Towards NUMA-Aware Distributed Query Engines

Alessandro Fogli, Peter Pietzuch
Imperial College London

Darko Makreshanski
Oracle

Jana Giceva
TU Munich

***Abstract*—Efficient execution of distributed queries in the cloud is of paramount importance, both for cloud vendors and their customers. In this paper, we investigate the performance impact of different deployment policies of distributed query engines over multicore machines.**

We corroborate prior observations that traditional data systems have limited scalability on modern machines. Since a complete redesign to make them hardware-conscious can be prohibitively expensive, we explore whether treating the machine as a distributed system underneath can in fact bring performance advantages, while being transparent to the query engine itself.

Our key observation is that, for a range of popular distributed query engines (SparkSQL, Presto, Greenplum, SingleStore), a deployment policy that maps an engine's worker instances to the compute and memory resources of a NUMA node can bring around 2× performance improvement for the TPC-H workload (SF 100) over a standard deployment.

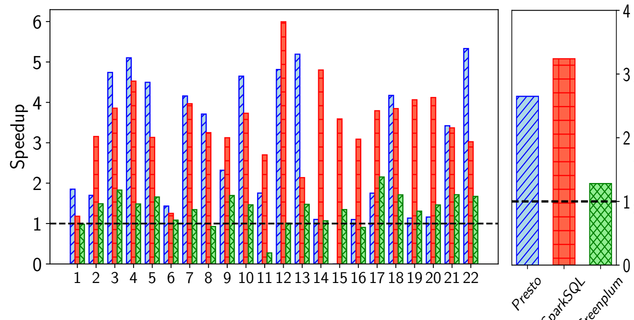
I. Introduction

Efficient query execution in the cloud is critical both for cloud vendors and their users. At large scale, even small percentage improvement in how efficiently one uses the underlying hardware resources bring considerable cost savings for anyone offering data analytics services in the cloud.

The most common approach to create a distributed query engine is to begin with an existing engine design and evolve it into a distributed version by addressing the scalability, fault tolerance and elasticity requirements of a competitive cloud solution [2], [3], [5]. As a result, there has been considerable research into how to design efficient distributed query optimizers [72], [73], optimal data and work partitioning among the database worker instances [42], [74], [75], while, e.g., minimizing the amount of data transfer [37], [46].

However, there has been relatively little attention paid into *how efficiently deployments of distributed query engines use resources* on such a scale. In fact, existing commercial systems use different approaches for mapping their worker instances to hardware resources: e.g., *Redshift* partitions resources into slices made up of individual CPU cores and assigns computation nodes such to slices [5]–[7]; in contrast, *SingleStore* partitions computational resources according to the number of NUMA domains, thereby fully leveraging the internal parallelism of the base engine to exploit multi-threading [2]; other systems, e.g., Presto [3], SparkSQL [4], etc., exploit resources in their entirety, assuming a full machine for each instance.

This raises the question if there is a single deployment approach for distributed query engines that fits all scenarios: if yes, which of these configurations is optimal? If no, which factors are relevant when choosing the most suitable deployment



(a) TPC-H query speedup (SF 100). (b) Geo-mean.

Fig. 1: A single-machine performance improvement achieved by applying NUMA's conscious deployment over the standard deployment of NUMA-agnostic distributed query engines (Greenplum (green), SparkSQL (red), and Presto (blue)).

strategy? How much efficiency can be gained by deviating from a *default* deployment strategy for each of the analyzed systems?

To address the above questions, we explore how resources should be allocated to distributed query engines in order to get the most performance out of modern multiprocessor (NUMA) systems. We perform a systematic experimental study that investigates and answers the following three questions:

1) What is the optimal number of worker instances of a database query engine to run on a given multiprocessor machine?
2) How to assign the computational and memory resources to these worker instances?
3) How to place data to ensure both data locality and low communication between worker instances across machines?

We evaluate overall engine performance both within a single multicore/NUMA machine and when using of a cluster of such machines. We also investigate the impact of the chosen workloads and datasets (e.g., in the presence of data skew), as well as the importance of the underlying machine topology and its hardware properties.

Based on our experiments, we identify a deployment strategy that works best for all systems evaluated. Our key results, shown in Figure 1, indicate that, by applying a NUMA-conscious deployment policy, we can get performance improvements of up to 3.2× for queries executing on the same hardware, when compared to more naïve, yet commonly used, baseline approaches that use a database worker instance per machine or per core. Some of our other key insights are:

- Per-machine deployments fail to fully exploit the abundant parallelism of multiprocessor systems. Deploying a database worker instance per NUMA domain often shows better multicore scalability, at times even close to linear.
- The bottleneck caused by congested interconnects in a per-machine deployment can be eliminated when running multiple database worker instances on a machine. The communication shifts across the network (TCP loopback), and the optimizer generates plans that minimize data transfer.
- Deploying a database worker instance per NUMA domain shows performance benefits even when applied to skewed data (JCC-H benchmark) and when executed on different hardware. Unsurprisingly, the observed benefits are dependent on the hardware properties, such as the interconnect bandwidth.
- Deploying a database worker instance per NUMA domain with a corresponding memory binding policy shows good horizontal scalability, and the speedup improves with the number of machines. Using multiple network interface cards (NICs) can easily increase the bandwidth, overcoming a possible network bottleneck.

Our recommendations are directly applicable to on-premise distributed query engines, enabling them to maximize resource utilisation transparently, without necessitating cumbersome changes to the engine's design. For engines deployed in the cloud, our insights can potentially lead to better provisioning of hardware resources and their assignment as containers. Our study of deployment approaches does not only bring significant performance-cost savings, but it can also influence how we design and configure future cloud-native data systems.

II. Background and Motivation

To discuss the trade-offs between the different deployment configuration policies for distributed query engines, we first introduce relevant concepts. We begin by giving an overview of typical hardware setups. On-premise and data-center level hardware resources are organized in racks.

While software-defined data-centers with resource-disaggregation over fast network interconnects pick up momentum, as they cater to a wide range of workloads [56]–[58], our work primarily focuses on a more traditional setting in which commodity multi-processor/multi-core machines are stacked in a rack, connected through a network interconnect (see Figure 2). Modern multi-processor machines follow a non-uniform memory access (NUMA) approach ③, with each CPU core exhibiting different access times to memory depending on the hop-distance of the memory node. Accessing local memory is faster than accessing remote memory, and this non-uniformity poses a challenge to existing systems when it comes to efficient scaling and fully leveraging hardware capabilities [37], [39], [41].

