In-Memory Query Execution

Aristotelis Koutsouridis
Giorgos Liontos
Stefanos Baziotis

January 2020

1 Introduction

We present a simple system that can execute RDBMS-like queries in memory. Among the query operations are natural joins, selection and projection. A cache-aware approach is used in the implementation of the natural join operator. The execution of a query is split into 3 stages: parsing, query-optimization(plan generation), plan execution. Information needed by the query-optimizer is constructed at the initialization of the system when the relations are loaded into memory. This information is mainly heuristics about the characteristics of relations: cardinality, frequency of values, etc. Lastly, we utilized parallelism on 3 levels using a multi-threaded architecture.

2 Join Operator

The join operator is implemented using the sort-merge-join method, that is, it **first sorts** the 2 relations (`O(nlogn)` complexity but with average `O(n)` complexity), **then merges** them(`O(n*k)` complexity, where `k` is the size of a group with same key).

2.1 Data

The database consists of a set of relations which in turn consist of a set of columns. Each column consists of 8-byte values. These data are stored in memory column wise for fast processing. For example if we have a relation of 100 rows and 2 column then we would keep an array of two arrays in memory. The first array would store the values of the first column for all tuples while the second array would store the values for the second column.

2.2 Sorting

The join operator is to be applied to data that are quite big - big enough so that they don't fit in the L1 cache of most (if not all) today's processors. This is an important concern as cache-misses are of the most dominant causes of performance degradation in modern hardware.

So, we have to use a sorting algorithm that tries to avoid cache misses as much as possible - one that respects cache locality. The algorithm we used is a hybrid one: It uses radix sort, until the data size is below or equal the L1 data cache size[1]. Below that threshold, a variation of quicksort is used with the reasoning that because we have maximum memory performance, we can use the most optimal algorithm in terms of complexity.

[1]We are given a default threshold of 64KB. But, if the actual L1 data cache size is smaller, the performance degrades dramatically. So we took the pragmatic approach of not using the default and instead asking the OS for the actual L1 data cache size.

2.2.1 Algorithm - A Variation of Radix Sort

This variation of radix sort performs 8 rounds, because the data that it sorts are 8-byte values. The `n`th round sorts data according to their `n`th byte and separates them into groups. All a members of a specific group have the same first `n` bytes. For example, consider the data:

Byte 0	Byte 1	Byte 2	Byte 3	...
0	6	23	56	...
1	7	78	7	...
0	6	78	98	...
1	54	13	134	...
0	88	77	63	...

We assume that byte 0 is the most significant byte, byte 1 the second most significant and so on. After the first round, the data will be like:

Byte 0	Byte 1	Byte 2	Byte 3	...
0	6	23	56	...
0	6	78	98	...
0	88	77	63	...
1	7	78	7	...
1	54	13	134	...

because the first round will sort data according to their 1st byte (which is byte 0).

Note that now 2 groups have been formed: Those with the keys with
byte 0 = 0 and those with byte 0 == 1 (In the second round, the 2 keys with byte 0 == 0 and byte 1 == 6 will form a group on their own).

After each round, groups are sorted relative to each other. That is, if we have 3 groups then all the members of the first group will be less than all the members of the second group and all the members of the second group will be less than all the members of the third group (however, the members of a specific group might not be sorted relative to each other). Also, each group will have taken its final position. For example, consider the data:

Byte 0	Byte 1	Byte 2	Byte 3	...
0	7	78	7	...
1	6	23	56	...
1	6	78	98	...
1	6	77	63	...
2	54	13	134	...

This can be the result after a second round. The 3 middle elements form a single group Note that this group starts from index 1 (counting from 0) and ends in index 3. This group is in the correct position [1,3]. That is, even when we finish the whole sorting, this group will take indexes 1 to 3 in the final relation. What is missing is that this group is not necessarily sorted inside. That is, keys 1-3 are all smaller than key 4 and all bigger than key 0 (because their first 2 bytes at hand are bigger). But keys 1-3 themselves might be unsorted relative to each other (and indeed they are as for example 77 <78).

So, how do we sort them completely?

