NUMA obliviousness through memory mapping

Mrunal Gawade
CWI, Amsterdam
mrunal.gawade@cwi.nl

Martin Kersten
CWI, Amsterdam
martin.kersten@cwi.nl

ABSTRACT

With the rise of multi-socket multi-core CPUs a lot of effort is being put into how to best exploit their abundant CPU power. In a shared memory setting the multi-socket CPUs are equipped with their own memory module, and access memory modules across sockets in a non-uniform access pattern (NUMA). Memory access across socket is relatively expensive compared to memory access within a socket. One of the common solutions to minimize across socket memory access is to partition the data, such that the data affinity is maintained per socket.

In this paper we explore the role of memory mapped storage to provide transparent data access in a NUMA environment, without the need of explicit data partitioning. We compare the performance of a database engine in a distributed setting in a multi-socket environment, with a database engine in a NUMA oblivious setting. We show that though the operating system tries to keep the data affinity to local sockets, a significant remote memory access still occurs, as the number of threads increase. Hence, setting explicit process and memory affinity results into a robust execution in NUMA oblivious plans. We use micro-experiments and SQL queries from the TPC-H benchmark to provide an in-depth experimental exploration of the landscape, in a four socket Intel machine.

Categories and Subject Descriptors

H.4 [**Information systems**]: *performance measures*

Keywords

NUMA, memory mapped IO, multi-socket CPUs

1. INTRODUCTION

Most low end servers are equipped with two socket CPUs. In contrast, most high end servers tend to have four or eight socket CPUs, in a shared memory setting. The memory access latency in a two socket CPU is relatively low, however, in four and eight socket CPUs it is considerably expensive, if the memory being accessed is remote.

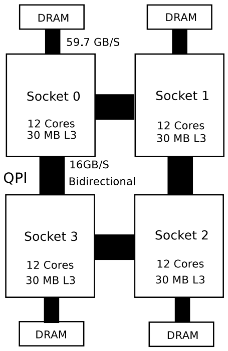


Figure 1: **Schematic diagram for Intel Xeon E5-4657LV2 @2.40GHz CPU**

Figure 1 shows a shared memory four socket CPU system where each CPU socket is associated with its own memory module (DRAM), and can also access a remote DRAM through Quick Path Interconnect (QPI) [3]. The memory access latency thus varies considerably based on whether the memory being accessed is local or remote. For example, the process residing on socket 0 accesses its local memory much faster than the remote memory on socket 2, as socket 2 is 2 hops away from socket 0. Non-uniform memory access (NUMA) [11] is thus a result of different memory access latency across sockets, in a shared memory system.

The graph in Figure 2 shows such an example for TPC-H Q1 (Scale factor 100 GB on four socket CPU). We plot an average of 6 runs (minimal variations are observed between consecutive runs), clearing the buffer cache between independent query executions. The database server process (using memory mapped storage) is allowed to execute strictly only on two sockets (0 and 1), by pinning the process's affinity to both sockets, using the tool *numactl* [5]. On the other hand, the memory allocation for the process is allowed to take place on different sockets (0 to 3), using numactl's memory binding option, to emphasize that the locality of data and the memory access distance matters, and affects the execution time.

When the memory allocation is local (socket 0 and 1), the execution time is lowest, as there is minimal cross socket data access. The execution time is highest when the memory allocation occurs only on socket 2, as memory on socket 2 is 2 hops away from socket 0, and 1 hop away from socket 1. The operating system does not allocate memory pages in an uniform manner across sockets, hence the part of the process executing on socket 0 tends to access more pages, compared to process execution on socket 1. This also explains why the

DaMoN'15, June 1, 2015, Melbourne, VIC, Australia
Copyright 2015 ACM ISBN 978-1-4503-3638-3/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2771937.2771948.

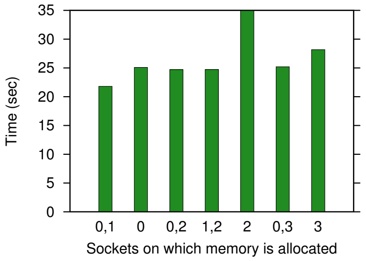


Figure 2: **Response time variations for TPC-H Q1 (100GB) on a 4 socket CPU, when the database server process is spawned across both sockets 0 & 1, while the memory allocation is varied between sockets 0 to 3.)**

execution is second highest when memory allocation occurs only on socket 3. Memory on socket 3 is only one hop away from process on socket 0, compared to the process on socket 1. From Figure 2 the execution time for the rest of the memory allocation affinities is more or less similar.

