.43 |
| 20 | 2 | 624 | 11 | 18.9 | 37.8 |
| 20 | 1 | FAIL | FAIL | FAIL | FAIL |
| 20 | 0 | FAIL | FAIL | FAIL | FAIL |

From the table shown above, we can see that as the block factor decreases, the IO cost gradually increases as more blocks are required to hold the relation. However when the block factor reaches 1, the number of sublists will exceed the number of memory buffers available and the merge will fail.

###### Finding the Minimum Memory Based On Fixed Block Factor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Min Memory Based on Block Factor** | | | | | |
| **Memory Size** | **Block Factor** | **Actual IO** | **Number of sublists** | **Sublists Average Length (blocks)** | **Sublists Average length (tuples)** |
| 20 | 20 | 63 | 2 | 10.5 | 208 |
| 19 | 20 | 63 | 2 | 10.5 | 208 |
| 18 | 20 | 63 | 2 | 10.5 | 208 |
| 17 | 20 | 63 | 2 | 10.5 | 208 |
| 16 | 20 | 63 | 2 | 10.5 | 208 |
| 15 | 20 | 63 | 2 | 10.5 | 208 |
| 14 | 20 | 63 | 2 | 10.5 | 208 |
| 13 | 20 | 63 | 2 | 10.5 | 208 |
| 12 | 20 | 63 | 2 | 10.5 | 208 |
| 11 | 20 | 63 | 2 | 10.5 | 208 |
| 10 | 20 | 63 | 2 | 7 | 138.67 |
| 9 | 20 | 63 | 3 | 7 | 138.67 |
| 8 | 20 | 63 | 3 | 7 | 138.67 |
| 7 | 20 | 63 | 3 | 7 | 138.67 |
| 6 | 20 | 63 | 4 | 5.25 | 104 |
| 5 | 20 | FAIL | FAIL | FAIL | FAIL |
| 4 | 20 | FAIL | FAIL | FAIL | FAIL |
| 3 | 20 | FAIL | FAIL | FAIL | FAIL |
| 2 | 20 | FAIL | FAIL | FAIL | FAIL |
| 1 | 20 | FAIL | FAIL | FAIL | FAIL |
| 0 | 20 | FAIL | FAIL | FAIL | FAIL |

We can see from the table above that the amount of memory does not affect the number of IOs as the amount of memory used depends on the number of sublists, which itself, is determined by the block factor. When the memory falls below the minimum of 6 buffers, processing will fail completely. This is because number of sublists generated is more than M-1.

# Refined Sort Merge Join

The idea of refined sort merge join is to sort and merge two relations at the same time so as to reduce the number of total IOs. This is in comparison to first sorting each relation, then merge.

The memory buffer required is much larger than merge sort, as the memory must be big enough to hold both relations, it is roughly *(B(R) + B(S)) <= M2*, but the total number of sublists of both relations must not exceed *M-1* condition for the algorithm to successfully work in phase 2. The number of IOs are computed by:

1. Reading from the relation for sorting the sublist
2. Writing back sublists to the disk
3. Reading the blocks for merging and sorting

A total of 3 IOs are required for each block of Relation **R** and **S**.

## 2.1 Code Analysis

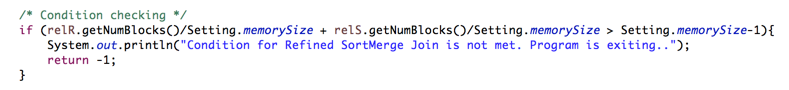
Below are the screenshots and code analysis for each individual phase:

##### Phase 1 – Generating Sorted sublist

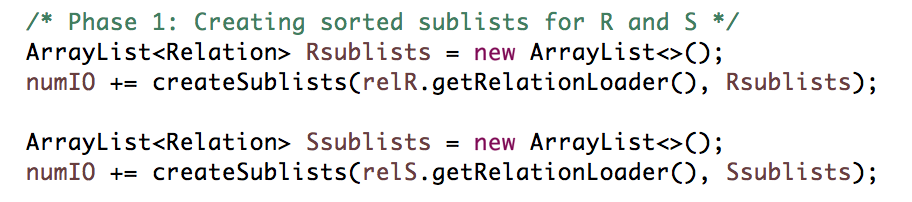
**Step 0:** First we check if the assumption for Refined Sort Merge Join is met, that is

*(B(R)/M + B(S)/M <= M-1)*

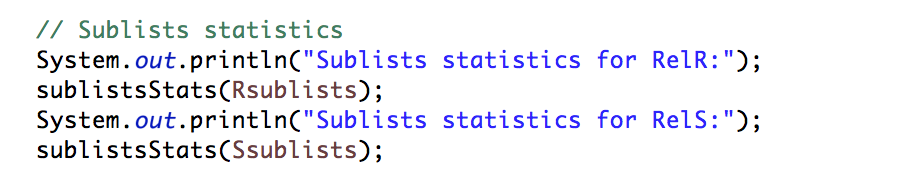
Note that the condition here is stricter than just *(B(R) + B(S)) <= M2*because we will be testing with very small *M*, hence *(M2-M)* and *M2* make a difference.



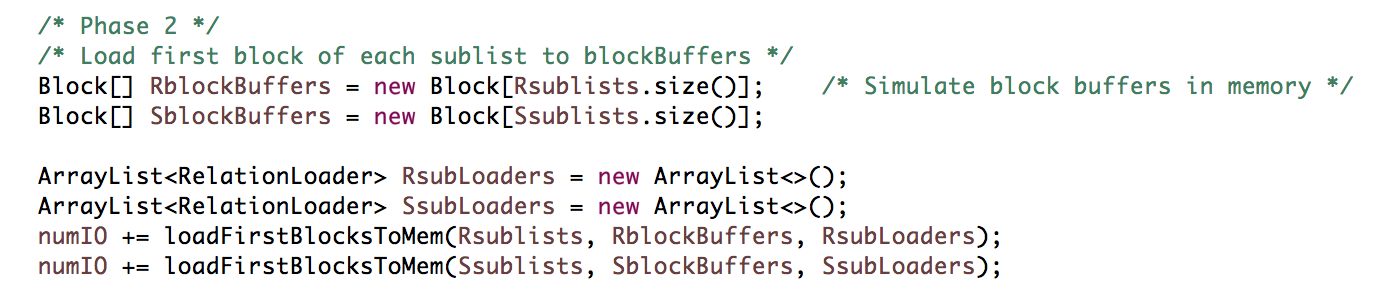
**Step 1:** We reuse the function *createSublists()* from *mergeSortRelation()* to generate corresponding sublists for relation **R** and **S**