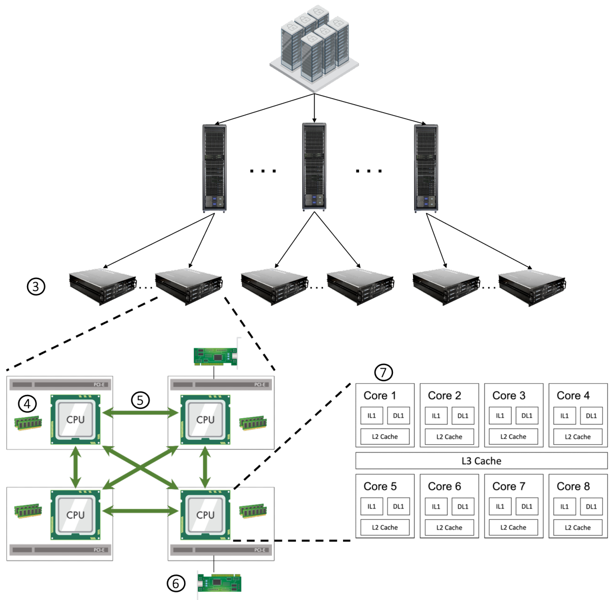


Fig. 2: Data center organization

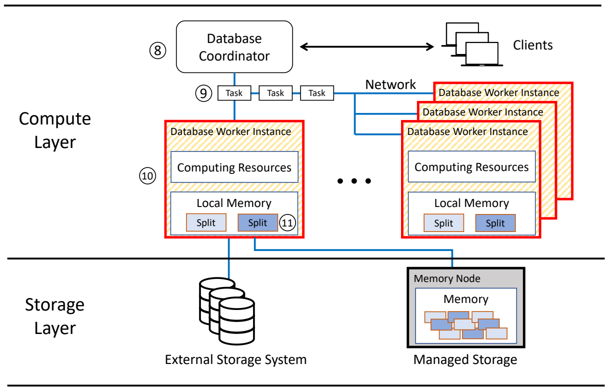


Fig. 3: Distributed query engine architecture

A. *Architecture of distributed query engines*

From a structural and design perspective, many distributed cloud-based query engines (e.g., AWS Redshift [8], Athena [11], Google's Big Query [10], Microsoft's Polaris [9]) have similarities with both on-premise data warehouses (e.g., Exadata [12] and Teradata [13]) and big data systems (e.g., Hadoop [14], Presto [3] and Spark [4]).

As shown in Figure 3, systems separate the compute from the storage layer to support scaling out and to allow for cost-effective customizations of each layer [8], [50], [51]. Recent systems even propose to further separate state so that the engine can better address the new challenges of different cloud models, e.g., running on reserved cloud resources or serverless, better support fault tolerance, or react to workload changes [9].

Regarding the compute layer, we assume that it consists of a fixed set of *database worker instances* ⑩, which execute the SQL queries (i.e., a pipeline of the query [9], or a big data job). In the literature, database worker instances are also referred to as worker/leaf/segment nodes or query processes [5]. They operate on *data fragments*, which are assigned to each database worker instance, also called a *split* ⑪. The worker instances process many splits in parallel by executing a dedicated list of tasks, as assigned by the coordinator ⑧. The coordinator (also referred to as the *leader* or *master* node) is often responsible for admitting, parsing, planning and optimizing the queries, in addition to orchestrating how the query's tasks ⑨ are distributed across worker instances.

The worker instances communicate with each other over the network, either to exchange data when performing joins over multiple tables or to broadcast the results to the coordinator. The general rule is to avoid expensive network communication as much as possible [1]. Hence, each database worker instance processes its own queue of tasks on its own data partition (splits), which both maximizes data locality and reduces network traffic. The coordinator computes an optimal distributed cost-based query plan, which accounts for all factors involved, including data distribution (to minimize the overhead of communication among the database worker instances) and data placement within multi-processors.

B. Data placement within multi-processors

Choosing an appropriate thread- and data-placement policy for a NUMA machine is a non-trivial task for any system software. The Linux scheduler, for example, can migrate threads and data in order to achieve a good work balancing on each CPU core or across the memory controllers [15], [16]. However, such a policy in a noisy environment can result in significant performance drop, especially for data-intensive tasks [17], [18]. As an alternative, user-space libraries such as *libnuma* enable applications to override the default system policies ④ to ones that are more appropriate when having knowledge of the workload. The *numactl* tool provides several memory allocation policies to choose from:

(1) First Touch (FT) is the default policy in modern Linux systems. In Linux, memory is not allocated at the time of a call but at the time of first access. When a process performs a write/read operation on a memory page for the first time, the page is allocated to the NUMA node to which the process belongs. If the node does not have enough free memory, an adjacent node is used.

(2) Interleaving (INT) allocates memory in a round-robin fashion across all NUMA nodes available in the system. It generates an even memory load across all the NUMA nodes and interconnects.

(3) Membind (MEM) allows processes to bind to specific NUMA nodes. Once specified which memory a process should use, all data allocated or generated by the process is assigned entirely to the memory of the designated node.

C. NUMA effects on data processing

In this section, we analyze the impact of NUMA affects on the query engines explored in the evaluation. We run the TPC-H benchmark and monitor the amount of remote memory accesses and the load on the interconnects during query execution, both of which can seriously impact performance [27], [28]. Interestingly, the systems tested exploit different data- and thread placement policies, leading to diverse results.

TABLE I: Profiling results for TPC-H Q13 obtained by Intel's VTune Profiler. (QPI Bandwidth is the percentage of elapsed time with a high use of interconnections.)

System	# DRAM accesses	Remote accesses	QPI bandwidth
Presto	30,564	76	2
SparkSQL	9,165	73	47
SingleStore	7,343	0	0
Greenplum	5,880	70	0

Table I shows the outcome of profiling TPC-H query 13 (Q13) with a scale factor 100. The benchmark executes on a single machine with 4 NUMA domains, and we evaluate Presto, SparkSQL, SingleStore and Greenplum. For each system, we use the deployment policy from its documentation. We focus on Q13, because it is one of the longest running queries and has a high transient memory consumption; it is also less affected by plan optimizations [76].

We observe that its execution on Presto, SparkSQL and Greenplum generates a large number of expensive remote memory accesses. With Presto and Greenplum, this mainly happens during the initial scanning phase of tables. SparkSQL uses a SortMerge Join for query execution, so remote memory accesses also occur during the sort phase. In all three systems, the data is placed across all NUMA domains, and each task has the ability to process both local and remote splits.

The results in Table I also indicate that, when executing Q13 with SparkSQL, the machine's interconnects are congested 47% of the query execution time. This is an artefact of not placing the data in a balanced fashion among the NUMA domains. Since a large portion of the data resides on a single NUMA domain, most data accesses are directed there, creating congestion over the interconnect. In contrast, Presto and Greenplum balance data between NUMA domains, thus avoiding congestion.

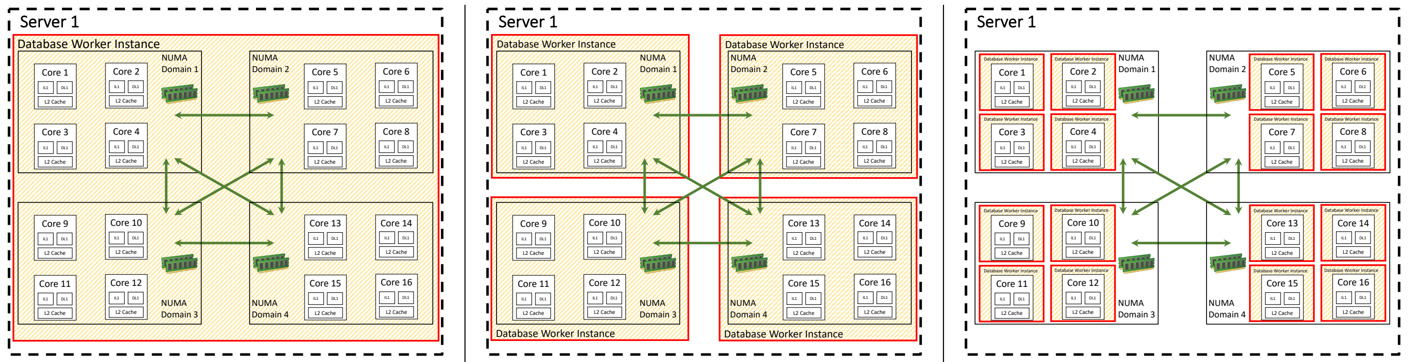
Finally, we observe that SingleStore is the only system that does not incur remote memory accesses. Each split in SingleStore is bound to the NUMA domain that also hosts the tasks that process it. Furthermore, SingleStore evenly places the data among the 4 NUMA domains and does not allow tasks to process remote splits.