An important note before we move on is that this procedure is not done in place. According to the algorithm, we have 2 places where we can save data. The one is called **R**, which contains our initial data. The other is an auxiliary array of the same size, which we name **aux**. During the rounds, we move data back and forth from **R** to **aux** and conversely (where **R** becomes the auxiliary and **aux** the primary and vice versa). This is not necessary! According to [1], we can do this procedure in place with a somewhat more complicated algorithm.

After a round has finished, there are 3 categories of groups:

1. Groups of only one element.
2. Groups that have more than one elements and but can be quick-sorted (their size is less than the L1 data cache size).
3. All the other groups.

2.2.2 Cases 1 and 2

Note that in cases 1 and 2, the elements inside the group are completely sorted. Completely meaning not just the group relative to other groups but also its elements relative to each other. Second, note that when we finish a round, every group has been put to an auxiliary space. So, if a group is also sorted, then we're done.

But are our data saved in **R** or **aux**?

Assuming that we have never gone to cases 1) and 2), then we finish all the rounds. Following what we said above, i.e. that data will be moved strictly back and forth, then obviously the final data will be saved back in **R** (we start from **R** moving to aux, then back to **R** etc. and the number of rounds is even).

However, we don't know when are going to fall in case 1) or 2). In either of these cases, since the group is already completely sorted and in the correct position, we can just move it to where we will save the final result (note here that for this to work correctly, **aux** and **R** should be exactly equal so that we always have memory to move to the same positions).

We chose arbitrarily to save the final result in **R**. One can't take an informed decision on this given that the data are random (and with no pre-processing / guessing). But one thing to note is that it's quite possible to do more moves that it is actually required. If for example most of the data fall in case 1) or 2) after the first round consider what will happen: We'll first sort the data according to the first byte (as the first round is defined). We will move the data to **aux**. Then we'll deduce that the data can be e.g. quicksorted and we'll sort them (in place) in **aux**. Now, we'll see that our data are not where we save our final result, so we'll move them there. The "there" is **R**. Consider that what was just described could be done without having **aux** at all, by just staying in **R**. Even if we do that though, we still have one (possible) move to do because the group, although sorted (and in our tests, that optimization was not worth it).

2.2.3 Case 3

With case 3, we simply move the group to the next round. It should be noted here that if we do this procedure for 8 rounds, the data will be completely sorted.

There are a couple of subtle decisions here. First, while the algorithm is recursive, we don't use actual recursion. We find that the code is more manageable with a custom-built stack. Furthermore, we never actually move memory when we move a group to the next round. The way it works is we just keep indexes to the initial memory and swap the relation pointers them each round. That is, say we have this big array of data and we want process a part of it.

key 1
key 2 — from-index
key 3
key 4 — to-index
key 5

So, if keys 2 through 4 form a group, this is passed to the next round as essentially 2 indexes and now we operate in that slice of the original array. So, at any point during the sort, the only memory used is the original space (i.e. **R**), the auxiliary space (i.e. **aux**) and the stack. As a side note, the stack never deallocates during the sorting. Only grows. The advantage of this is that we don't spend continuously time in allocating and de-allocating.

3 Plan Execution

In this section we will cover the implementation of the 'engine' that executes the query plan.

