ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2022 - 2023

PROJECT REPORT

Christos Ioannou

A.M: 1115201900222

Polydoros Tamtamis

A.M: 1115201900184

GOAL

The goal of the project is the development of an application that performs **SQL queries** to a database. Each query uses a number of relations on which different filters and joins are applied.

Any predicate that compares the elements of a specific column of a relation with some non-negative integral number is a **filter**.. The operators that may be used are the following: >, <, =. The '=' operator is also used to perform a join between columns of two relations. The result of each filter is a set of the row ids of the relation that satisfy the filter's condition. Similarly, the result of a join between two relations **R** and **S** is a set of row id pairs in the form of **{row_id_R, row_id_S}**. For each pair, the join condition must hold true.

Join operations are generally considered computationally expensive. Therefore, one of the main goals of the project is the efficient and performant execution of join predicates. For that reason, we try to take advantage of the hardware as much as possible;specifically the L2 cache of the CPU. We aim to have the most oftenly used elements of a relation (at a given time) loaded into the cache to avoid cache misses and improve performance during join. An effective solution to this problem is called **Partitioned Hash Join** (or **In Memory Hash Join**). It consists of 3 phases: the partitioning phase, the build phase and the probing phase.

By reorganizing the original relations into small enough partitions that fit in the L2 cache, we seek to have all the elements of a relation that share a specific feature loaded into the cache during a join. The partitioning process is executed twice at most: if some partition doesn't fit into the cache after the second partitioning, then we leave it as is. The initial partitioning happens based on the last **X bits** of a value: by the end of the first partitioning, a new relation is created which has all the elements having the same **X bits** equal into the same partition. If one partitioning isn't enough, we increase the bits to **Y** and repeat the process. Ideally, each partition should fit into the cache by the end of the partitioning phase.

During the build phase, we construct hashtables for each partition separately. Each element of the newly formed relation **R'** is inserted into its corresponding Partition Hash Table. Collision resolution is handled by **Hopscotch's algorithm**. If duplicate values exist, they are all inserted into the same bucket's linked list. Each hash table entry consists of a key - which holds the value of R[i][j], where **i** is the row and **j** is the column of the relation - and the payload - which is the row id **i** to which the value corresponds.

The probing of the hashtable is the final step. We know which partition each element of **S'** belongs to; therefore, we can simply look up the element's value at that specific's partition Hash Table. If the value exists, a result entry **{ row_id_R, row_id_S}** is appended to the final result.

We can improve the performance even further by implementing multithreading and having a query optimizer. The partitioning phase, the construction of the hashtables and probing can be parallelised. Additionally, each thread can operate on a different query so as to keep the CPU as busy as possible. For the purposes of multithreading, we have implemented a **Job Scheduler** that assigns a variety of jobs (e.g. **build jobs**) to an existing thread pool.

Before any join happens, all filter results should be computed, since this helps minimise the relations that are going to be joined later. More specifically, the execution order looks like this: filters using operator '=', filters using operator '>', filters using operator '<', joins in the order decided by **the query optimizer**. The query optimizer can estimate which series of joins results has the lowest amount of intermediate results and execute them in that order. To do that, four statistic variables need to be used for each column of each relation:

$\{ l_A, u_A, f_A, d_A \}$ *where A refers to the name of a column of some relation R*

The value l_A refers to the minimum value of column A whilst u_A refers to maximum. The value f_A refers to the amount of data in column A (count), and finally the value d_A refers to the number of unique values of column A (distinct count).

DESIGN CHOICES

- We have opted for a vector as the data type that holds the intermediate results of each query. **simple_vector** is implemented as a dynamic array, which grows whenever its capacity is reached. Whenever this happens, its capacity is doubled and the existing elements are copied to a new memory location.

  The reasoning behind using a vector over a linked list is that we need to have **O(1)** access time for each element. Additionally, arrays behave better when it comes to cache since the elements live in contiguous memory, unlike a linked list.