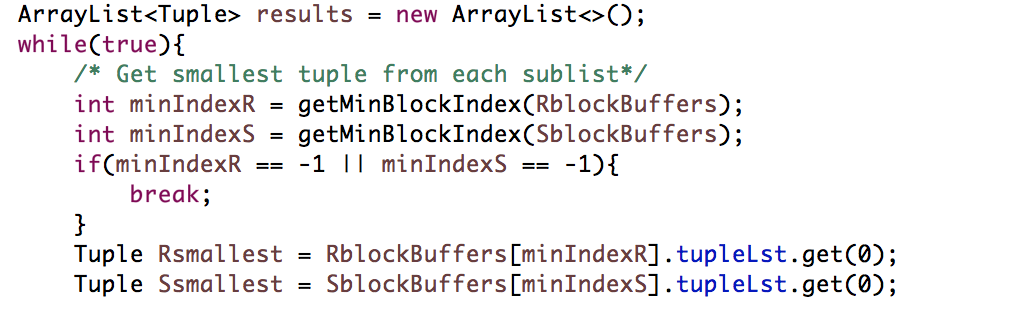
For testing purposes, sublists statistics are run as well



**Step 2:** Now we enter phase 2. The first half of phase 2 is similar to *mergeSortRelation()* where we the first block of each sublist is loaded into memory. Therefore, we can reuse *loadFirstBlocksToMem()* in *mergeSort* (step 5 in *mergeSort* algorithm)

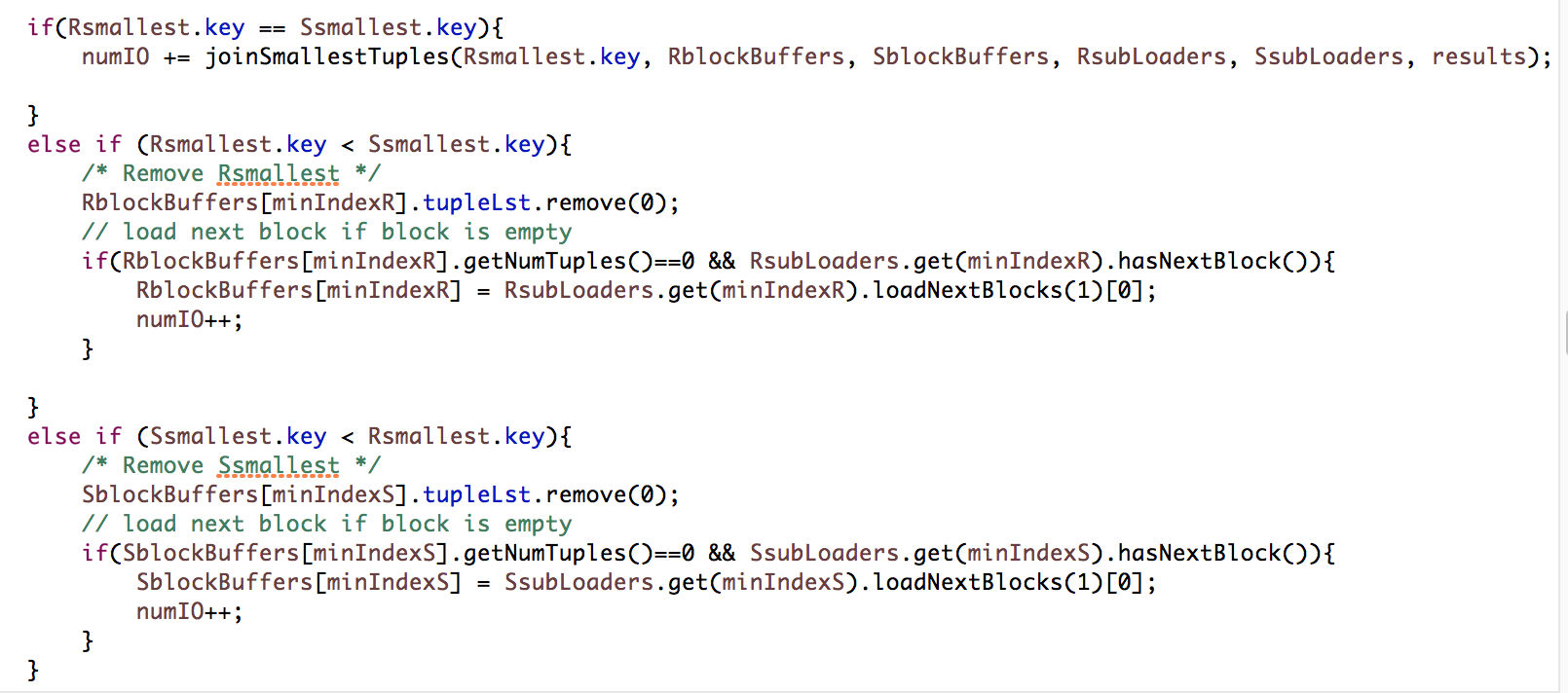


**Step 3:** Now that we have all first blocks of relR and relS, we will extract the smallest tuple for each relR and relS in memory.



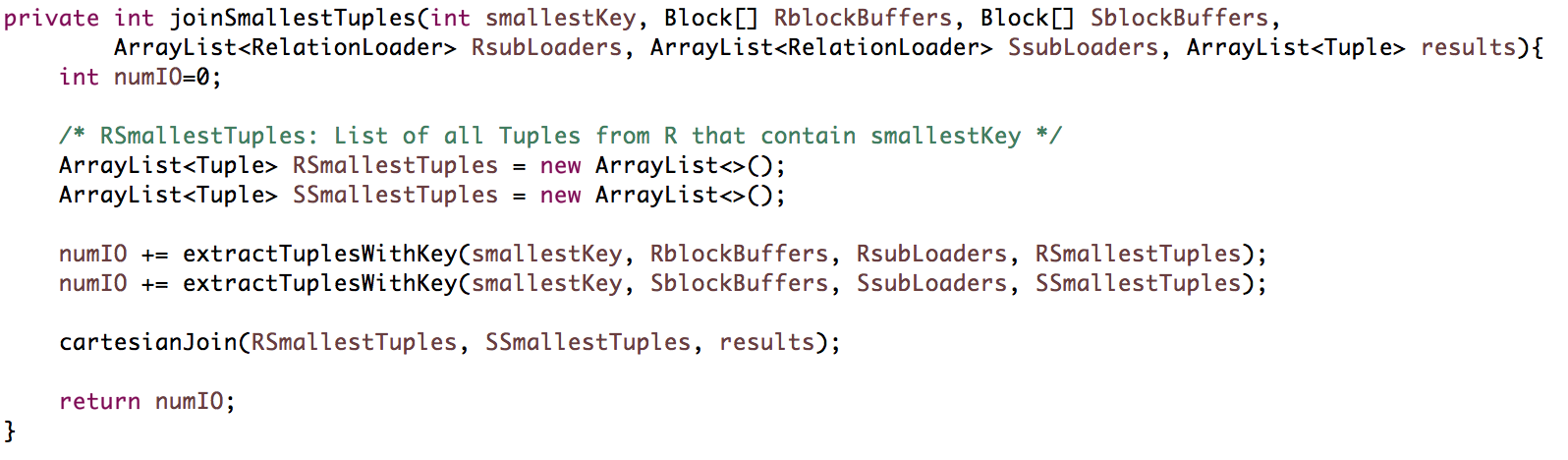
*getMinBlockIndex()* will return the block index corresponding to the current *blockBuffers* in memory. Note that the while loop will break when *getMinBlockIndex(blockBuffers)* returns -1, which happens when there is no tuple present in *blockBuffers* anymore. Algorithm stops when one of the relation is empty.