III. Design Space of Distributed Query Engines

Current distributed query execution engines have different design configurations for their deployment within a single multiprocessor machine. In this section, we describe the main characteristics of the most widely adopted designs, highlighting strengths and weaknesses.

A. Design space overview

The design space can be divided into three categories: (1) Single-Worker Instance per Machine (WIM), (2) Single-



(a) WIM: Single Database Worker Instance per Machine

(b) WIN: Single Database Worker Instance per NUMA Domain

(c) WIC: Single Database Worker Instance per CPU Core

Fig. 4: Different design configurations for distributed query engines

Worker Instance per NUMA Domain (WIN), and (3) Single-Worker Instance per CPU Core (WIC). The main differences between these approaches concern the number of database worker instances, the way computing resources are allocated to each worker instance and data allocation policies. Different designs affect the degree of parallelism, amount of data to be exchanged and efficient use of NUMA.

(1) WIM: Single-Worker Instance per Machine. The Single-Worker Instance per Machine design is the most widely adopted by existing distributed query engines. The deployment of this design is not dependent on the topology of the underlying hardware and involves a single database worker instance that can manage all resources, as shown in Figure 4a. In addition, there are no restrictions on how memory is accessed. All threads can access both local and remote memories. This type of design is adopted by systems such as Presto or SparkSQL.

(2) WIN: Single-Worker Instance per NUMA Domain. In the Single-Worker Instance per NUMA Domain design, there are as many database worker instances as NUMA domains. The worker instances are placed by considering the physical layout of the multiprocessor machine and can only use the computational resources of the NUMA nodes that they are associated with. Communication between instances takes place through the network stack. The OS can migrate threads but only within the NUMA domain to which they are bound. In terms of data allocation, the WIN design relies on the Membind policy. The Membind policy allows each instance to be forced to use only its local memory. SingleStore suggests adopting this design when deploying its database in multiprocessor systems [29].

(3) WIC: Single-Worker Instance per CPU Core. Figure 4c shows the Single-Worker Instance per CPU Core design. It involves using one database worker instance per CPU core and is adopted by systems such as Greenplum [30], [31], H-Store [32] and HyPer [33]. The deployment varies according to the number of effective CPUs or cores. A rule of thumb is to use as many database workers as CPU cores. As in the WIN design, all instances exchange data using the network stack. In addition, there are no constraints on data placement strategies, but each instance prioritises memory local to the core on which it runs.

B. Trade-offs

One objective of this study is to explore the trade-offs between different deployment designs in multiprocessor systems. The main requirements that need to be balanced to achieve an efficient system are: data locality, efficient communication, and low resource contention. Each of these affects the others.

The Single-Worker Instance per Machine design is the only one among those discussed whose implementation is not conditioned by the physical system architecture. Moreover, with just a single database worker instance for each node in the cluster, communication between nodes can be minimised. However, maximising the data and threads locality is challenging. Inadequate allocation of data between NUMA domains risks congesting interconnects, leading to severe performance reductions. Redesigning the database engine to make it more NUMA-aware requires cost and effort, and, in many cases, this is an unviable option. To use all hardware on a single instance, there is always a NUMA cost somewhere.

In contrast, the Single-Worker Instance per NUMA Domain design can guarantee high data locality. Each instance is forced to use only the memory of the NUMA node to which it is bound, eliminating remote memory access and interconnect congestion. In addition, the work is partitioned by the database engine, thus generating a fair use of the resources shared in NUMA nodes. The downside of the WIN design is that the intra-node communication takes place via the network stack with less bandwidth than what can be achieved through interconnects. In addition, a larger number of database worker instances increases the inter-node communication and may vary the degree of parallelism generated by the database engines.

Similarly, the Single-Worker Instance per CPU Core design increases the amount of communication required for data processing but at the same time can improve data and thread placement. It allows explicit control over the contention within each database worker instance. As a result, WIC-based systems exhibit high single-thread performance and low contention. Running a single thread per worker instance allows them to disable locking and latching. However, ignoring thread safety limits growth and re-usability.

In the next section, we evaluate the performance of the above designs by varying their characteristics, e.g., the number of database worker instances, their data allocation policy, the degree of parallelism, etc., workflows, and hardware topological properties.

IV. Experimental Design Evaluation

We carry out an extensive empirical study with 4 popular distributed query engines running analytical queries. We analyze the effects that different system design decisions and configurations have on the overall performance and resource usage. Our analysis focuses on Presto [3], SparkSQL [4], SingleStore [2] and Greenplum [30]. Interestingly, all these systems use a different default deployment policy (e.g., different numbers of database worker instances, different memory and thread placements, etc.) when running on a multi-processor system. This makes them an interesting set of systems for exploring which design and configuration uses hardware resources most efficiently. We want to answer the following questions:

1. How many worker instances should run on a machine?
2. Which data- and thread- placement policies should be adopted by each worker instance?
3. Does the resulting deployment policy affect the engine's multi-core scalability?
4. Is the performance of the deployment policies affected by the underlying hardware?
5. Can we expect a similar behaviour of the policies when running a skewed workload?

After performing experiments on a single machine, we proceed with experiments in a distributed setting and explore:

1. How do the deployment policies perform in a scale-out setting with multiple multi-processor machines?
2. How much does the hardware setup play a role (e.g., the number of NICs per compute node, or the NICs' supported bandwidth)?

A. Setup and methodology

In the first half of our evaluation, we use an Intel Xeon CPU E5-4660 v4 with 4 NUMA domains, with 16 physical CPU cores each and 128 GB of RAM. The system runs Ubuntu 16.04.7 LTS, kernel version 4.4.0-201-generic, with the automatic NUMA balancing capability enabled.

To evaluate different deployment settings, we use the *libnuma* library [36] when starting a worker instances for thread binding and data placement. Our analysis focuses on the TPC-H [34] and JCC-H [35] benchmarks with scale factor 100 (approx. 100 GB of data). Every data point presented is an average of 5 runs.

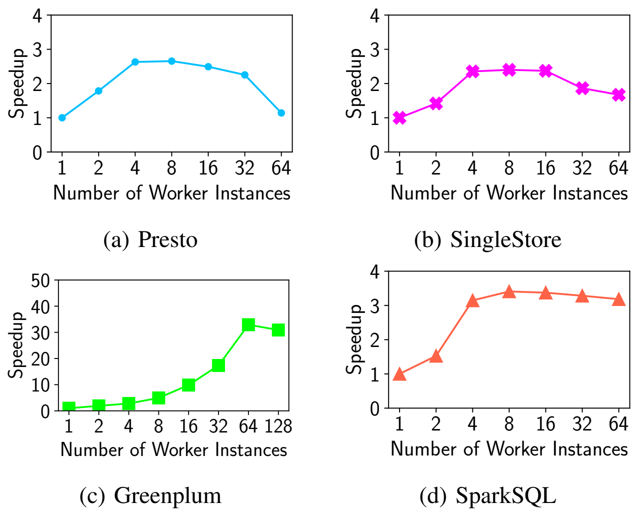


Fig. 5: Varying the number of database worker instances

B. Number of worker instances

To find the optimal number of database worker instances per machine, we run the full TPC-H benchmark on all systems, and vary the number of instances N. The resources assigned per database worker instance (e.g., the number of CPU cores), depend on the total number of cores in the system C, i.e., each worker instance is assigned C/N cores, and, where possible, all cores are from a single NUMA domain.