  Furthermore, the **simple_vector** interface offers a **steal** method which helps avoid unnecessary copying (for example when assigning a temporary (local) vector to another, which outlives it) by simply moving the resources of the temporary vector to the other.

- When it comes to the Job Scheduler, the job queue is implemented as a linked list. That's because we are not interested in random access performance - we are practically only interested in the first element of the queue. Linked List also allows for fast element removal (which is required by the **pop** operation).

- We have added padding in certain data structures to avoid false sharing of the **CPU cache** between threads. False sharing occurs when a thread writes that belongs to a cache line which is shared by another thread. Even if the second thread has nothing to do with the object that's changed, it still has to invalidate its cache and fetch the cache line again, which can lead to performance loss.

  The padding that's added assumes that each cache line is 64 bytes, which is the case for the majority of modern architectures today. However, it can be configured differently to accommodate for different sizes as well.

- Initially, we were using **int64_t** as the primary datatype inside our containers (e.g. simple_vector). In most cases, this was unnecessary: **int32_t** is enough for holding **row ids** and other objects. Therefore converting **int64_t** to **int32_t** was feasible. This conversion significantly lowered the overall RAM usage and seems to have even slightly improved performance.

PERFORMANCE ANALYSIS

There are numerous settings we can configure that may lead to better performance. Most notable ones are:

- **Thread count**: Having more threads allows for more jobs to run in parallel. However, it's possible that having more threads than CPU (virtual) cores can lead to performance degradation as the overhead of creating and scheduling the threads might be significant. Additionally, it's unlikely that all the threads will run in parallel in that case. Ideally we want to keep adding threads as long as they improve the performance and the efficiency remains somewhat same or even improves. Of course, this behavior varies among machines and there's no universal thread count that's optimal for all systems.

- **Bits used in partitioning**: The number of bits that are used when partitioning a relation may affect the performance noticeably. That is because using a higher bit count allows for each partition to contain less elements, at the cost of an increased partition count. If each partition is small enough to fit in the **L2 cache** after only one partitioning, we can avoid having to do a second pass, as mentioned previously. This potentially comes at the cost of increased **RAM usage**, since we need an increased amount of hashtables (higher bit count leads to more partitions and therefore hashtables).

- **Neighborhood size (Hopscotch algorithm)**: If all the neighboring buckets can fit into the same cache line, this can lead to performance gains (since less **LOAD** operations would be required by the CPU). Furthermore, neighborhood size defines the amount of buckets we have to examine during an insertion or lookup operation.

MEASUREMENTS

We have observed the performance of the program when executing different workloads under different configurations. For each configuration, the average execution time after 5 runs is reported in milliseconds.

The **public** workload is the larger of the two, putting significant stress on most systems' CPUs and consuming large amounts of memory.

The following measurements are performed using:

- **Neighborhood size (for Hopscotch): 32**
- **Bits (partitioning pass 1): 4**
- **Bits (partitioning pass 2): 8**

Ryzen 5900HX, 32GB DDR4 RAM L2 Cache Size = 4MB Workload: **small**		
Without query optimizer		
# threads query	**# threads join**	**Average Time**
1	1	776.80
2	2	556.40
4	4	467.00
6	6	454.80
8	8	452.00
With query optimizer and only one query at a time		
# threads query	**# threads join**	**Average Time**
1	2	507.40
	4	453.80
	6	431.80
	8	428.80
With query optimizer		
# threads query	**# threads join**	**Average Time**
1	1	627.80
2	2	442.00
4	4	384.80
6	6	377.60
8	8	366.40

Ryzen 5900HX, 32GB DDR4 RAM L2 Cache Size = 4MB Workload: **public**		
Without query optimizer		
# threads query	**# threads join**	**Average Time**
1	1	107928.00
2	2	68813.80
4	4	52660.80
6	6	44802.20
8	8	41632.40
With query optimizer and only one query at a time		
# threads query	**# threads join**	**Average Time**
1	2	78738.60
	4	68604.20
	6	64585.40
	8	63057.20
With query optimizer		
# threads query	**# threads join**	**Average Time**
1	1	94279.80
2	2	71518.40
4	4	57293.00
6	6	52746.60
8	8	50489.00