**Step 4:** Now that we have smallest tuple for each relation, we will compare them. If they are equal, it means we have a match. If not, we will discard the smaller tuple from memory and continue the while loop. Note that when we remove the smallest tuple from memory, we need to check if the block is empty. If it is and there is next block, we load the next block.



Note that a huge chuck of code is dedicated in *joinSmallestTuples()*, so we shall explain it in the next step.

**Step 5:** In *joinSmallestTuples()*, the main algorithm is to find all tuples from **R** and **S** that has the same smallest key. After that, do a Cartesian Join for these tuples, and save them to Relation results. The code in *joinSmallestTuples()* is pretty self-explanatory.

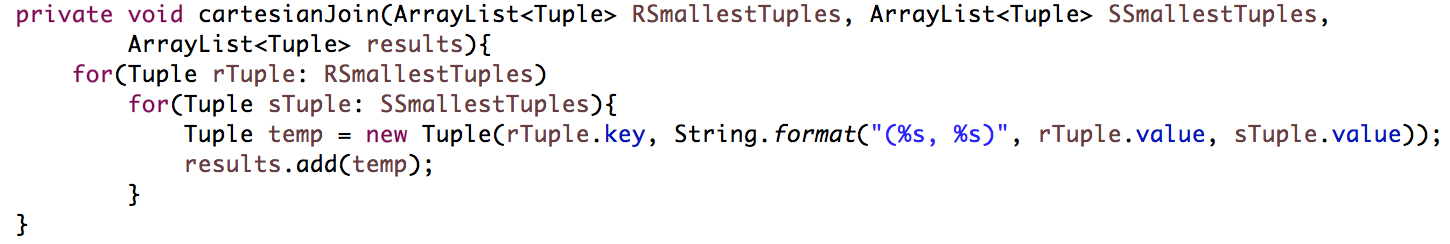


*extractTuplesWithKey()* is to find all tuples from the relation, loading next block to memory if necessary. We then store all the tuples into an *ArrayList* of *Tuple*, called *RSmallestTuples* or *SSmallestTuples*.



Note that by storing all matched tuples in an *ArrayList*, we are not eating into M buffer allocated for the algorithm. We assume that in the memory, it is able to store all tuples with common value. Hence, the actual memory used in reality will be slightly more than M. This is actually one of the requirements for Refined Sort Merge Join.

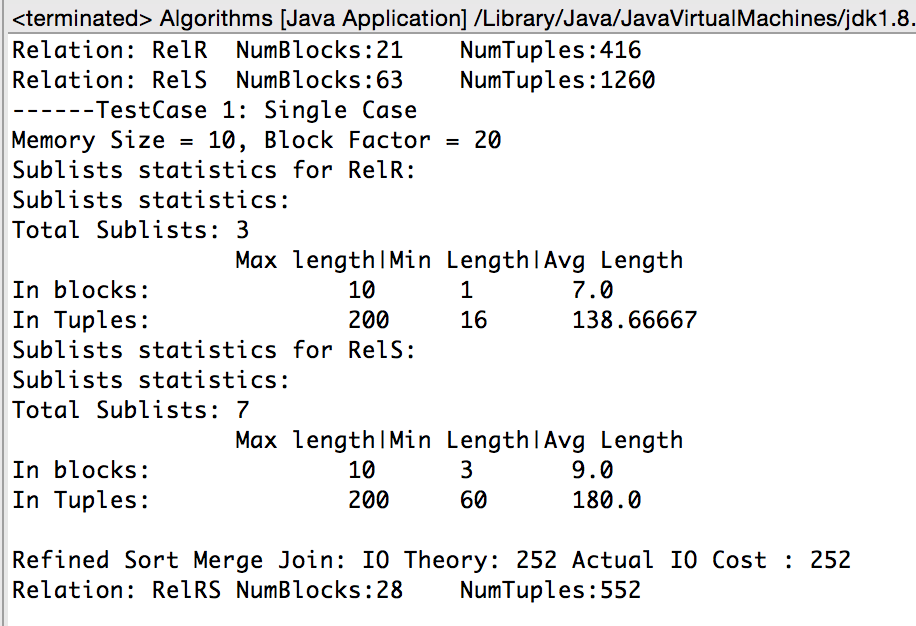
**Step 6:** Lastly, we perform a *cartesianJoin()* for all the matched tuples:

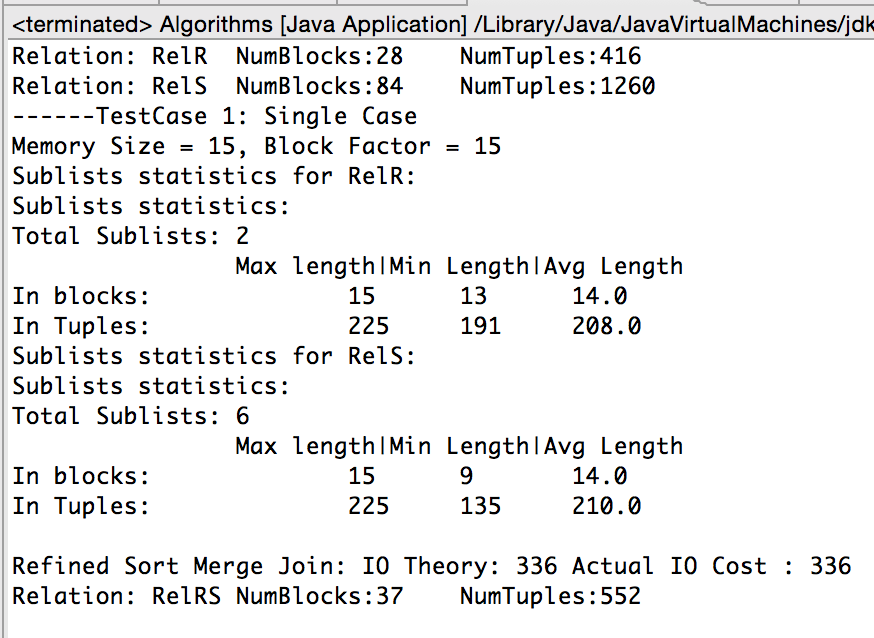


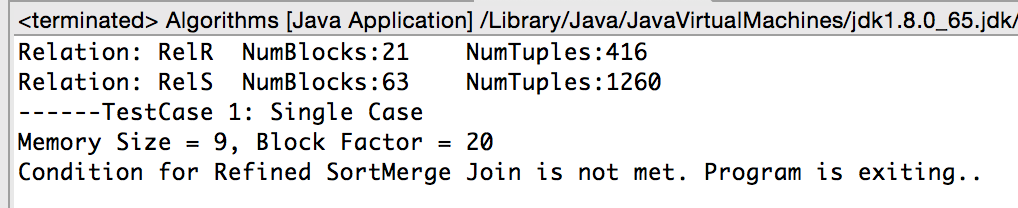
**Step 7:** Repeat step 3 until either relation R or S is empty.