Database systems try to mitigate the data affinity problem in NUMA configuration by partitioning the data across CPU sockets either using range or hash partitioning[13]. For example, in a star schema, while the fact tables are horizontally range partitioned, the dimension tables being relatively small are replicated. As an example consider a 100 GB data set, where the data is horizontally partitioned in 25 GB piece each, affiliated with the sockets by introducing corresponding query plan partitions. The thread affinity is set to the corresponding sockets. This design however requires query plan level changes to introduce data location aware partitions in the plan, to maintain the data affinity to the sockets.

The observations from Figure 2 show minimal variations except for socket2. This motivates us to explore the viability of an alternate approach using memory mapped storage, to minimize the need for explicit data partitioning across sockets. Memory mapped IO uses the operating system's virtual memory infrastructure to control mapping of disk files to the memory. Memory mapped IO ensures that only the portion of the file gets loaded when its access is required. For example, during execution of a binary file the first step by the operating system is to do memory mapping of the disk file. As and when page faults occur, the corresponding portion from the file is brought to the memory based on the mapping. The same logic could be used to load data from disk files into memory such that locality with respect to sockets is maintained. Hence, we expect through memory mapped storage the operating system could offer an oblivious access to the data, while maintaining its locality with sockets, in a NUMA setting [4].

Consider the case of columnar database systems [16, 8, 10], where memory mapped storage could be used to represent in memory columnar data, backed on disk by a suitable file representation. During in memory data access the data is brought to the memory from disk and stays in memory as long as there is no need to swap it out. As an example of a possible mapping, consider the case when two select operators work on a column that is partitioned into two equal halves. If the first operator is scheduled to execute on the socket 0, then its data is mapped onto the memory module for the socket 0. Whereas, if the second operator's execution is scheduled on socket 1, then its data gets mapped on the memory module for the socket 1. The operating system thus tries its best to keep the data affinity to sockets, depending on the source of access.

Our main contributions are as follows.

1. We show NUMA oblivious query plans provide reasonable performance compared to NUMA aware partitioned plans.

2. We provide insights into the behavior of memory mapped columnar storage in a NUMA setting.

3. We show remote memory accesses lead to performance degradation in NUMA oblivious plans. Treating multi-socket system as a distributed database system results into minimal remote memory accesses, improving the execution upto 3 times.

The paper is structured as follows. In Section 2 we briefly describe the NUMA oblivious and NUMA aware plans. In Section 3 we provide the experiments to analyze the memory mapped IO behavior in a NUMA setting. In Section 4 we provide a perspective compared to a leading database system. Section 5 describes the related work. We conclude in Section 6 citing major lessons learned.

2. NUMA OBLIVIOUS VS NUMA AWARE PLANS

Columnar database systems are a good experimental platform since the columnar storage could be represented in a memory mapped file. MonetDB, the only open-source columnar database system is a good choice, since it uses a memory mapped columnar storage for base tables and intermediate data. It uses operator-at-a-time execution model, completely materializing the intermediate results.

We use three separate configurations to test the effect of NUMA oblivious data partitioning vs NUMA aware data partitioning. We describe these configurations next.

NUMA oblivious data partitioning: MonetDB uses a simple heuristic such that a parallel plan is generated from a serial plan by range partitioning the largest table in the plan. The number of equi-range partitions equal the number of the available cores. Operators in MonetDB plans operate on the range partitioned data, where they get scheduled on the available cores using the default operating system scheduling policy (CFS). This scheme represents multi-core intra-query parallelism, where data partitioning is done at plan level, without explicit socket knowledge. The operating system takes care of scheduling the operators on the sockets such that the memory affinity is maintained in a NUMA setting [11, 4]. This scheme thus does not involve any kind of explicit NUMA related optimization with respect to explicit horizontal data partitioning, and hence is termed as **NUMA_Obliv**.

NUMA aware data partitioning: To explore the effect of socket aware partitioned data access we use a modified implementation of MonetDB tailored towards the socket based data locality. The data is partitioned horizontally in 4 pieces such that the *lineitem* and the *orders* table are partitioned across sockets, while the rest of the tables are replicated. This modified implementation of MonetDB uses an optimizer that generates socket aware partitioned plans. Inspired by [20] we use MonetDB in a distributed master slave architecture. We name this implementation **NUMA_Distr**.

We assign one MonetDB server instance per socket which acts in a slave configuration, whereas a Master MonetDB server instance executes on any one of the four sockets. Thus we have a total of five MonetDB server instances, one of which is a master and the rest four are slaves. Slaves execute when master is not executing, hence the presence of a separate master does not involve resource sharing. The slaves carry out the execution of partitioned plan corresponding to their partitioned data, while the master is responsible for the final aggregation of individual results from each of the four slaves. Each one of the slaves in turn operates on an intra-query partitioned plan where maximum partitions are the number of cores per socket. The intra-query partitioned plans that each one of the slave uses are generated using the same NUMA oblivious parallel plan generation logic. Thus the NUMA_Distr mechanism essentially limits the access of data locally and prevents across socket interference.