Ryzen 2600, 16GB DDR4 RAM L2 Cache Size = 3MB Workload: **small**		
Without query optimizer		
# threads query	**# threads join**	**Average Time**
1	1	1138.40
2	2	861.60
4	4	754.60
6	6	668.20
8	8	759.60
With query optimizer and only one query at a time		
# threads query	**# threads join**	**Average Time**
1	2	782.20
	4	690.40
	6	672.20
	8	713.20
With query optimizer		
# threads query	**# threads join**	**Average Time**
1	1	1046.00
2	2	756.20
4	4	628.60
6	6	573.60
8	8	581.00

Ryzen 2600, 16GB DDR4 RAM
L2 Cache Size = 3MB

Workload: **public**

Without query optimizer

The program takes a significant amount of time to finish (approx. 7-8 minutes) and the RAM reaches max capacity. This workload is practically impossible to run without an optimizer on this machine.

With query optimizer and only one query at a time

# threads query	# threads join	Average Time
1	2	127359.00
	4	108520.00
	6	103238.00
	8	101241.80

With query optimizer

# threads query	# threads join	Average Time
1	1	152951.20
2	2	108694.40
4	4	91778.00
6	6	87150.80
8	8	92732.80

THE IMPACT OF PARTITIONING

It's possible that performance improves if we force partitioning to happen more often. For example, setting cache size to a smaller value than the actual one can sometimes yield better results.

Ryzen 2600, 16GB DDR4 RAM L2 Cache Size = 3MB Workload: **public**			
With Query Optimizer			
# threads query	**# threads join**	**Actual L2 Cache Size (3MB)**	**Force Partitioning (256KB)**
		Average Time	
2	2	108694.40	78164.80
4	4	91778.00	70485.80
6	6	87150.80	69616.60

Ryzen 5900HX, 32GB DDR4 RAM L2 Cache Size = 4MB Workload: **public**			
With Query Optimizer			
# threads query	**# threads join**	**Actual L2 Cache Size (4MB)**	**Force Partitioning (256KB)**
		Average Time	
2	2	71518.40	47750.60
4	4	57293.00	39105.60
6	6	52746.60	36241.60
8	8	50489.00	36223.20

This indicates that the cost of partitioning is justified even when there's no *real* need for it, because of multithreading. When partitioning happens it allows for a greater degree of parallelisation: we can have parallel building of the hashtables and parallel joins. The build phase especially seems to have a large impact on the overall impact.

Generally, it seems beneficial to partition a relation, even if it fits inside of the L2 cache. A large relation might still fit into the L2 cache of a modern CPU but partitioning it allows for jobs to run parallel for each partition, which can drastically improve performance, as seen in the figures above.

THE IMPACT OF NEIGHBORHOOD SIZE

We now observe the impact that the size of neighborhood can have on the overall performance when using **Hopscotch's algorithm**. The neighborhood size matters because it defines how many buckets we need to examine in the event of a collision or during a lookup. It can also affect the amount of rehashing that's needed for a specific workload.

Ryzen 2600, 16GB DDR4 RAM (using optimal thread configuration)				
NBHD_SIZE	8	16	32	64
Time (ms)	68001.20	62572.00	65263.00	91977.20

Ryzen 5900HX, 32GB DDR4 RAM (using optimal thread configuration)				
NBHD_SIZE	8	16	32	64
Time (ms)	34328.60	34276.60	36223.20	39586.20

THE IMPACT OF PARTITION BITS

Finally, we examine the effect that the bits used for partitioning can have on performance. Using more bits means that there will also be more partitions (2^n, where n = partition bits) but also that each partition will be more sparse. Additionally, we have to repartition the entire relation a second time (using an increased bit count) even if only a single partition can't fit in the L2 cache. This means that initial bit count may have a noticeable impact on performance and shouldn't be too small.