## 2.2 Test Case Analysis

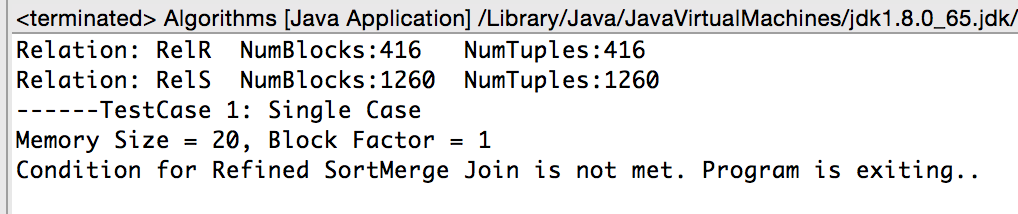
Below are a few test cases for Refined Sort Merge Join. Note that since we start of with relation R and S condensed, number of actual IO cost we obtained always equal to theoretical IO cost (which can be calculated as *3(B(R) + B(S))*

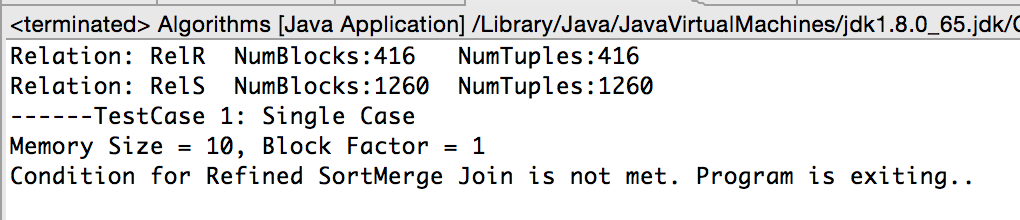


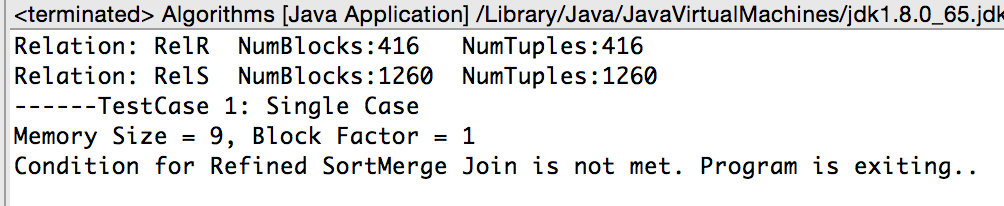




Refined sort merge join requires at least 10 memory slots for the algorithm to work as it requires 1 memory slot for the buffer, loading of relation R and loading of relation S. The block factor cannot be reduced to 1. Please refer to table in section







# 3. Hash Join

The main idea of a hash join is to partition the relations into buckets based on hash value from the tuples’ key before joining them to reduce the number of joins we need to perform. The minimum memory buffer requirement is 2; one for an input buffer and one for an output buffer.

In addition, the number of blocks for the smaller relation must not exceed M-1, where M is the memory buffer. The number of IOs are computed from:

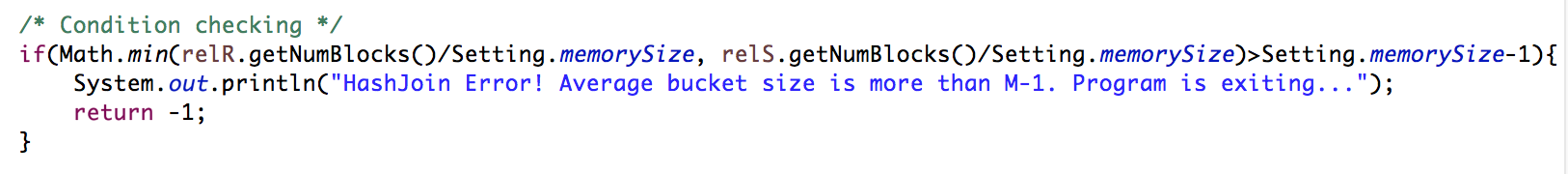
1. Reading each relation into memory for hashing
2. Writing back hashed buckets to disk
3. Loading hashed buckets from each relation for joining

A theoretical IO Cost for Hash Join is *3(B(R)+B(S))*. However, different from *RefinedSortMergeJoin* and *MergeSort*, *HashJoin* does not always incur exactly *3(B(R)+B(S))*, because there are a lot of hashed buckets produced in HashJoin, when put side by side, total blocks will be more than the condensed blocks in original relation.

## 3.1 Code Analysis

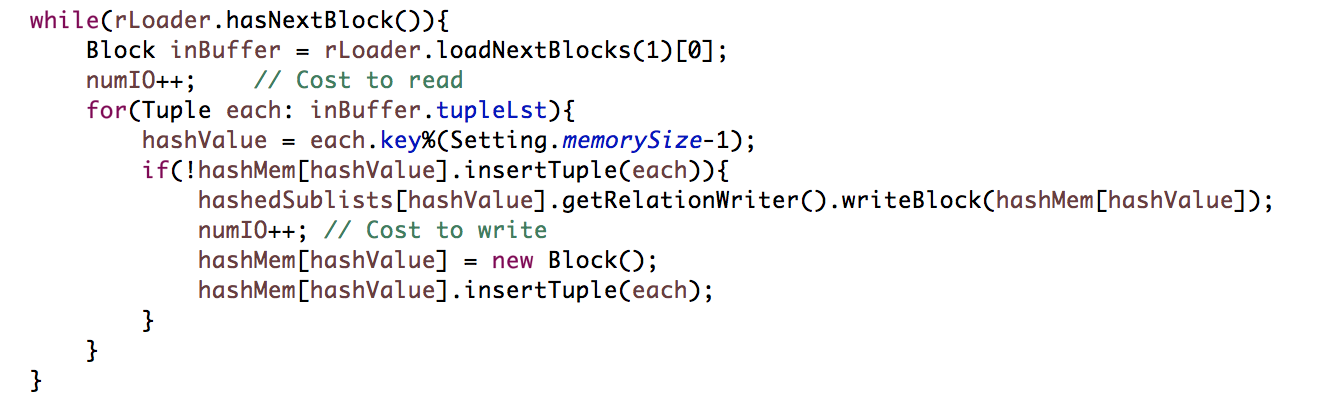
Below are the screenshot snippets and explanation for each phase:

**Step 0:** Before we commence the algorithm, we need to make sure the assumptions are met. That is, . This is the code:

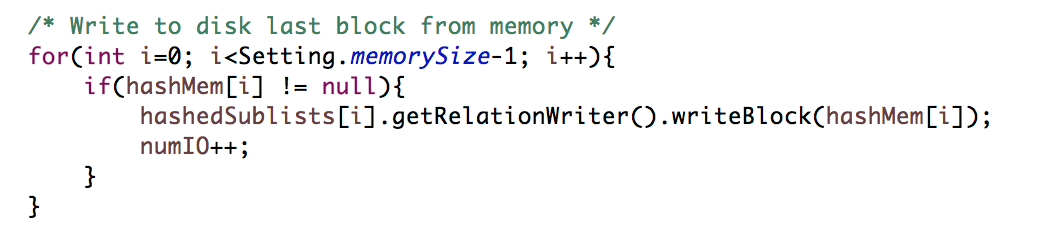


##### Phase 1 – Hash to M-1 buckets

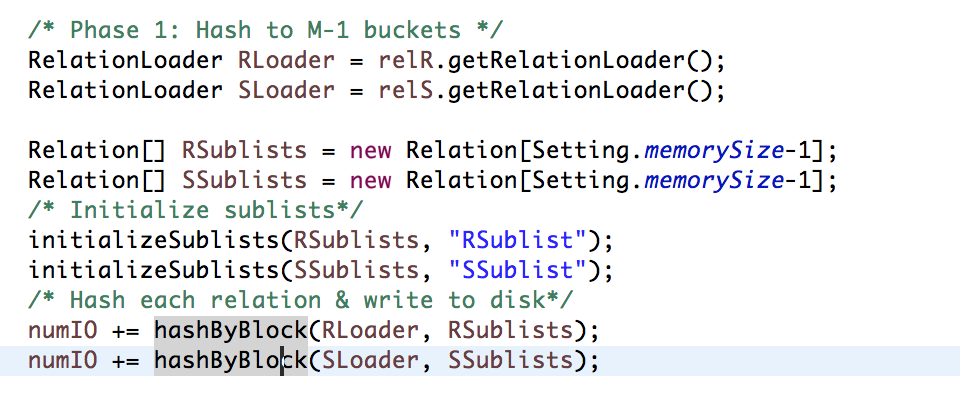
**Step 1:** The algorithm for this phase is as follow: For each relation, first load one block after one another. For each tuple, we hash into M-1 buckets. If any hashed bucket is full, we write to the hard disk, and empty the hashed buffer. We enable this in our *hashByBlock()* function. Below is a code snippet of the main algorithm:



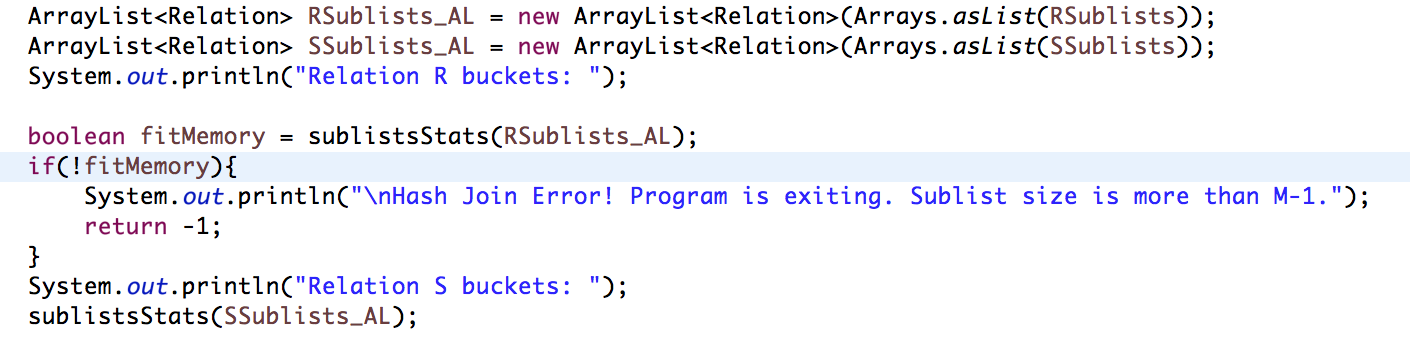
After the while loop, we will write the remaining tuples in memory to the disk:



Here is the code from the *hashJoinRelations()* function that makes use of *hashByBlock()* function:



**Step 2:** Print out sublists statistics for testing purposes, using *sublistsStats()* function.



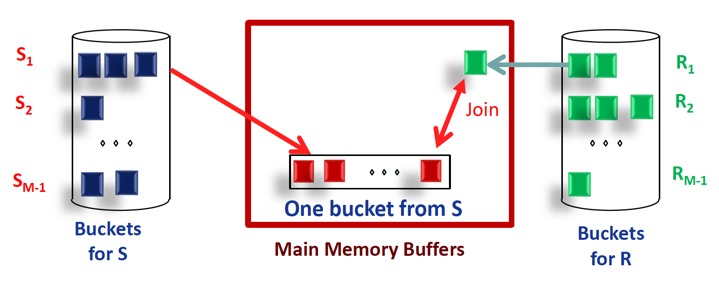
There are 2 things to note for this code:

Firstly, we wrap the sublists in *ArrayList* because we initially use array for memory optimization.

Secondly, we perform a check while printing out sublists, to pass back a boolean telling us if the maximum length of R’s sublists is less than M or not. If it hits M and above, the program will terminate.

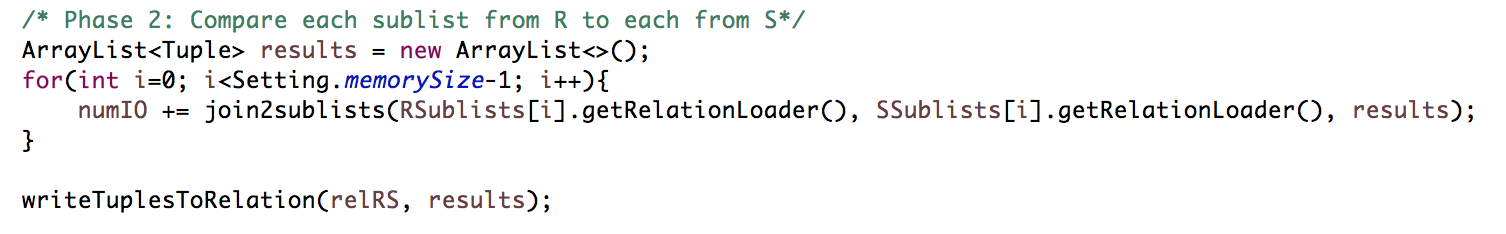
##### Phase 2: Compare each sublist from R to each from S

**Step 3:** Having hashed buckets for each relation in the disk, we then load them back into memory to do the joining (refer to diagram).



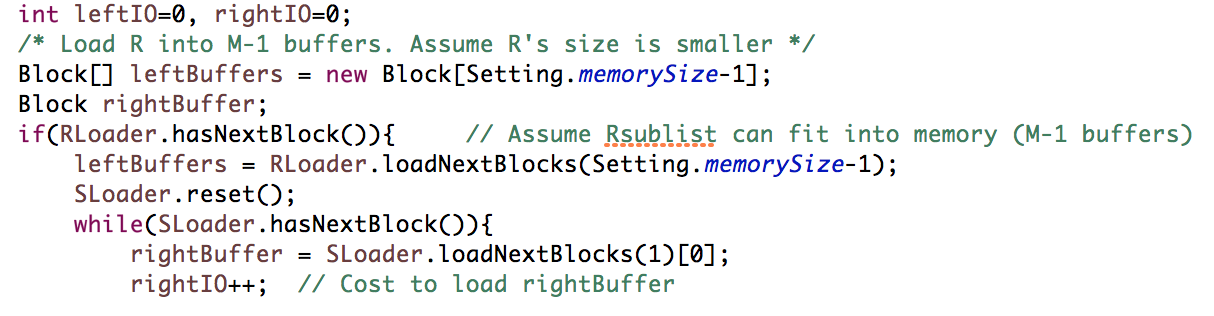
Note that there are leftBuffers (in red) that consist of M-1 block in memory and there is one rightBuffer that occupies 1 block, and that was loaded from right Relation. To make sure all joins are matched, and this algorithm of joining (i.e. phase 2) is a one-pass join algorithm, the whole bucket for each hash from left relation must be able to fit into M-1 blocks in memory. Therefore, as an optimization, we will make the smaller relation on the left and the bigger relation on the right.