3.1 Intermediate Result Object (IR)

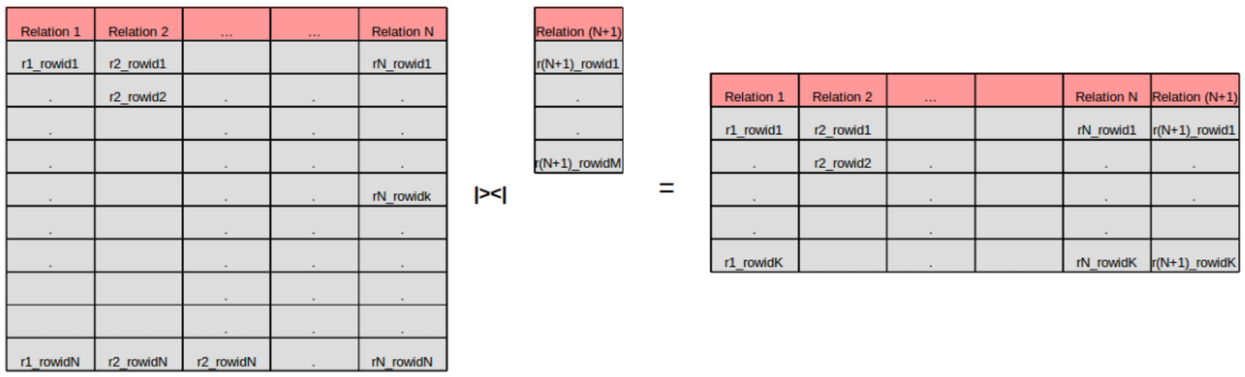

Figure 1: Example of joining a new relation into an Intermediate Result. The cardinality is incremented by 1 after the join.

When joining 2 relations, we obtain a result that contains pairs of row-ids(refer to the tuples of the corresponding relation). But what happens when a 3rd relation is joined to the existing result? How do we handle the cardinality change in the result?

In order to solve these problems, we introduce an object called 'Intermediate Result'. This object holds a collection of relation row-ids. On a semantic level, it is a 2D array. The rows represent **result tuples** after an arbitrary join operation. The columns represent **relations** that participate in the result up to that moment.

The implementation is quite different though. For cache efficiency reasons this 2d array is stored in a column major manner. When a join result is obtained and resizing of the IR is needed, we avoid reallocations of memory by using a **vector**, and checking for sufficient capacity.

Information on the sorting of intermediate results. After a join between 2 relations is performed and the IR is updated, it's tuples are sorted on the corresponding join columns. We can use this information in order to avoid unnecessary sorts. This optimization is very important due to the huge overhead of sorting. We must bear in mind that sorting is a $\theta(nlogn)$ operation, and for large data-sets this can have a significant penalty in execution time as well as memory requirements.

Profiling confirmation of our assumptions on sorting time The overhead is understood better by taking a look at the execution time of each stage of the join operator, as well as the total fraction of sorting time over total execution time of a query batch. Using the linux utility **perf** we observe the following execution time distribution over some routines:

3.2 Executing the Plan Tree

Even though the IR object allows us to perform as many joins as we need to complete the execution, it does not support joins that operate on 2 relations none of which is currently present in the IR. For example let's

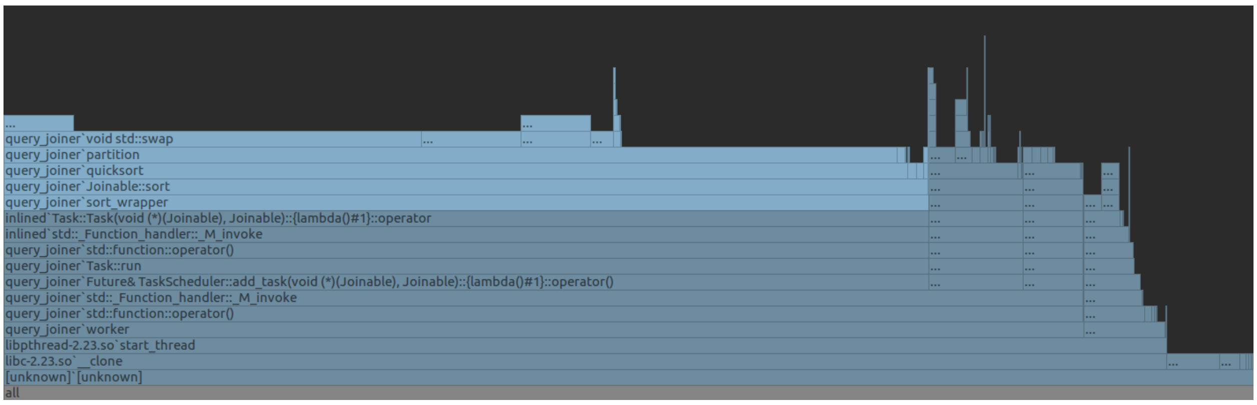

Figure 2: 'Perf' output for a query batch. We can see that approximately **75% of the total time** is spent in sorting.

assume that relation **A** is initially joined with relation **B**. Then the execution plan requests from the engine to perform the join operation between relations **C** and **D**. These relations do not exist in the current IR. The only solution to this is to create a new IR. If we do this arbitrary many times, we end up with a list of IR's. Eventually they will all collapse to a single IR, on which we will perfrom the projection operation of the query. This happens due to the fact that the queries do not contain any cross-products and all relations participating in a query are connected by means of join operations.