We also use another variation of plans which are similar to NUMA_Distr plans in their physical representation, however in their execution behavior are similar to NUMA_Obliv. In this scheme a single MonetDB instance uses horizontally partitioned data (lineitem and orders tables) across four sockets. The parallel plans generated in this manner are socket aware, however since we do not use any kind of thread binding across sockets, the operating system is free to schedule the threads based on its default scheduling policy, thereby making them behave in a NUMA_Obliv configuration. We name this mechanism as **NUMA_Shard**, because it works on sharded data like in NUMA_Distr scheme, however without the master slave configuration. This configuration is used to overcome the partitioning problems in NUMA_Obliv configuration, that arises due to lack of partitions on the orders table. In Section 3.1 we elaborate it using TPC-H Q4 as an example.

Summary: Note that all these three configurations use memory mapped storage, as MonetDB uses memory mapped files to store columnar base and intermediate data. Though in NUMA_Distr separate database servers execute on each socket, the individual operators in the plan work on the memory mapped stored data, restricted to each socket. Hence, execution performance comparison of these techniques reflect the effect of memory mapping.

3. EXPERIMENTS

The hardware comprises of Intel Xeon E5-4657L v2 @2.40GHz with 4 sockets, 12 cores per socket for a total of 96 threads (Hyperthreading enabled), L1 cache=32KB, L2 cache=256KB, and shared L3=30MB, and 1TB four channel DDR3 memory where each socket is attached to 256 GB memory. The operating system is Fedora 20. The results are an average of 6 runs. The buffer cache is cleared [1] between successive query executions to allocate new memory mapped pages, to avoid interference from previously pinned pages. As MonetDB uses memory mapped columnar storage for base and intermediate data, memory mapping is always enabled for all three configurations, NUMA_Obliv, NUMA_Shard, and NUMA_Distr.

We use the tools *numactl* and *Intel PCM* to get insights into the effects of local vs remote memory accesses.

Numactl: We use numactl [5] to control the process and memory allocation affinity to individual sockets. An example command to set the database server process affinity to sockets 0,1 and memory affinity to socket 2 is as follows.

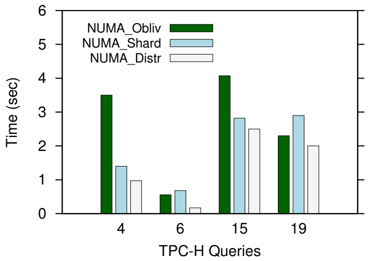


Figure 3: **Query execution performance of NUMA oblivious vs NUMA aware partitioned plans, for scale factor 100.**

```
numactl -N 0,1 -m 2 database_server_process
```

Intel PCM: We use Intel Performance Counter Monitor (PCM) tool [2] to measure the CPU performance events. PCM is different from frameworks such as PAPI [18] and Linux Perf [9] because it not only supports *core* but also *uncore* events. The uncore is the part of the processor that deals with integrated memory controllers, the Intel QuickPath interconnect, and the IO hub. We use the executable *pcm-numa* to measure the local and remote DRAM access, of cache-line size unit. Linux Perf [9] tool is used to measure the CPU migrations.

3.1 SQL query analysis

We use a subset of SQL queries from the TPC-H benchmark on a 100 GB data-set, to analyze their execution performance in NUMA oblivious vs NUMA aware partitioned plans. We then switch to micro-benchmark queries for a fine grained analysis of the observations from the SQL queries.

Setup: NUMA_Obliv setup uses a single instance of MonetDB with varying number of threads, executing on 96 cores, with default operating system scheduling policy (CFS). The NUMA aware plan execution setup (NUMA_Distr) uses four instances of MonetDB with 24 threads each, bound to each one of the four sockets, using *numactl* tool. The client invokes queries on a separate MonetDB instance which acts as a master. The other NUMA aware plan setup (NUMA_Shard) also uses a single instance of MonetDB with varying threads, on 96 cores, with sharded lineitem and orders table. The dimension tables are not replicated.

Figure 3 shows query execution time comparison for selected TPC-H queries. We use this query set as it provides sufficient insights into the overall behavior of the techniques under comparison.

The first observation is NUMA_Distr shows the best execution time in all the queries. This is expected because of minimum cross socket interference due to master slave configuration.