Below is the code snippet for phase 2:

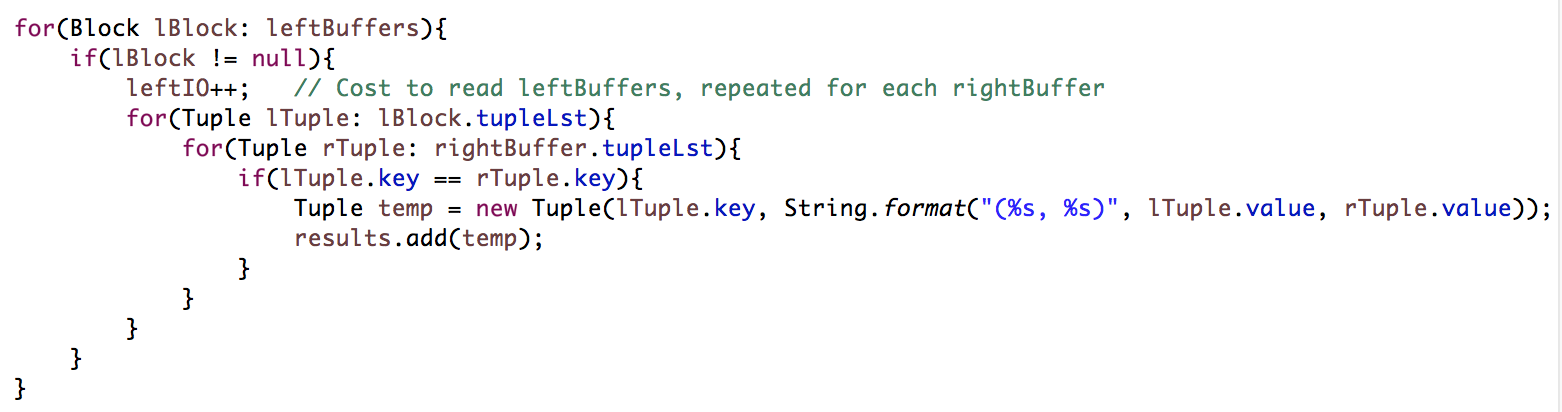


Here we will traverse each bucket according to hash *0* to hash *M-2*. Then we will call *join2sublists()* function, where we take in buckets from 2 relations and return an *ArrayList* of *Tuple* that’s called *results*.

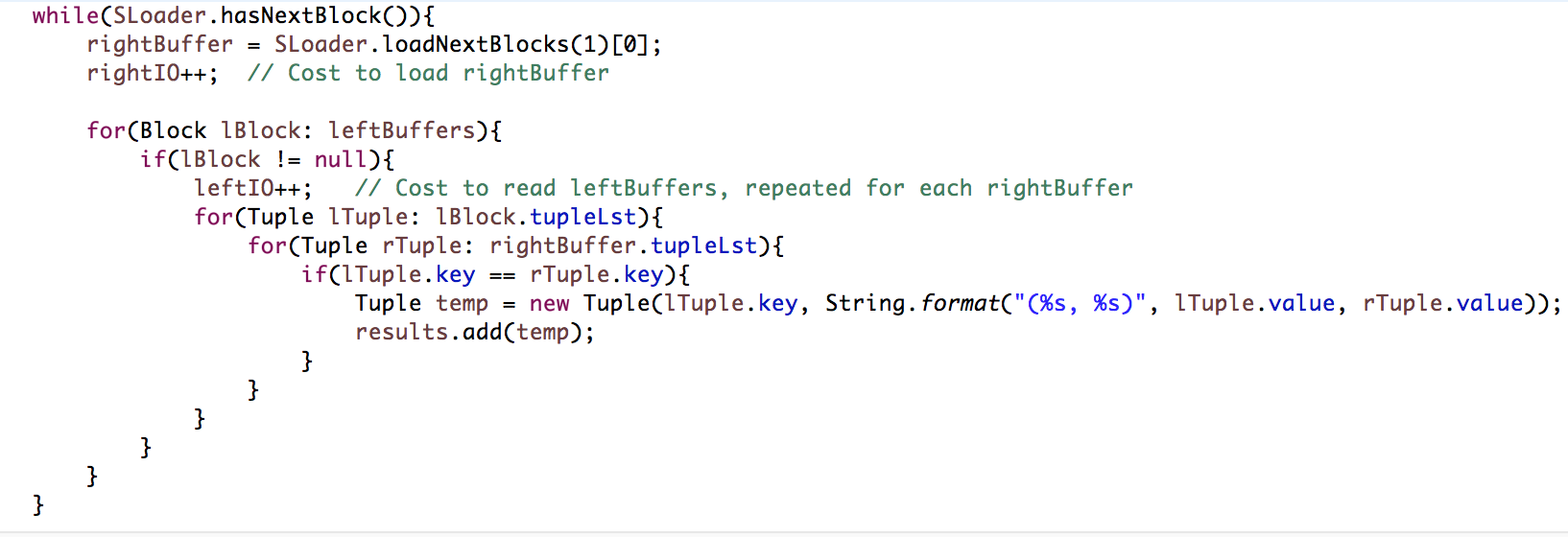
**Step 4:** In *join2sublists()*, first we will load M-1 blocks from left relation (in this case, *relR*), followed by loading 1 block from right relation (*relS*).



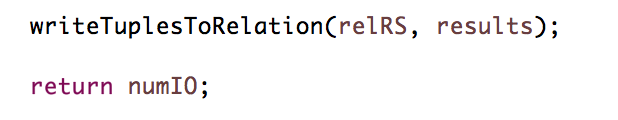
After we have block array of leftBuffers and rightBuffer, we do a nested loop join, that is similar to Block Nested Loop Join, i.e. for each tuple in leftBuffers, traverse the entire rightBuffer to find match.