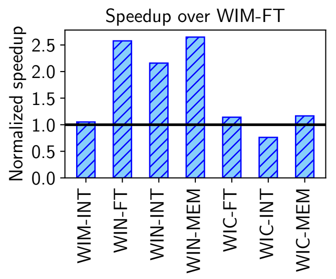
Figure 5 shows the geometric mean of each TPC-H query speed-up over the default deployment policy that runs a single worker instance per machine. We observe that Presto, SingleStore, and SparkSQL all have a similar behaviour and reach a peak performance when using 4 worker instances (2.53×, 2.35×, 3.2×, respectively). The speed-up plateaus for higher number of instances and deteriorates when going beyond 32.

In contrast, Greenplum shows linear growth until 64. Its engine is based on Postgres 9.4, which does not support parallel processing. Deploying more database worker instances better exploits computational resources. We note that Greenplum's performance drops when using 128 worker instances, meaning that it does not benefit from hyper-threading.

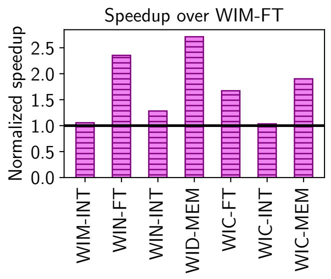
In conclusion, all systems benefit from running more worker instances per machine. For systems that support parallel query execution, the peak can be reached by running as many instances as there are NUMA nodes on the machine. Unsurprisingly, for systems that only support single-threaded query execution, running as many worker instances as there are physical cores on the machine gives the best results.

C. Data placement

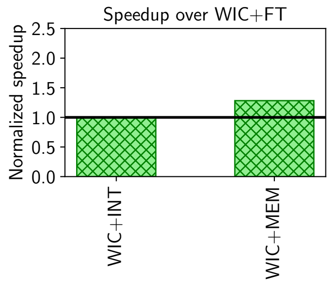
As we briefly discussed in Section II, choosing a suitable data- and thread-placement policy plays a critical role in how efficiently a system uses the underlying hardware resources. Hence, we continue our analysis by fixing the number of database worker instances to the standard Single-Worker In-



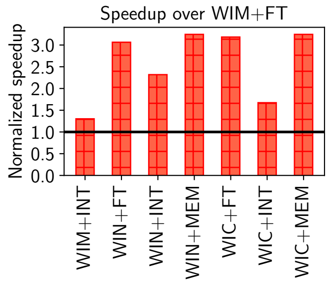
(a) Presto



(b) SingleStore



(c) Greenplum



(d) SparkSQL

Fig. 6: Effects of data placement strategies

stance per Machine or Single-Worker Instance per NUMA Domain, and evaluate what happens when altering the default data placement policy from First Touch (FT) to Interleaving (INT) or Membind (MEM).

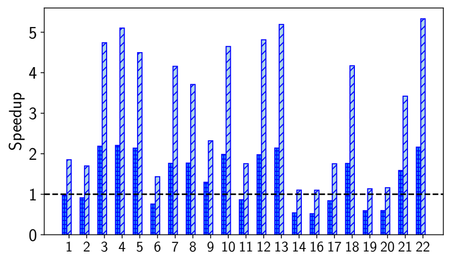
Figure 6 shows the results. We use WIM+FT policy as the baseline, and the speed-up is the geometric mean of the speed-ups obtained for all TPC-H queries. Presto, SparkSQL and SingleStore exhibit the best performance when running a worker instance per NUMA domain in combination with the Membind policy. Specifically, they each achieve speed-ups of 2.61×, 3.33× and 2.65×, respectively, over to the baseline (WIM+FT), and a smaller improvement over WIN+FT.

Unsurprisingly, interleaving memory shows the worst performance, because it disregards the locality requirements even if balancing load well across NUMA domains. Due to its single-threaded execution engine for Greenplum, we only evaluate the WIC design. Again, using the Membind policy shows a 1.25× performance improvement over the FT policy, whereas the INT option has almost no effect.

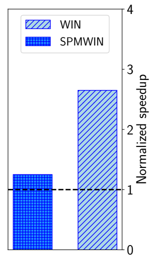
Finally, SparkSQL is the engine with the biggest gain from the default deployment strategy, achieving a speed-up of 3×. Here, we refer back to the fact that SparkSQL is the engine that suffers from a higher percentage of interconnect congestion in the default setting. The WIN+MEM deployment policy overcomes these issue by avoiding resource sharing among the instances and enforcing a local computation over local data.

D. Thread placement

Until now, the core assignment to the worker instances has followed a policy that bundles them in proximity to one another, ideally in the same NUMA domain. An alternative approach is to spread them out across the machine, i.e., assign



(a) TPC-H query speed-up



(b) Geo-mean

Fig. 7: Effect of thread-placement strategies

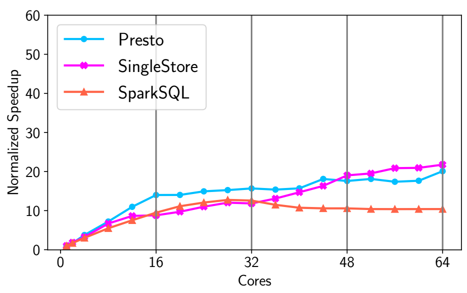
one CPU core from a different NUMA node to a worker instance. We call this variant *SPread Multiple database Worker Instances* or SPMWIN. To be clearer, the WIN and SPMWIN policies use the same number of worker instances, and assign the same number of cores to each instance. As previously, we compare these two with the default configuration of Presto (WIM+FT) and evaluate their performance with the TPC-H benchmark.

Figure 7 shows the results. As expected, locality plays a key role and the WIN design shows superior performance compared to SPMWIN. In fact for WIN, the speed-up for individual queries is between 1.21× and 5.23× with a geometric mean of 2.65×. In contrast, the mean speed-up for SPMWIN is just 1.25×, with almost half of the TPC-H queries experiencing a slow-down compared to the default deployment configuration.

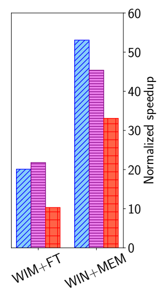
E. Multi-core scalability

Previously, we have shown that the deployment with a Single-Worker Instance per NUMA Domain produces better performance than other deployment strategies. The next step is to evaluate how this deployment scales as the number of cores increases. In this experiment, we compare the multi-core scalability of the WIN+MEM deployment against the more classical WIM+FT deployment.

Figure 8a shows the scalability of Presto, SingleStore and SparkSQL when using the WIM+FT deployment policy. For comparison, in Figure 8b, we show the maximum speed-up of

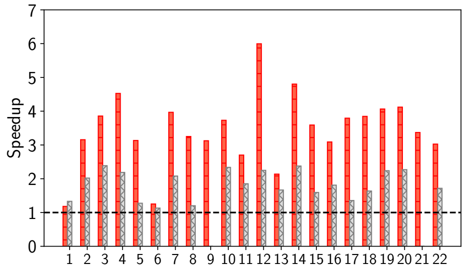


(a) WIM+FT scalability

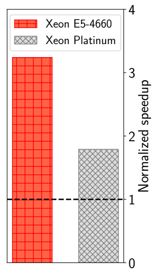


(b) 64 cores

Fig. 8: Multi-core scalability

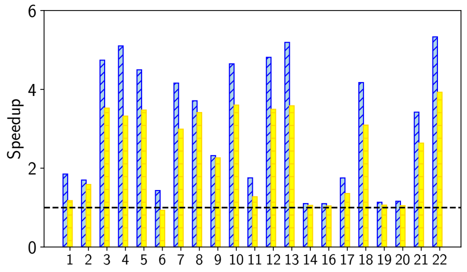


(a) TPC-H query speedup

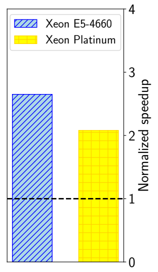


(b) Geo-mean

Fig. 9: SparkSQL: Different hardware platforms



(a) TPC-H query speedup



(b) Geo-mean

Fig. 10: Presto: Different hardware platform

the WIM+FT deployment policy alongside the performance achieved when applying the WIN+MEM deployment.