Ryzen 2600, 16GB DDR4 RAM (using optimal thread configuration and NBHD size set to 16)			
BITS USED	(2,4)	(4,8)	(8,10)
Time (ms)	84760.40	62572.00	59388.80

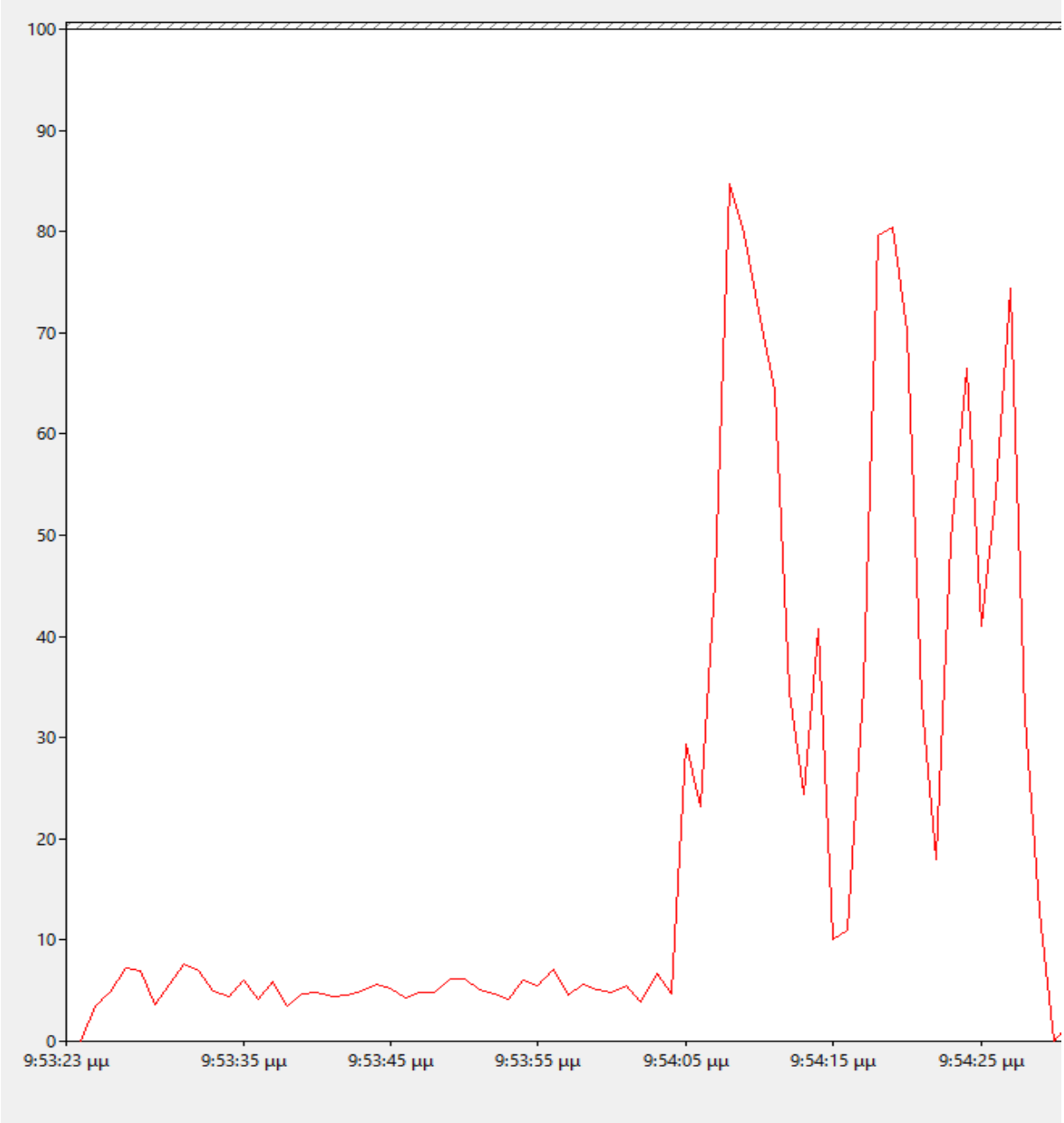
Ryzen 5900HX, 32GB DDR4 RAM (using optimal thread configuration and NBHD size set to 16)			
BITS USED	(2,4)	(4,8)	(8,10)
Time (ms)	47621.00	34276.60	33504.40