**Step 5:** Return to while(SLoader.hasNextBlock()) until there is no more block in right bucket. This is the entire while loop:

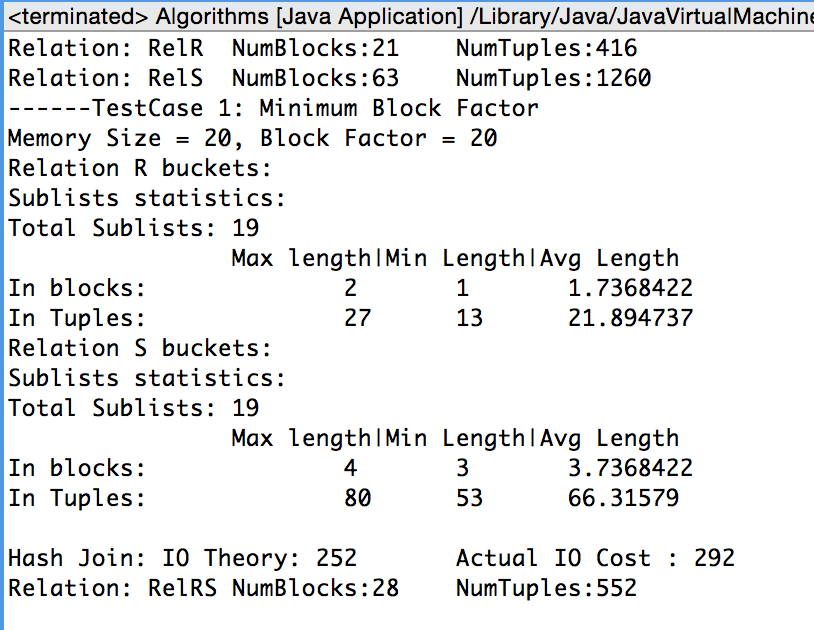


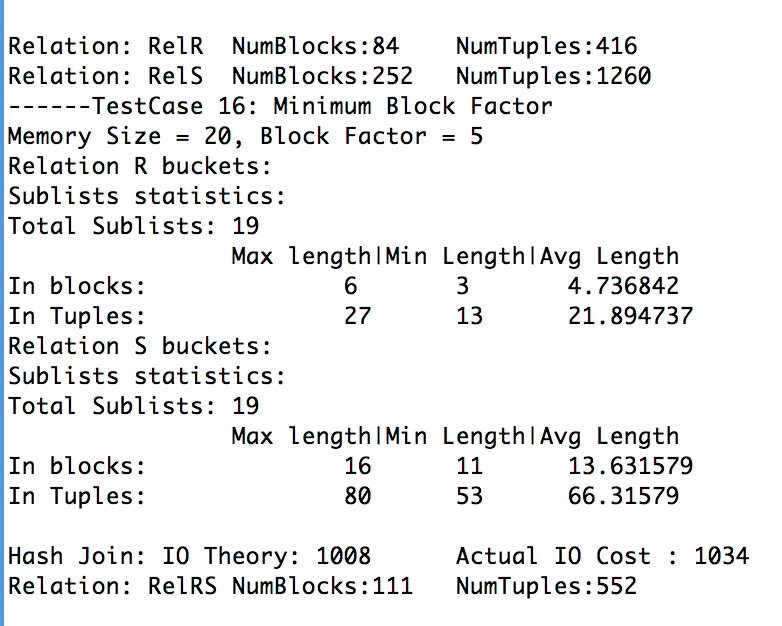
**Step 6:** With *ArrayList<Tuple> results* which consists of all the matched tuples, we write them to relRS using the code from MergeSort and end the algorithm.



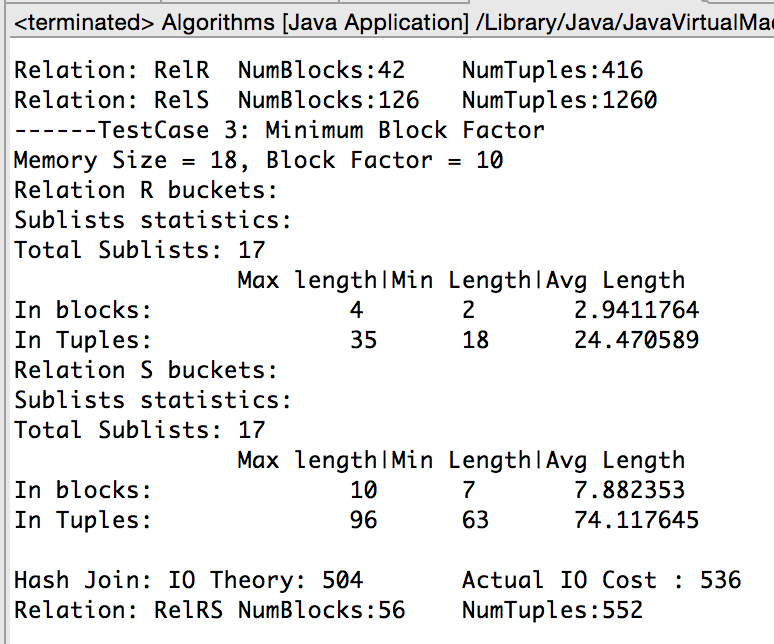
## 3.2 Test Case Analysis

Below are a few output screenshots for HashJoin algorithm

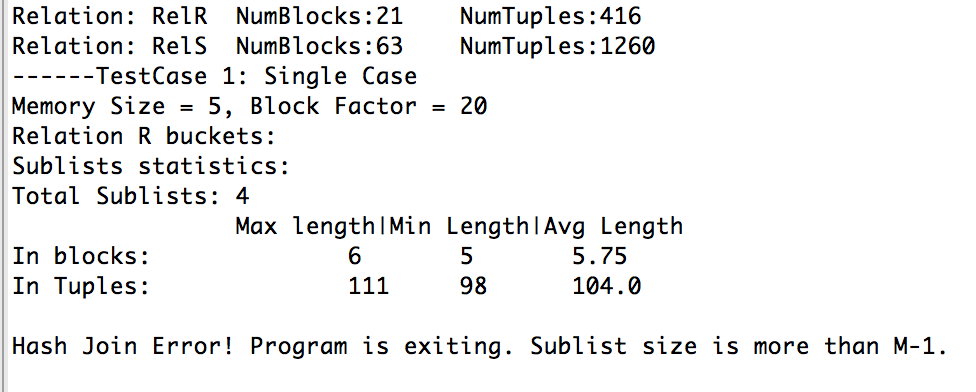




Note that, different from MergeSort and Refined Sort Merge Join, Hash Join’s actual IO Cost is not always equal to theoretical IO Cost. The reason is explained at the beginning of section 3.

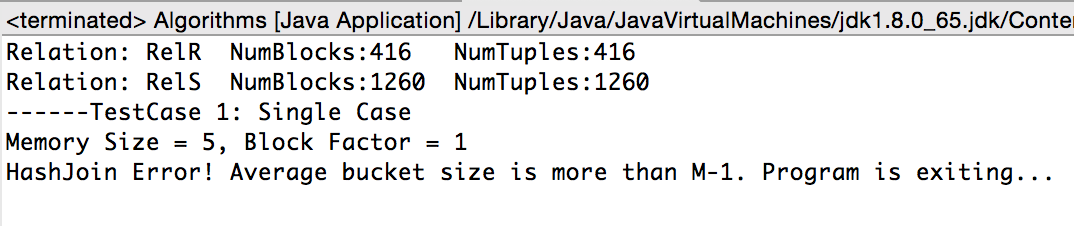


Hash join requires a minimum of five memory slots for the algorithm to work as it requires one memory slot for loading one block from the relation to memory and another memory slot to save the tuples after partition.