All systems have a similar performance with lower core counts, and scale well within a single NUMA node. However, as the number of cores increases, the performance of all systems is affected by the NUMA effects and their scalability decreases. With 64 cores, the speed-up for Presto and SingleStore is around 20× over a single core; for SparkSQL, it is only around 10×. In contrast, when applying the WIN+MEM design with 64 cores, Presto achieves a speed-up of 53× over the single core execution. SingleStore and SparkSQL also exhibit a good scalability of around 46× and 33×, respectively.

In conclusion, the main problem with the Single-Worker Instance per Machine deployment policy is that it fails to exploit the abundant parallelism of multiprocessor systems. By using a single database worker instance, there will always be a NUMA cost somewhere. These costs inevitably limit the multicore scalability. By contrast, isolating database worker instances within NUMA nodes avoids remote memory accesses and does not congest interconnects, resulting in better scalability.

F. Different hardware platforms

Our hypothesis is that the properties of the underlying hardware are also a key factor in the observed behaviour and bear the potential for performance improvements. For example, often when profiling the engines under test, the congestion in the interconnect is one of the main bottlenecks.

Therefore, we evaluate if we can also get significant speed-ups on a newer generation Intel machine (Xeon Platinum). It replaces the standard QuickPath Interconnect (QPI) with the Ultra Path Interconnect (UPI), allowing transfer speeds of up to 10.4 GT/s. For this experiment, we run the TPC-H benchmark using SparkSQL and Presto, applying the WIN design, and compare performance over the standard WIM design. We used an Intel Xeon Platinum CPU 8275CL with 96 physical cores, 2 NUMA domains and 192 GB of RAM. The system runs Ubuntu Pro 18.04 LTS with the automatic NUMA balancing capability enabled.

Figure 9 shows the results for SparkSQL. As anticipated, the performance gains from using the WINdeployment are smaller than before, but are still non-negligible. We point out that SparkSQL previously showed the highest level of interconnect congestion among the systems tested (see Section II-C). The UPI technology mitigates this weakness by diminishing the advantages obtained with the WIN design. Nevertheless, using the Intel Xeon Platinum 2nd Gen processor, the Single-Worker Instance per NUMA Domain design shows an average speed-up of 1.78× compared to the default WIM design. Moreover, all queries show a speed-up that varies in the range of 1.13× and 2.39×.

Figure 10 shows the results when running the same experiment using Presto. We observe a speed-up of the WIN design over the WIM ranging from 1.18× to 3.93× for the individual queries, with a geo-mean of 2.1×.

Query number 6 is the only one that shows a performance degradation with a speed-up of 0.93×. It is one of the shortest running queries, and does not have the Orders table as a base table, thus avoiding the costly remote accesses when evaluating the predicate. Hence, eliminating the remote accesses on the newer generation Intel machine does not outweigh the communication overhead of running multiple database worker instances.

G. Skewed workloads

Next, we evaluate if the WIN+MEM's deployment policy is also preferable to the others when running skewed workloads. We repeat the experiments with the JCC-H benchmark [35], which introduces join-crossing-correlations (JCC) and skew within the TPC-H dataset. It uses the same queries with "skewed" parameters to experience strong join-crossing-correlations and distortion in filter, aggregation and join operations. The objective of this experiment is to assess the efficiency or need for additional balancing to the data allocation [35]. We conduct the experiments with SingleStore and SparkSQL using the first generation Intel Xeon processor.

Figure 11 shows the results for both the individual queries and a geo-mean for the whole benchmark suite for both systems. The results indicate that the WIN+MEM design does not affect the skew data management strategies of the systems. More specifically, one can also observe performance improvements over the default WIMdeployment, with an observed speed-up range from 1.06× to 8.19× for SingleStore and from 1.05× to 3.65× for SparkSQL, respectively. The geo-mean speed-up for SingleStore and SparkSQL is 2.17× and 2.25×, respectively.

V. Scale-out Experiments

The results in the previous sections have consistently shown that the Single-Worker Instance per NUMA Domain deployment policy achieves more favorable performance over the WIM+FT policy for all the systems under test in a single-machine setting. We now proceed with an exploration if the same observation holds in a scale-out setting.

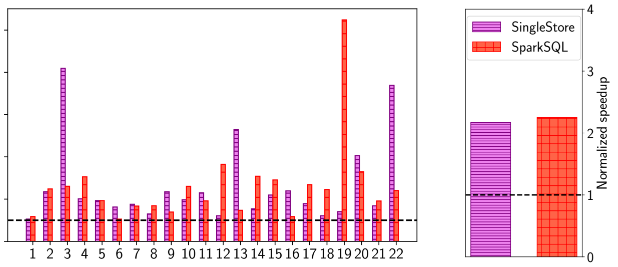
In the next experiment, we run the TPC-H benchmark on an on-premise cluster, consisting of 8 multiprocessor machines. Each machine has a dual-socket Intel Xeon CPU E5-2630 v4 processor, with 40 CPU cores, 32 GB of RAM and 2 NUMA domains. It also contains a 1 Gbps NIC and a 10 Gbps NIC. Due to lower memory capacity of our cluster, we use a smaller scale factor than in the previous experiments, which we increase proportionally as the number of machines increases. We use a scale factor of 15 per machine during the experiments with Presto and a scale factor of 6 per machine for SingleStore.

A. Multiple machines

We compare the performance of both systems when using the WIN+MEM deployment policy over the default WIM+FT. In the baseline WIM+FT configuration, we use a 1 Gbps NIC. For the WIN+MEM design, we explore the impact when both database worker instances on the machine share the NIC. In configuration (1), they share the 1 Gbps network card; in option (2), they share the 10 Gbps NIC.

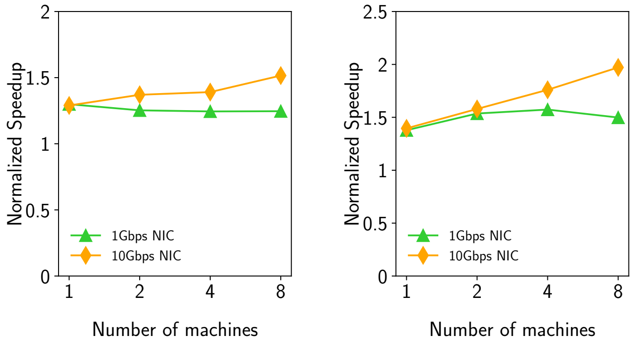
Our hypothesis is that we will observe linear horizontal scalability as we increase the number of machines, i.e., the speed-up obtained when running multiple instances on a single machine is maintained when using a cluster (our dataset size increases linearly as we add more machines) until a certain point. Scaling to more machines, and hence more worker instances, will eventually be dependent on the communication fan-out for each instance and the available resources. Furthermore, we expect the performance to be better when using the 10 Gbps NIC, as it enables less congested communication between the instances.