Next we analyze individual queries by focusing on Q6 first, where NUMA_Distr shows around 3 times improvement compared to the other two configurations. Both NUMA_Obliv and NUMA_Shard show similar times. Q6 has a single lineitem table with only select operations, which get parallelized easily. The difference in execution with NUMA_Distr is due to cross socket traffic, as both NUMA_Obliv and NUMA_Shard does not have explicit data affinity in plans, as their threads get scheduled according to the default operating system scheduling policy.

NUMA_Obliv shows highest time for Q4, due to MonetDB's parallel plan generation limitation. Q4 has both the lineitem and the orders table. Orders table is the second largest table in TPC-H after lineitem table. As MonetDB

[1] echo 3 | sudo /usr/bin/tee /proc/sys/vm/drop_caches

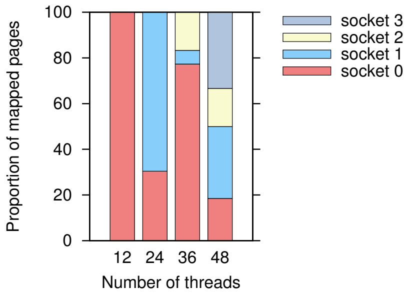
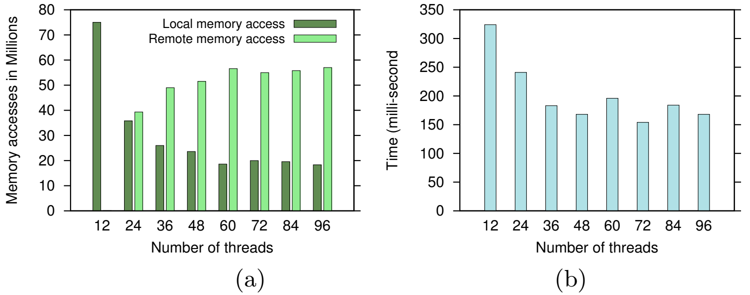


Figure 4: **Proportion of memory mapped pages on each socket when threads and memory allocation per socket is increased by including sockets one by one, using numactl, for modified Q6.**



(a) (b)

Figure 5: **a,b)Process and memory affinity to sockets controlled using numactl, for modified Q6. Buffer cache cleared.**

optimizer partitions only the largest table for generating a parallel plan, only the lineitem table is partitioned. Q4 has a join on the lineitem and the orders table attribute, which contributes to the lengthy execution as the orders table is not partitioned. In both NUMA_Shard and NUMA_Distr versions both the lineitem and the orders table are partitioned, which explains why both of these configuration are much faster.

Query 19 illustrates the effect on a more complex query over two tables, the lineitem and the part table. However, in NUMA Shard only the lineitem table is being partitioned into four pieces using row-id ranges. The part table is not partitioned. This implies that all cores involved in the join will randomly access the part table, increasing the intra-core memory accesses. A hash-based partitioning could alleviate this overhead, but is currently not part of the MonetDB standard repertoire.

As observed from Q6 and Q19 we expect NUMA_Obliv configuration to be competitive to NUMA_Distr configuration provided majority of the tables in the plan are correctly partitioned. The NUMA_Shard configuration is used just to prove this point, since otherwise as could be seen in Q4, NUMA_Obliv execution looks too expensive. For a drilled down analysis of the effect of the memory mapped IO in a NUMA setting, we focus on Q6.

Why focus on Q6? Q6 is a simple query with select operation on the largest table, lineitem. Parallelized select operators are expected to execute with memory affinity maintained to sockets due to memory mapped storage. Hence, analyzing it gives a baseline to analyze the numa effects. We hypothesize that the difference in timings for Q6 (See Figure 3) is due to cross socket interference. This is confirmed from Table 1 which shows that NUMA_Distr has much less number of remote memory accesses compared to NUMA_Obliv. Hence, we investigate where do these remote memory accesses arrive and if they could be curtailed to improve the NUMA_Obliv execution further.

Table 1: Q6 memory accesses (cache line size unit).

	#Local accesses	#Remote accesses
NUMA_Obliv	69 Million (M)	136 M
NUMA_Distr	196 M	9 M

3.2 Micro-experiments

We use a modified Q6 from the TPC-H benchmark. Q6 operates on the largest table *lineitem*. The query is modified to have only a single select operation, without any output. It allows us to experiment with the read only aspect of the memory mapped IO. The query is as follows.

```
select count(*) from lineitem where l_quantity > 24000000;
```

The select operator acts as a good example to demonstrate the effects of memory mapped IO in a NUMA setting. It is easily parallelizable by range partitioning of the data, such that each partition is operated upon by one select operator. Since each partition uses a memory mapped storage representation, we hypothesize that the operating system would schedule select operators on sockets, such that the data affinity is maintained, resulting into NUMA obliviousness.