In order to achieve all the above functionality we need to implement the routines below:

- **Initial Join**: Add the 2 first relations to an IR.
- **Common Join**: In case one relation is present in some IR and the other is not. The new relation is joined and then appended in the IR as shown in Figure 1.
- **Filter Join**: Both relations exist in the same IR, by convention execute a filter(WHERE like clause)
- **IR Merge Join**: When the 2 relations of the join clause exist in 2 different IR's. A merge of the 2 IR's is needed. In this case the number of IR's is reduced by 1.
- **Filter clauses**: By convention we execute all filter clauses in a lazy manner. Right before a join for a relation is requested, the filtering takes place.

4 Utilizing Available Parallelism

Modern CPU's have multiple cores that give the ability of executing multiple parts of or even a single part of a program in parallel. Even low-end PC's have more than 2 cores and 4 threads. In order to take advantage of the computational power of a system we enhance our approach by adding multi-threading. Parallelism takes place on 3 levels, from query batch down to the sort-merge algorithm itself.

In order to implement these levels, we need a comfortable way for thread handling. A thread pool along with a simple scheduler does the job. When using the scheduler, we specify the number of threads that will be running. We must be cautious with the number of threads in order to avoid context-switch and resource overheads so for that reason we normally set the number of threads in the pool equal to the number of the threads that the CPU has to offer. That can be easily done by querying the Operating System.

4.1 Query Level Parallelism

On this level we execute each query in parallel by giving each query to the task scheduler. Although at a first glance this may look a really promising way to increase overall performance, there are a few downsides.

First of all there is no guarantee that the queries have the same running time. This means that the overall **execution time is bounded** by the **execution time of the most time consuming query**. In addition to that, the more the queries running concurrently, the more the memory requirements(hence a slower system). This is something that we expected. Concurrent queries must have all their IR's in memory. On large data-sets, 3 or 4 concurrent queries may be enough to eat-up all system memory. To address this issue we bound the number of queries executing in parallel to the half of the number of threads available.

4.2 Query Operation Level Parallelism

On this level we try to run operations like joins, filters, etc in parallel. We only implemented this level for joins, because other operations are seemingly a lot less time-consuming for most data-sets. A good example to illustrate this kind of concurrency is the example given in section 3.2. We start by giving a join between two relations **A** and **B** to be executed in a thread in the task scheduler. Then we can **independently** join relations **C** and **D**. When we merge the 2 IR's of the two operations, we first **wait** for their operations to finish.

This level fills in some of the lost utilization of the previous level. To increase the level of parallelism even more we move to the next level.

4.3 Operator Level Parallelism

On this last level, we try to break down each join into simpler tasks that can run in parallel. For example if two relations need to be sorted on a column, these two sort operations can run concurrently. In addition to that the merge algorithm is parallelized: the sorted relations are split into buckets and each bucket can be merged in parallel.

5 Query Optimization and Plan Generation

Our main concern in this stage of the query execution is to reorder the operations of a query, in order to minimize the query execution time. In order to judge the quality of join trees, we need a cost function that associates a certain positive cost with each join tree. Then, the task of join ordering is to find among all equivalent join trees the join tree with lowest associated costs. One common cost function is a cardinality estimate. It is based on the cardinalities of the relations, i.e. the number of tuples contained in them.