CPU USAGE

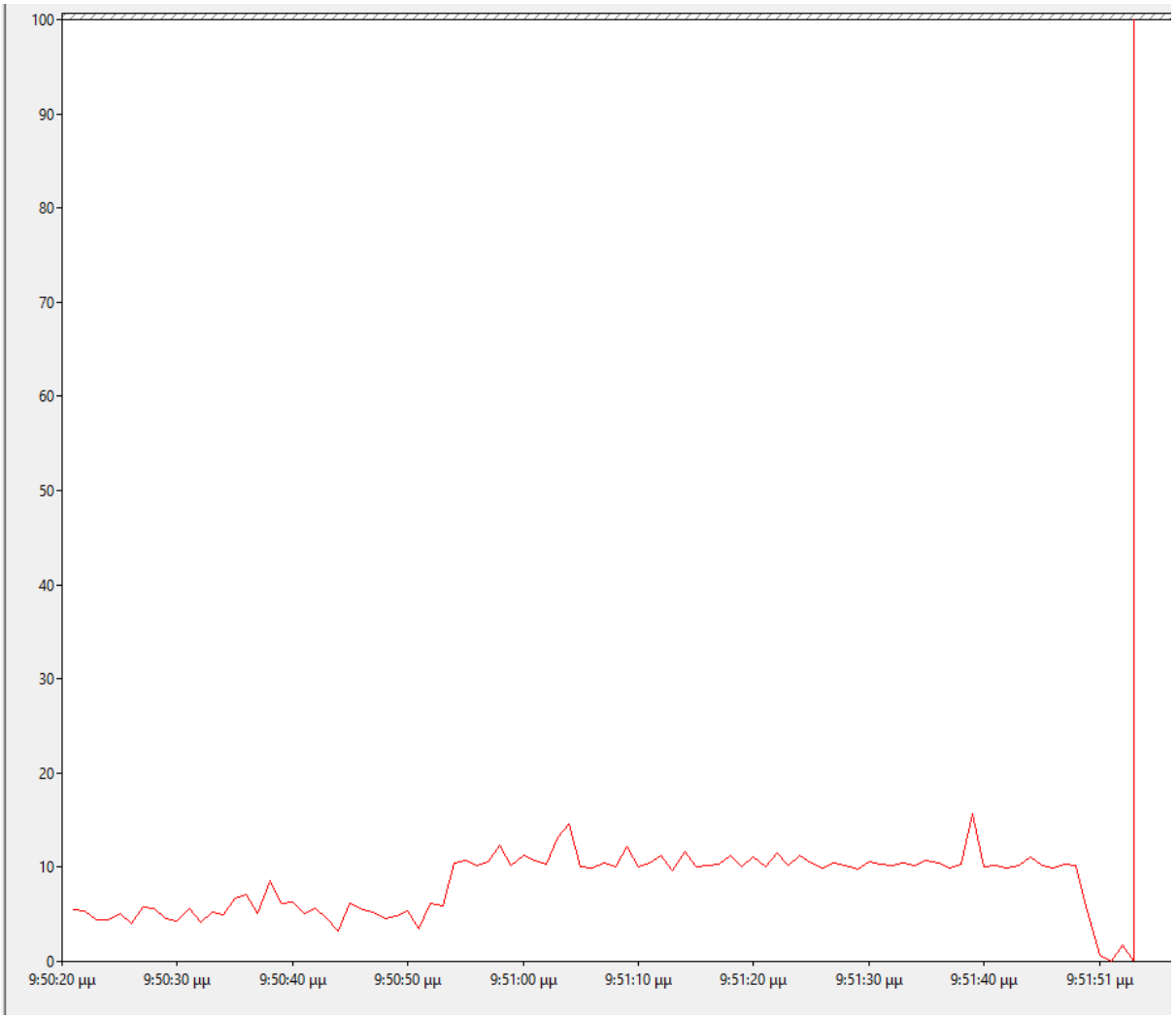
The following graphs are created while monitoring the performance of the program with the public workload. Each execution is taking advantage of additional partitioning whenever possible (which improves performance, as was evident in the previous tests). We used “Performance Monitor” to generate the graphs.