Figure 12 shows the geometric mean speed-up of the whole TPC-H workload over the baseline deployment. As expected, the WIN+MEM retains the speed-up when we increase the number of machines for both systems. With



(a) JCC-H query speedup (b) Geo-mean

Fig. 11: Impact of skewed workloads



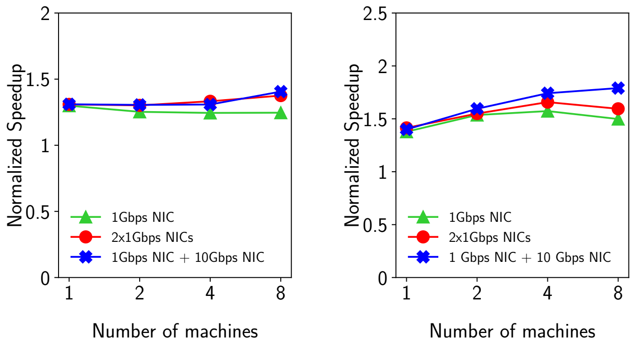
(a) Presto (b) SingleStore

Fig. 12: Speedup over WIM+FT with 1 Gbps NIC (TPC-H)

Presto, the WIN+MEM deployment using the 1 Gbps NIC shows a slight decrease in speed-up as the number of machines increases. From 1.3× with 1 machine, it decreases to 1.25× with 8 machines. By increasing the number of machines, we also increase the amount of data exchanged between the instances, making the 1 Gbps network a bottleneck. This is confirmed when using the 10 Gbps NIC as an alternative, which enables an increase in the speed-up with a peak of 1.52× when using all 8 machines. While SingleStore exhibits a similar behaviour to Presto in the 1 Gbps deployment, it shows a constant increase in speed-up when scaling-out to 8 machines with the 10 Gbps NIC, thereby confirming our second hypothesis that a higher network bandwidth leads to a better performance.

B. Multiple network interface cards

In the previous section, we have seen that the network bandwidth can become a bottleneck when using the Single-Worker Instance per NUMA Domain design and that using a higher bandwidth NIC can help improve performance. However, there is another attractive aspect that we can explore using the WINdeployment: making use of potentially multiple NICs in the machine, by associating different cards with different worker instances running on the machine.



(a) Presto (b) SingleStore

Fig. 13: Scalability when using 2 NICs per machine

We repeat the previous experiment by evaluating two more alternatives: (1) in each node of the cluster, we assign a 1 Gbps NIC to each of the database worker instances, i.e., we limit the 10 Gbps NIC to 1 Gbps; (2) we assign a 1 Gbps NIC to the first worker instance, and the other 10 Gbps NIC to the second. As before, we compare the performance of both systems when deployed with these two alternatives to the standard configuration of using one 1 Gbps NIC shared among both worker instances.

Figure 13 shows the geometric mean of each TPC-H query speed-up over the WIM+FT design. In Presto, both alternatives show the same results. The speed-up is only marginally higher than in the experiment with only one NIC and reaches up to 1.40× with 8 machines. Using multiple NICs is better than having both instances sharing one, but there is no clear benefit when using a higher bandwidth NIC. This suggests that Presto's query engine is not bottlenecked by transferring data over the network. However, the experiment with SingleStore clearly shows that there is a difference in speed-up between the two alternatives, with the 1 + 10 Gbps alternatives showing better scalability than the 2 × 1 Gbps one.

In conclusion, by using multiple NICs, the WIN+MEM design can benefit even more, but the impact of the gain is not only influenced by the underlying hardware. By showing a different behaviour, Presto and SingleStore, suggest that the efficiency of the query engine itself and the distributed query plan execution can also play a role in how much we can benefit from the deployment configuration.

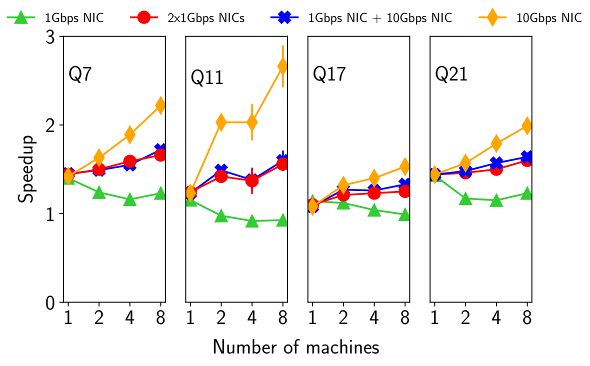


Fig. 14: Scalability of TPC-H queries with Presto

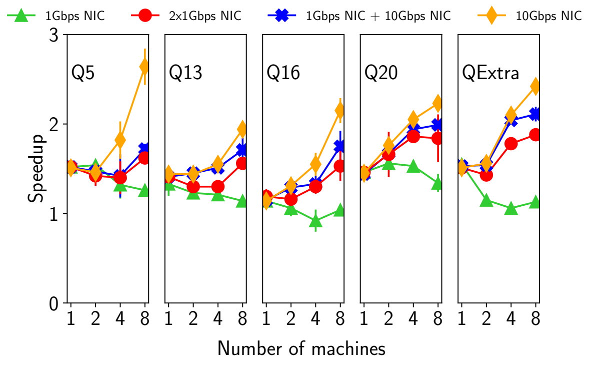


Fig. 15: Scalability of TPC-H queries with SingleStore

C. Network-intensive queries

Next, we focus our analysis on selected TPC-H queries. More specifically, we investigate the ones that are more network-intensive when performing distributed joins.

In Figure 14, we show the speed-up for Presto when using the WIN+MEM deployment policy over WIM+FT for Q7, Q11, Q17 and Q21. As before, with a 1 Gbps NIC per machine, these queries are easily saturating the network bandwidth. Consequently, by increasing the number of machines, we also increase network pressure and, unsurprisingly, the performance drops to the point of losing the benefits of the WIN+MEM design.

As expected, using multiple or a more powerful NIC improves speed-up. With a 10 Gbps NIC, the speed-up increases with the number of machines up to 2.2×, 2.7×, 1.53× and 2× with 8 machines. Interestingly, there is not much difference between the two variants using two NICs. For both versions, their speed-up increases to around 1.7×, 1.6×, 1.3× and 1.6× with 8 machines.

Figure 15 shows the same experiment with SingleStore. In this case, the TPC-H queries with the most network traffic during data processing are Q5, Q13, Q16 and Q20. In addition, we evaluate the following extra query that performs a distributed join using non-primary keys *select count(*) from lineitem, partsupp where l_partkey = ps_partkey and l_suppkey = ps_suppkey.*

The experiment with a 10 Gbps NIC for both database worker instances again shows the best results. The speed-up increases as the number of machines increases to 2.64×, 1.94×, 2.15×, 2.23× and 2.42×, respectively, with 8 machines. On the other hand, as with Presto, the experiment with the 1 Gbps NIC shows a decrease in speed-up as the number of machines increases. With 8 machines, the speed-up decreases to 1.26×, 1.14×, 1.04×, 1.34× and 1.13×.

In contrast, experiments with multiple NICs show an increasing speedup. Unlike Presto, the two configurations evaluated do not always show the same performance. The experiment with one 1 Gbps NIC and one 10 Gbps NIC shows slightly better results than the one with two 1 Gbps NICs for Queries 13, 16, 20 and our extra query. With 8 machines, its speed-up reaches 1.72×, 1.71×, 1.75×, 1.99× and 2.11×, respectively; the experiment with two 1 Gbps NICs reaches 1.62×, 1.56×, 1.53×, 1.84× and 1.88×, respectively.