To test our hypothesis we control the socket allocation for memory and database server process execution. The 4 socket CPU has 12 physical cores and 12 hyper-threads per socket, in the following order.

Table 2: CPU core allocation across sockets.

	Socket 0	Socket 1	Socket 2	Socket 3
Cores	0-11	12-23	24-35	36-47
Cores	48-59	60-71	72-83	84-95

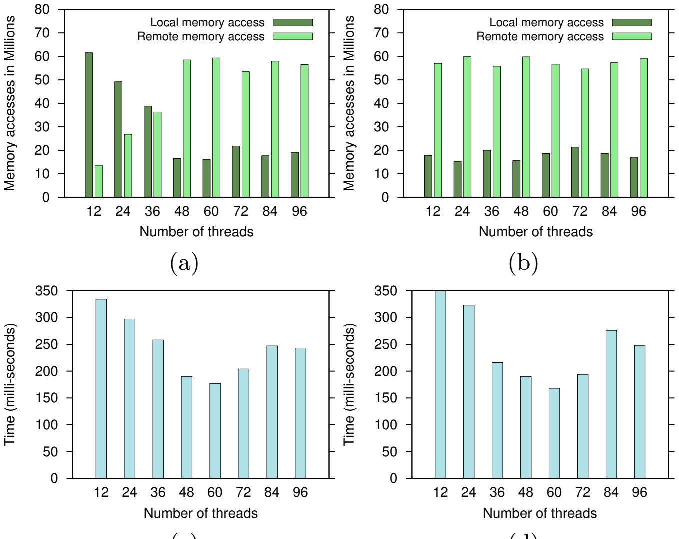
3.2.1 *Execution with numactl affinity setting*

Setup: The graph in Figure 5a quantifies the remote vs local memory accesses, when process execution and memory allocation affinity is set using numactl. The process affinity is set on the sockets in steps of 12 threads, as per the core order in Table 2. The memory affinity to sockets is also allocated in increments of one socket each. For example, when 12 threads execute on socket 0, the memory allocation is also pinned to socket 0, whereas when 36 threads execute on three sockets (as per core order in Table 2), the memory allocation is pinned to three sockets. The corresponding command for 36 threads is as follows.

```
numactl -C 0-11,12-23,24-35 -m 0,1,2 Database_Server
```

First observation in Figure 5a is when 12 threads execute on the socket0, their remote memory access is almost negligible, and the entire memory access arrives from the local memory. As the number of threads increase across the sockets, the local memory access decreases, while the remote memory access increases until 60 threads are in use. Note that in Table 2, hyper-threads form the range 48-95. Hence, 60 threads on-wards both remote and local memory accesses almost stabilizes. Figure 5b, shows the corresponding execution time, which also almost stabilizes after 60 threads of execution.

Figure 4 shows the proportion of pages mapped on each socket as the number of sockets increase. Consider the case, when 24 threads execute on the socket0 and the socket1. Since only around 30% pages are mapped on the socket0, almost two third of the threads executing on the socket0 do remote memory access to access pages from the socket1, rest do local access. Since socket1 has all its pages mapped, we expect all threads on socket1 to do local access. This indicates the remote access is around 1/3rd of the total page ac-



(a)

(b)

(c)

(d)

Figure 6: **Without process and memory affinity to sockets. a,c) Buffer cache cleared. b,d) Buffer cache not cleared.**

cesses, while the local accesses are 4/6th of the total accesses. However, the numbers from the 24 thread case, in Figure 5a does not reflect it. Remote accesses are higher, than local accesses. Digging deeper in */proc/process_id/numa_maps* shows some of the remote accesses also arrive from the memory mapped libraries for the database server process.

We also expect some of the remote accesses to arrive due to cache coherency and thread migrations. As the number of threads increase, their migrations across sockets also increase, as shown in Figure 7. Numactl just prescribes the affinity across sockets, but at run-time the operating system is still free to do migrations to do load balancing [4]. For example, when process affinity is pinned to socket0 and socket1, operating system will not schedule threads on socket2 and socket3, however, it is free to migrate threads across socket0 and socket1, if the need arises. This explains why with an increase in the number of threads, the remote memory accesses increase, while the local memory accesses decrease.