Ryzen 2600

Query Threads: 6 | Join Threads: 6 | NBHD: 16 | Bits: (8,10)

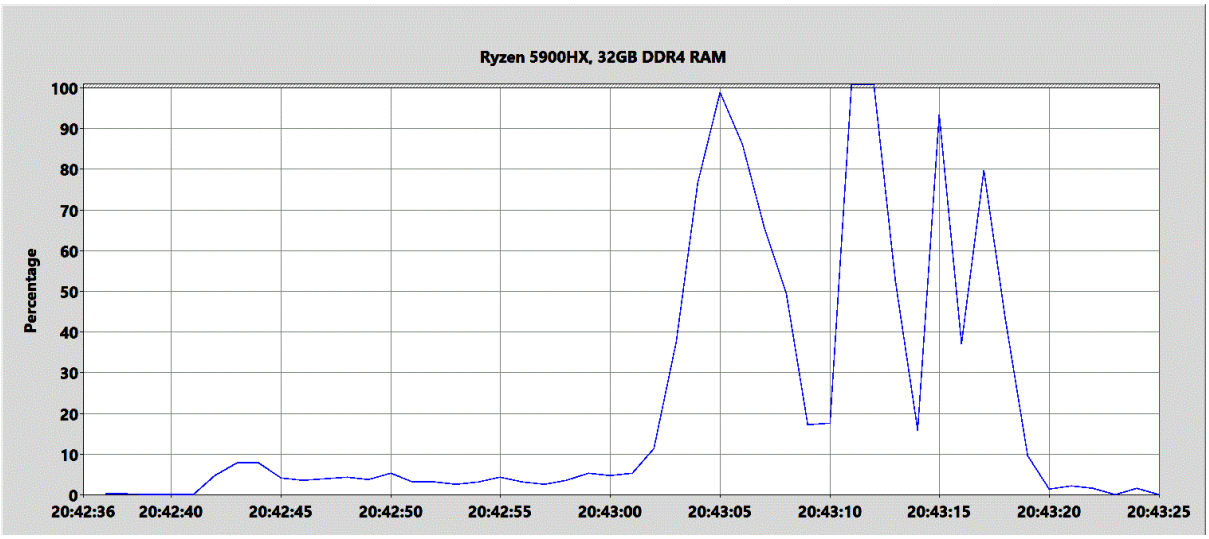


Query Threads: 1 | Join Threads: 1 | NBHD: 16 | Bits: (8,10)