We also perform an additional synthetic join benchmark. The realized synthetic dataset consists of two tables with 4-byte keys and payloads. The two tables have a cardinality of 65 million and 125 million, respectively, when run with a single machine. Their size increases proportionally as the number of machines increases.

Figure 16 shows the speed-up of the WIN+MEM design over the WIM+FT one as the number of machines changes. As in the previous cases, the configuration with two NICs yields a higher speed-up than the configuration with a single 1 Gbps

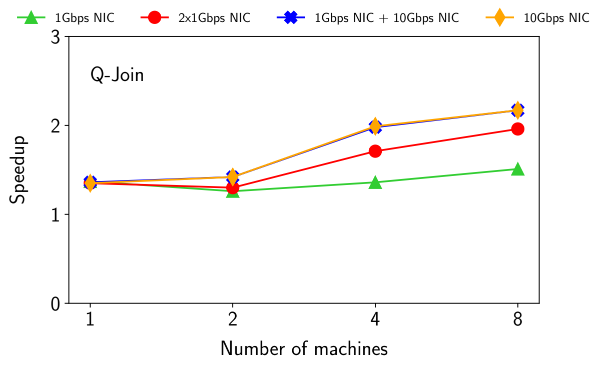
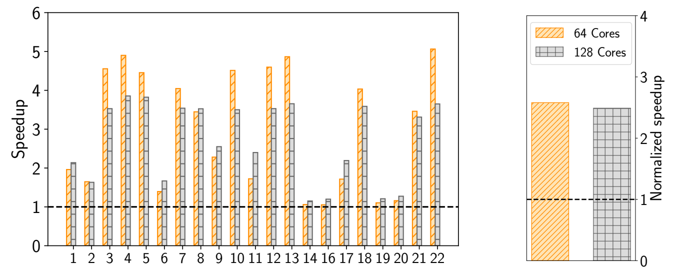


Fig. 16: Join Performance with SingleStore



(a) TPC-H query speed-up

(b) Normalised speed-up

Fig. 17: WIN+MEM speed-up over WIM-FT with/without hyper-threading

NIC. Interestingly, the experiment with one 1 Gbps NIC and one 10 Gbps NIC achieves the same speed-up as the one with the single 10 Gbps NIC. With 8 machines, the WIN design is 2.17× faster than the WIM design, while the same design but with a single 1 Gbps NIC is only about 1.3× faster. Finally, the experiment with two 1 Gbps NICs, after an initial slight decrease with 2 machines, increases the speed-up to 1.96× with 8 machines.

Insights. For queries that move large amounts of data, the network can become the bottleneck and reduce the benefits obtained from the WIN+MEM design. The use of multiple database worker instances makes it easy to use multiple NICs within a single machine, increasing bandwidth and reducing this bottleneck.

VI. Discussion

In this section, we discuss the impact of other system parameters and justify their configuration in our experiments.

Hyper-threading. The goal of hyper-threading is to hide high memory accesses latencies. We evaluate the impact of using hyperthreading for analytical workflows. It is clear from the previous experiments that the WIN+MEM design is the one that provides the largest performance benefits. Therefore, for this experiment, we use Presto to evaluate the WIN+MEM design over the WIM+FT with and without hyper-threading.

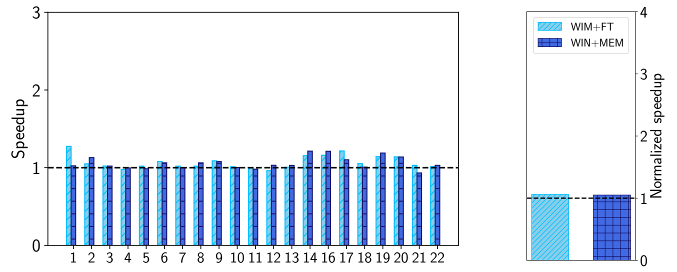
Figure 17 shows the results. In general, the speed-up achieved by the WIN+MEM design over the WIM+FT design is about 2.5× in both cases. On a query basis, half of the TPC-H queries show a higher speed-up when executed with hyper-threading disabled; the other half shows a slight improvement in speed-up with hyper-threading.

From the results, we can conclude that hyper-threading mitigates the benefits of the WIM+MEM design for the queries most affected by the negative effects of NUMA. This is due to a performance increase in the WIM+FT design when enabling hyper-threads. On the other hand, the rest of the queries show more benefits in using the WIN+MEM design. Thus, the use of hyper-threading does not have an impact on the overall performance benefits achieved by the WIN+MEM design compared to WIM+FT.

Based on the previous results, we decide to only use the physical cores. Enabling hyper-threading raises additional questions related to the contention on the computing resources inside a CPU core. The caches shared by the hyper-threads generate high contention, and data is loaded from main memory more frequently. Our study evaluates the computing resource utilisation at a per-core level and focuses on the correct data placement at the DRAM level.

Automatic NUMA balancing. We evaluate the impact of automatic NUMA balancing during data processing. We use Presto in two different design configurations (WIM and WIN) and evaluate the response times when enabling and disabling balancing. During the experiment, we use the First-Touch data allocation policy.

Figure 18 shows Presto's speed-up achieved by disabling automatic NUMA balancing for both designs. It reveals that both designs do not achieve a significant improvement in response time: both speed-ups are close to one. Interestingly, Figure 18a shows that several queries improve performance by disabling automatic balancing, albeit slightly. Since the results show that using or not using this capability does not have a significant impact, we therefore decide not to change the OS configuration by keeping it enabled.



(a) TPC-H query speed-up

(b) Normalised speed-up

Fig. 18: Speedup by disabling automatic NUMA balancing for WIM+FT and WIN+MEM

VII. Related work

A. NUMA-awareness in query processing

Some studies have evaluated the performance of distributed databases when running on modern multiprocessor machines [25], [26], [37]. Porobic et al. [37] focus on determining the right trade-off between shared-everything and shared-nothing deployments in multiprocessor systems measuring the impact of distributed transactions and skewed requests on different OLTP workloads. Salomie et al. [26] propose to partition a multicore machine and replicate the existing databases as if the machine were a distributed system. These projects focus highly on transactional workflows and do not explicitly address the NUMA-awareness.

A great deal of effort has been put into improving NUMA-awareness at the OS level. Operating systems such as Mach [81], exokernel [80] and Barrelfish [52] implement message passing to facilitate the development of NUMA-aware systems, because communication between threads is done explicitly through messages in a NUMA-aware way. In addition to not being specific to database systems, all of these proposals would require substantial changes to the database engine.

B. Black-box approaches

Several papers propose black-box approaches to improve NUMA awareness. Dashti et al. propose an algorithm that work at the OS level to define a placement of threads and data that minimizes the level of congestion on the interconnect between processors [27]. Their algorithm, called Carrefour, uses global traffic congestion observations and obtains up to 3 times higher performance, but it is highly dependent on the presence of hardware performance monitoring. In a cloud environment though, containers do not expose performance counters, making this solution challenging to apply.

Calciu et al. have developed a black-box method for defining competing NUMA-aware data structures [19]. Despite the high level of generality of the proposed solution, NUMA-aware tailor-made data structures are able to achieve better performance and scalability. Another transparent approach is the Linux's automatic NUMA balancing, which monitors performance metrics to move threads closer to the memory they need to access. In our study, we show that this functionality leads to sub-optimal results.

A more effective approach is presented by Giceva et al. [41]. The authors propose a deployment algorithm to multicore systems based on the behaviours of individual database operators and dataflow information.