3.2.2 *Execution without explicit affinity setting*

Setup: Both Figures 6a and 6b show the number of local vs remote memory accesses, when the process or memory affinity is not set using numactl. The only difference being in 6a after each independent run, [2] the buffer caches are cleared using a kernel utility [3]. This is a crucial setting as if caches are not cleared memory mapped pages might stick around on previously allocated sockets, preventing their new locality based allocation. Figures 6a and 6b makes the difference prominently visible.

Figure 6a shows a pattern similar to Figure 5a, where the local memory accesses decrease with an increase in the number of threads. However, the change of both local and remote accesses for 12,24,36, and 48 threads in 6a is gradual, compared to a sudden change in Figure 5a. This indicates that when explicit affinity is not set, and when buffer caches are not polluted, the operating system does a good job of executing the process on sockets to maintain data locality. Both in 6a and 6b, the memory access pattern stabilizes when execution also uses hyper-threads starting from 60 threads, and almost matches the memory access pattern in Figure 5a.

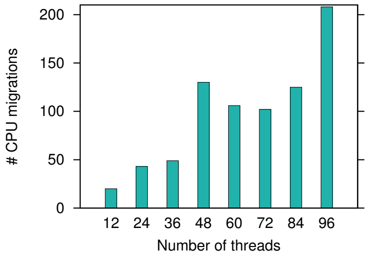


Figure 7: **Number of CPU migrations increase as the number of threads increase, for modified Q6.**

Consider the case of 12 threads from Figure 6a, where the number of local accesses are less compared to Figure 5a. This is a result of the lack of process and memory affinity in Figure 6a execution. An important observation is unlike Figure 5a, the local memory accesses stay dominant than remote memory accesses until 36 threads, which indicates the operating system does an overall good job of scheduling. However, this does not get reflected accordingly in the execution times from Figures 5b and 6c, where for 24 and 36 threads, Figure 6a execution time should have been better than 5a. We are unable to explain this behavior.

Figure 6b offers an interesting perspective as well, as it shows without setting process and memory affinity to sockets, and without buffer cache cleared, both local and remote accesses almost stay constant irrespective of the number of threads in use. However, the execution time (See Figure 6d) does change and shows best time of around 150 sec, when 60 threads are in use. This seems very good as it indicates, without much efforts, just by choosing the correct number of threads, better execution could be obtained. However, finding the correct sweet spot in terms of the number of threads might not be that easy [1]. On the other hand, execution time for Figure 5a almost stabilizes after 48 threads are in use, which seems like a more robust approach.

Summary: Overall, we conclude that setting explicit process and memory affinity in NUMA oblivious plans, leads to a more robust execution as seen from Figure 5b, where the execution time stabilizes after 48 threads are in use. However, execution without process and memory affinity, without clearing buffer cache seems a more practical approach, and Figure 6d shows, it does offer similar execution time, but finding the exact number of threads to get the best execution could be tricky [1]. We also observe that the presence of hyper-threads has a negligible effect on the number of local and remote memory accesses.

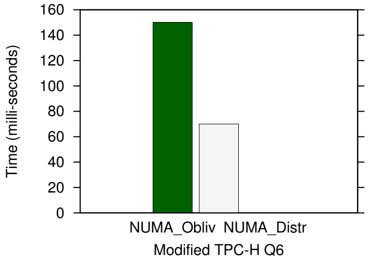


Figure 8: **More remote memory access in NUMA_Obliv slows down execution of modified Q6 by 2 times, compared to NUMA_Distr.**

Why remote memory access is bad? From Figure 8, the execution performance of modified Q6 in NUMA_Distr configuration is two times better than the NUMA_Obliv configuration. This indicates NUMA_Obliv shows relatively

[2] An independent run occurs after 6 runs on the same configuration to take an average.

[3] echo 3 | sudo /usr/bin/tee /proc/sys/vm/drop_caches

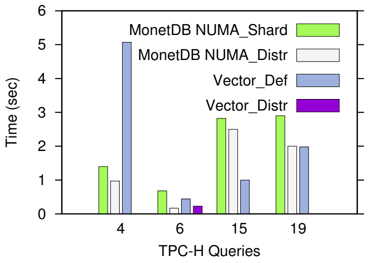


Figure 9: **Vectorwise's parallel execution compared to MonetDB's parallel execution (NUMA_Distr), for scale factor 100.**

good performance overall. The loss of performance could be mainly attributed to the high number of remote memory accesses in NUMA_Obliv. This could be verified as follows. When the memory access is prominently local as in the case of 12 threads (See Figure 5a), the execution time is around 320 ms (See Figure 5b). If we divide the time by 4, since there are 4 sockets, the new time per socket is 80 ms, which matches with the NUMA_Distr execution of modified Q6 from Figure 8.