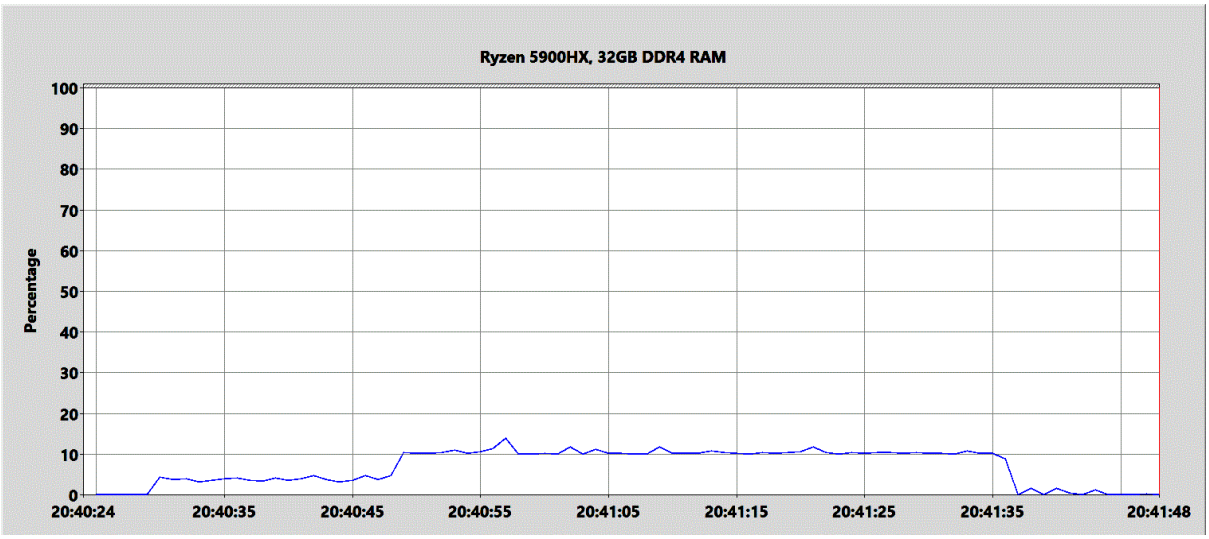
Ryzen 5900HX

Query Threads: 8 | Join Threads: 8 | NBHD Size: 16 | Bits: (8,10)



(33371ms)

Query Threads: 1 | Join Threads: 1 | NBHD Size: 16 | Bits: (8,10)



(64933ms)

As is to be expected, the CPU is being utilized significantly more during the execution of the process with multithreading, as opposed to the single threaded version. In the multithreaded version, a thread is easily kept busy, as it can start by executing all filter predicates and then start adding jobs for the threads dedicated to

joins to handle. This in turns leads to drastically improved execution times when multithreading is enabled.

RAM USAGE

16GB DDR4

Query Threads: 6 | Join Threads: 6 | NBHD: 16 | Bits: (8,10)
With additional partitioning

Highest recorded RAM usage with Query Optimizer : 7.3 GB

Highest recorded RAM usage **without** Query Optimizer: 13.5 GB

Query Threads: 1 | Join Threads: 1 | NBHD: 16 | Bits: (8,10)
With additional partitioning

Highest recorded RAM usage with Query Optimizer: 5.3 GB

Highest recorded RAM usage **without** Query Optimizer: 13.5 GB

32GB DDR4

Query Threads: 8 | Join Threads: 8 | NBHD Size: 16 | Bits: (8,10)
With additional partitioning

Highest recorded RAM usage with Query Optimizer: 8.15 GB

Highest recorded RAM usage **without** Query Optimizer: 15.1 GB

Query Threads: 1 | Join Threads: 1 | NBHD Size: 16 | Bits: (8,10)
With additional partitioning

Highest recorded RAM usage with Query Optimizer: 5.12 GB

Highest recorded RAM usage **without** Query Optimizer: 13.5 GB

It is evident that the query optimizer has a major effect on RAM consumption. By rearranging the queries, it allows for less intermediate results at a given time, therefore it manages to reduce memory usage. Without the query optimizer, the machine with 16GB RAM reaches maximum capacity and has to resort to swapping pages in and out of memory, which also greatly hinders performance.

CONCLUSION

Due to the complexity of today's systems, It is recommended that one experiments with different configurations in order to finally come up with the settings that offer maximum performance. This report focuses on numerous combinations and also proves that the performance of query execution can scale with the number of threads available. Moreover, it examines the impact of partitioning and how it can help with performance in more cases than expected.

Based on our testings, we can safely state that the following configuration offers optimal performance (or at least close to it) with our specific implementation:

- Threads: as many as the CPU threads. More specifically, a machine with hyperthreading seems to hit the sweet spot when using half of the available CPU threads for joins and the other half for query execution. This allows for a lot of flexibility, since a thread can execute all filters before finally reaching the join predicates and take advantage of the join threads.
- Bits: Using 8 bits for partitioning 1 and 10 if a second one is needed seems ideal in most cases.
- Neighborhood size: A size of 16 offers the best performance based on our testing.
- Partitioning: Partitioning large relations (even though they may still fit in the CPU cache) yields better execution times in almost every case.