Assuming such a function, we can then use a simple algorithm [2] to find quite a good join tree:

```
for (i = 1; i <= n; ++i) {
    BestTree({Ri}) = Ri;
}
for (i = 1; i < n; ++i) {
    for all S subset of {R1, ..., Rn}, |S| = i do {
        for all Rj in {R1, ..., Rn}, Rj != S do {
            if (NoCrossProducts && !connected({Rj}, S)) {
                continue;
            }
            CurrTree = CreateJoinTree(BestTree(S), Rj);
            S' = S + {Rj};
            if (BestTree(S') == NULL || cost(BestTree(S')) > cost(CurrTree)) {
                BestTree(S') = CurrTree;
            }
        }
    }
}
return BestTree({R1, . . . , Rn});
```

The optimizer is so effective that in the available hardware, the medium dataset, without it, simply **cannot** run.

6 Performance

Our final results are shown below. We tested the system in various configurations. What we observed is that, trying to run things concurrently is not always optimal. Parallel solutions often required a ridiculous amount of memory. On these results, we show the updated version that take memory abuse into consideration and try to balance things.

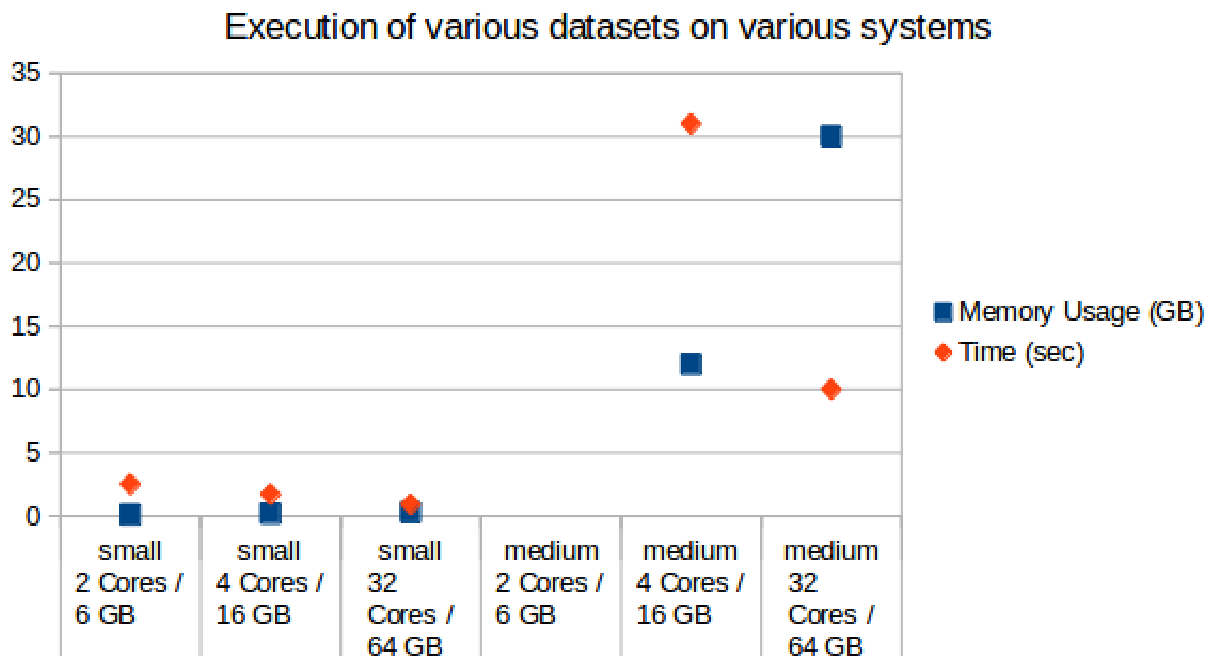

Figure 3: Overall Performance in various systems

We observe that as the number of cores is increased, memory usage is also increased. Also we can see that a system with only 6 GB of memory can't run the medium datasets due to insufficient memory.

If we compare these final results to a fully serialized version of the query executor, we see that the use of multithreading increases the average performace by around 200% (4-core systems). In some cases though, we obtained a speedup of up to 320%. This is a great improvement, given the nature of the operations that are needed: processing of very large memory chunks by multiple threads.

References

[1] Orestis Polychroniou Keneth A. Ross, *A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort.* Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014.

[2] Guido Moerkotte, *Building Query Compilers*, March 2019