4. PIPELINED EXECUTION COMPARISON

Comparable performance is a subjective term. In our context we consider upto 4 times difference as a comparable performance, whereas an order of magnitude improvement is considered worth the effort of a new system design.

Vectorwise (version 3.5) is a leading column store analytic system that uses pipelined vectorized execution. As it uses a dedicated buffer manager, rather than memory mapped storage, a comparison with Vectorwise (See Figure 9) provides a perspective of the possible role of NUMA in its execution performance. The only configuration change we made is to enable histograms to generate better plans, and set parallelism level=96.

Vector_Def is the single instance default parallel execution without NUMA awareness and without affinity control. We compare it with MonetDB's NUMA_Shard configuration in Figure 9. Note that NUMA_Shard has just the lineitem and the orders table sharded and represented accordingly in the plan, however, the plan itself does not have any socket affinities as a single database instance is used. Hence NUMA_Shard configuration also represents NUMA oblivious execution (as underlying storage is memory mapped), which we compare with Vector_Def configuration.

Vector_Def is relatively faster except for Q4, which hints at the executions with a NUMA oblivious buffer manager perform better than the executions with memory mapped buffers, as in MonetDB. We hypothesize that Vectorwise performs better even without NUMA awareness, due to its pipelined vectorized execution, and making it NUMA aware could improve its execution further. Depending on the query we observe a wide range of multi-core utilization (as reflected in CPU idleness using the *top* command), which we believe is a result of the cost model based parallel plan generation in Vectorwise. In [6] the authors illustrate problems of Vectorwise scalability beyond 8 cores due to locking and synchronization related overheads. We believe a NUMA aware approach similar to NUMA_Distr would benefit systems like Vectorwise to scale further, as it incurs minimal changes at the architectural level.

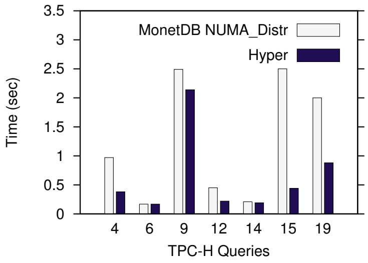


Figure 10: **Hyper's parallel execution compared to MonetDB's parallel execution (NUMA_Distr), for scale factor 100.**

Vectorwise does not have a NUMA aware plan generation. As we do not have source code access to implement it, to get a perspective of the NUMA aware partitioned execution, we partition the lineitem table in four pieces. As query 6 is the simplest parallelizable query with select operations on the lineitem table, we measure its execution time on each of these pieces, and plot the highest time as Vector_Distr. We expect minimal aggregation overhead as the only aggregation operation is the sum of the four numbers from the four sockets, which is negligible compared to the individual select operation's time. Compared to Vector_Def, we observe an execution improvement of around 2 times.

Hyper's morsel driven parallelism is NUMA aware, where morsels from hash partitioned data are fed to the just in time compiled fused operator pipelines. From [13] Q6 takes 0.17 sec on 100 GB data-set on a 64 core (hyper-threaded) four socket, Intel Xeon X7560 @ 2.3GHz machine, with maximum QPI hop=1. While Hyper's Q6 execution time is same as NUMA_Distr, its Q4 and Q19 execution is just 2 times faster, which could be due to less QPI traffic during join, due to hash based data partitioning in Hyper. Hyper is designed from scratch for optimal multi-core utilization and uses LLVM [12] generated just in time compiled fused operator pipelines. However, LLVM code generation also makes its code base much complex.

In contrast, the promising query execution time for the queries 4, 6, and 19 by MonetDB's NUMA_Distr approach prompted us to explore more queries. We plot their execution time in Figure 10. It shows that the query execution performance of MonetDB's NUMA_Distr approach is comparable to Hyper's parallel execution performance for the query set under evaluation. In Figure 10 MonetDB uses 96 threads in total. To match Hyper's hardware configuration we restricted MonetDB's execution to 64 threads. Even with this change MonetDB's NUMA_Distr numbers do not show much variations compared to execution times in Figure 10.

Overall, considering its simplicity, NUMA_Distr approach looks promising for existing database architectures to control the problem of remote accesses, which results into a lower execution performance.

5. RELATED WORK

In [15] the authors evaluate the memory performance of NUMA machines. One of the main findings is how guaranteeing data locality to sockets need not be optimal always, due to increased pressure on local memory bandwidth. Authors provide use cases to show how a balance of remote and local memory accesses tend to balance out bandwidth for an optimal performance. Our calculations indicate for the Figure 5a, a local bandwidth of upto 15 GB/sec for 12 threads, and a cumulative remote bandwidth of upto 20GB/sec in 48 threads. In [17] authors also offer a detailed evaluation of memory performance in NUMA machines.