We can see here the maximum length for a sublist is 6, while there is only 5 blocks in memory allowed. Based on the general condition , *Min(21/5, 63/5)=4 ≤ 5-1=4*, initial checkpoint is passed, but the actual sublists generated exceed the size of memory. Therefore, we need to check for size of sublists while printing out sublists statistics.

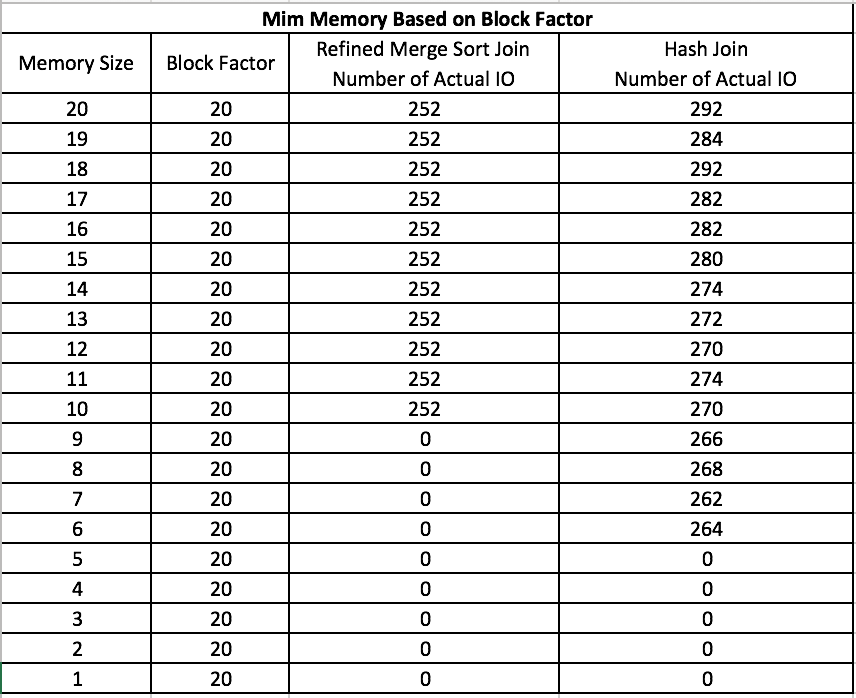
The blockfactor cannot be reduced to 1:



IO cost has high correlation with the block factor. Lower block factor results in higher IO cost. This is because lower block factor results in larger number of blocks in the relations, thus more block is required to be read into memory buffer and write into disk which results in high IO cost. On the other side, higher block factor will results in lower IO cost.

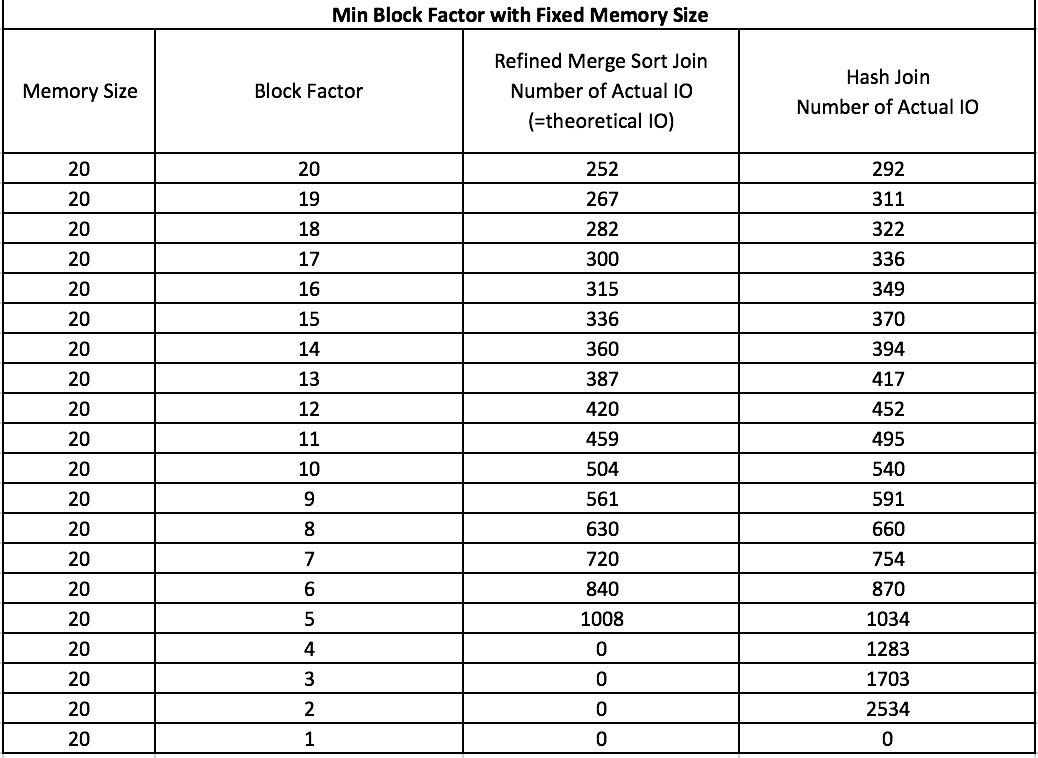
Lower memory size requires a high block factor in order for the algorithm to work. The block factor decreases as memory size increases until the block factor reached 1 which means only one tuple exists in each block.

# 3.3 Analysis Comparison between Refined Merge Join and Hash Join



From the above table, we started off with the default memory size and block factors and slowly reduced the memory size. We can see from the table that the changes in memory size does not affect the IO cost for refined sort merge join, but affects slightly for hash join. By reducing the memory size, if will only affect if the algorithm can or cannot be executed. If the memory size is too small and does not meet the buffer requirements, the algorithm will not work.

In this test case, refined sort merge join requires more memory than hash join. Minimum memory and block factor for Refined merge sort join was 10 and 20 respectively. Whereas for Hash join, it was 6 and 20 respectively.



However, based on another test case as per the table above, we fixed the memory size and only reduced the block factors value. We can see from the table that block factor will affect the IO cost for both refined sort merge join and hash join, because each block that were reading or writing is considered an IO cost. The lesser the blocks, the lesser the IO cost. By decreasing the number of block factors, it will result in a larger block amount for the relation.

# Appendix

## Comparison of results

|  |  |
| --- | --- |
| SQL Result on Relation R | **Merge Sort** on Relation R |
| Top Result  MergeSortJoinSQL1.JPG  Bottom Result  MergeSortJoinSQL2.JPG | Top Result  MergeJoinJava.JPG  Bottom Result  MergeJoinJavaBottom.JPG |

## 

|  |  |  |
| --- | --- | --- |
| Result of SQL | Result of **Refined Sort Merge Join** | Result of **Hash Join** |
| TopR Result:  otherstop.JPG    Bottom Result | Top Result:  refinejointop.JPG  Bottom Result  refinejoinbottom.JPG | Top Result  hashtop.JPG  After sorted using Merge Sort (Top 5):    Bottom Result  hashbottom.JPG  After sorted using Merge Sort(Bottom 5): |