In [20] the authors treat the multi-socket system as a distributed system of individual database servers, in a master slave configuration. However, unlike our analytic workload, authors primarily explore the transactional workloads, from throughput and client scalability perspective. Authors first elaborate how the traditional databases like MySQL and PostgreSQL fail to scale with NUMA systems and then propose a new middle-ware based system called Multimed that solves this problem, by using multiple database instances in a master-slave configuration. Data is replicated across all slaves, such that read only queries are handled by slaves, whereas the master handles update queries. Resource contention due to latching and synchronization in multi-cores is avoided by using multiple satellite database servers, instead of a single server. In our case by using multiple database servers affiliated with individual sockets, we try to minimize the remote memory accesses in analytical workloads, which we show is the prominent reason for the decreased query execution performance.

In [19] the authors treat sockets as hardware islands and explore the effect of different shared nothing database deployments from transactional workload perspective. The work is done in the context of SHORE-MT transactional system with a distributed transaction coordinator using two-phase commit protocol. Different possible deployment configurations are considered with different possible island formations, to explore its effects on the throughput of the transactional workloads by varying parameters such as the database size, granularity of partitions, skew, etc. This work is different from our work because we use analytical workloads which have different characteristics with long running queries, unlike transactional workloads where the queries are short and access a few rows only. We emphasize on improving the response time of individual queries by using query parallelization by range partitioning the data, while the transactional workloads prominently emphasis on the overall throughput. While the authors of [19] experiment with different possible database deployment sizes with different granularity of partitions, we use only two deployments, namely shared-everything (NUMA_Obliv & NUMA_Shard) and shared-nothing (NUMA_Distr), where we partition the large tables (lineitem & orders) in 4 partitions across the 4 sockets.

The NUMA architecture also influences new operating system designs. A new architecture, called multi-kernel [7] treats NUMA machine as a small distributed system of processes that communicate via message passing.

In [14] the authors show the case of NUMA aware algorithms, with a focus on data shuffling. A lot of work also focuses on NUMA aware operators, such as joins.

6. CONCLUSION

We analyzed the role of memory mapping in a NUMA system, by comparing NUMA oblivious vs NUMA aware plan execution in a database engine that uses memory mapped columnar storage. NUMA oblivious plans that execute using the operating system's default scheduling policy show relatively good performance. Remote memory accesses are identified as the main culprit in NUMA oblivious plans. When the database engine is used in a NUMA aware configuration by treating multi-socket CPU as a distributed system, remote memory accesses are minimal, leading to upto 3 times improvement on the TPC-H queries tested. For the query set under evaluation, the distributed system based NUMA aware approach competes with the parallelism approach by the state of the art systems such as Hyper.

7. REFERENCES

[1] Adaptive query parallelization in multi-core column stores- under submission vldb 2015. https://sites.google.com/site/mrunalgawade/AdPar.pdf.

[2] Intel pcm tool. https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[3] Intel qpi. http://www.intel.nl/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[4] Numa scheduling progress in linux. http://lwn.net/Articles/568870/.

[5] Numactl. http://linux.die.net/man/8/numactl.

[6] K. Anikiej. Multi-core parallelization of vectorized query execution. http://homepages.cwi.nl/ boncz/msc/2010-KamilAnikijej.pdf.

[7] A. Baumann et al. The multikernel: a new os architecture for scalable multicore systems. In *Proc of SIGOPS*, pages 29–44. ACM, 2009.

[8] P. A. Boncz et al. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[9] A. C. de Melo. The new linux perf tools. In *Slides from Linux Kongress*, 2010.

[10] A. Lamb et al. The vertica analytic database: C-store 7 years later. *Proc of VLDB*, 5(12):1790–1801, 2012.

[11] C. Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40, 2013.

[12] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[13] V. Leis et al. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proc of SIGMOD*, pages 743–754. ACM, 2014.

[14] Y. Li et al. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[15] Z. Majo and T. R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proc of Systor*, page 12. ACM, 2011.

[16] S. Manegold et al. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proc of VLDB*, 2(2):1648–1653, 2009.

[17] C. McCurdy et al. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Proc of ISPASS*, pages 87–96, 2010.

[18] P. J. Mucci et al. Papi: A portable interface to hardware performance counters. In *Proc of HPCMP*, pages 7–10, 1999.

[19] D. Porobic et al. Oltp on hardware islands. *Proc of VLDB*, 5(11):1447–1458, 2012.

[20] T.-I. Salomie et al. Database engines on multicores, why parallelize when you can distribute? In *Proc of Eurosys*, pages 17–30. ACM, 2011.