C. Data and thread placement

Memory allocation and data partitioning or replication strategies must be implemented to benefit from the distributed nature of the NUMA architecture. Some research helps by showing advantages and disadvantages in choosing strategies for data placement in the main memory. The work done by Yang et al. [42] highlights how a shared resource distribution policy can benefit the performance of multiprocessor platforms. In our study, we evaluate this policy and show that shared use of resources does not lead to optimal performance. Psaroudakis et al. [43] test different data distribution strategies with SAP HANA and state that the physical partitioning strategy to the available NUMA domains achieves the best performance.

There are a few database systems that have chosen to consider NUMA awareness as one of the design principles in implementing their system. Kissinger et al. [44] build an in-memory database engine (ERIS) that can reduce NUMA-related problems by applying partitioning approaches based on platform topology. ERIS executes tera-scale analytical workloads entirely in main memory using an adaptive partitioning approach that exploits the topology of the underlying NUMA platform and significantly reduces NUMA-related issues. BatchDB by Makreshanski et al. [45] is another database system designed to minimize the resource interference between analytical and transactional engines on NUMA systems. One of the main aspects of this work is that the implicit isolation of hardware resources has a major impact on performance by avoiding the high synchronization overheads over the interconnect.

D. Cloud-containers resource allocation

Container resource allocation is a key factor for cloud providers. Several studies propose resource management techniques to minimise energy consumption [46]–[48]. Other studies propose resource allocation models based on priority [49], [83]. These approaches focus more on load-balancing problems for specific cloud providers. None of these studies target maximising the use of hardware resources in modern multiprocessor systems. Jebalia et al. [82] propose a resource allocation approach based on neural networks. The study exploits a genetic algorithm to maximise the use of resources, but it is mainly efficient in cases without resource contention.

VIII. Conclusion

We investigated the performance impact of different deployment policies of distributed query engines over multicore NUMA machines. We showed that deploying a database worker instance per NUMA domain with a corresponding memory binding policy guarantees increased performance compared to other designs, a high multicore scalability and good horizontal scalability. Our proposed guidelines directly apply to on-premise distributed query engines, enabling them to maximize their resource utilisation without requiring significant changes to the engine. In the case of engines deployed in the cloud, our insights may enable better future provisioning and assignment of hardware resources. Overall, we conclude that treating NUMA machines as distributed systems is a an effective way to get the most out of the underlying hardware resources.

References

[1] Binnig, C., Crotty, A., Galakatos, A., Kraska, T., & Zamanian, E. (2016). "The End of Slow Networks: It's Time for a Redesign." ArXiv, abs/1504.01048.

[2] Chen, J., Jindel, S., Walzer, R., Sen, R., Jimsheleishvilli, N., & Andrews, M. (2016). "The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database." Proc. VLDB Endow., 9, 1401-1412. https://doi.org/10.14778/3007263.3007277

[3] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. "Presto: SQL on Everything." In 2019 IEEE 35th International Conference on Data Engineering (ICDE). 1802–1813.

[4] Armbrust, M., Xin, R., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., & Zaharia, M.A. (2015). "Spark SQL: Relational Data Processing in Spark." Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. https://doi.org/10.1145/2723372.2742797

[5] Armenatzoglou, N., Basu, S., Bhanoori, N., Cai, M., Chainani, N., Chinta, K., Govindaraju, V., Green, T.J., Gupta, M., Hillig, S., Hotinger, E., Leshinksy, Y., Liang, J., McCreedy, M., Nagel, F., Pandis, I., Parchas, P., Pathak, R., Polychroniou, O., Rahman, F., Saxena, G., Soundararajan, G., Subramanian, S., & Terry, D. (2022). "Amazon Redshift Re-invented." Proceedings of the 2022 International Conference on Management of Data. https://doi.org/10.1145/3514221.3526045

[6] Pandis, I. (2021). "The evolution of Amazon Redshift." Proc. VLDB Endow., 14, 3162-3163. https://doi.org/10.14778/3476311.3476391

[7] Amazon Web Services, Inc. "Amazon Redshift clusters." 2022. https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-clusters.html

[8] Gupta, A., Agarwal, D.K., Tan, D., Kulesza, J., Pathak, R., Stefani, S., & Srinivasan, V. (2015). "Amazon Redshift and the Case for Simpler Data Warehouses." Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. https://doi.org/10.1145/2723372.2742795

[9] Aguilar-Saborit, J., & Ramakrishnan, R. (2020). "POLARIS: The Distributed SQL Engine in Azure Synapse." Proc. VLDB Endow., 13, 3204-3216. https://doi.org/10.14778/3415478.3415545

[10] Fernandes, S., & Bernardino, J. (2015). "What is BigQuery?" In Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS '15). Association for Computing Machinery, New York, NY, USA, 202–203. https://doi.org/10.1145/2790755.2790797

[11] Amazon Web Services, Inc. "Amazon Athena." 2022. https://docs.aws.amazon.com/whitepapers/latest/big-data-analytics-options/amazon-athena.html

[12] Oracle. "Oracle Exadata database machine X8-2." 2017. https://www.oracle.com/technetwork/database/exadata/exadata-x8-2-ds-5444350.pdf

[13] Teradata. "Teradata Vantage™ - SQL Fundamentals." 2022. https://docs.teradata.com/r/Teradata-VantageTM-SQL-Fundamentals/June-2022/Introduction-to-SQL-Fundamentals

[14] Apache Software Foundation, 2010. "Hadoop", Available at: https://hadoop.apache.org.

[15] The kernel development community. " Documentation for /proc/sys/kernel/." 2022. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#numa-balancing

[16] The kernel development community. " Numa policy hit/miss statistics." 2022. https://www.kernel.org/doc/html/latest/admin-guide/numastat.html

[17] Lepers, B., Quéma, V., & Fedorova, A. (2015). "Thread and Memory Placement on NUMA Systems: Asymmetry Matters." USENIX Annual Technical Conference.

[18] Pusukuri, K.K., Gupta, R., & Bhuyan, L.N. (2012). "Thread Tranquilizer: Dynamically reducing performance variation." ACM Trans. Archit. Code Optim., 8, 46:1-46:21. https://doi.org/10.1145/2086696.2086725

[19] Calciu, I., Sen, S., Balakrishnan, M., & Aguilera, M.K. (2017). "Black-box Concurrent Data Structures for NUMA Architectures." Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. https://doi.org/10.1145/3037697.3037721

[20] Barthels, C., Alonso, G., Hoefler, T., Schneider, T., & Müller, I. (2017). "Distributed Join Algorithms on Thousands of Cores." Proc. VLDB Endow., 10, 517-528. https://doi.org/10.14778/3055540.3055545

[21] Daly, H., Hassan, A., Spear, M.F., & Palmieri, R. "NUMASK: High Performance Scalable Skip List for NUMA". DISC (2018).

[22] Lang, H., Leis, V., Albutiu, MC., Neumann, T., Kemper, A. "Massively Parallel NUMA-Aware Hash Joins." In: Jagatheesan, A., Levandoski, J., Neumann, T., Pavlo, A. (eds) In Memory Data Management and Analysis. IMDM IMDM 2013 2014. Lecture Notes in Computer Science(), vol 8921. Springer, Cham. https://doi.org/10.1007/978-3-319-13960-9_1

[23] Albutiu, M., Kemper, A., & Neumann, T. (2012). "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems." Proc. VLDB Endow., 5, 1064-1075.

[24] Li, Y., Pandis, I., Müller, R., Raman, V., & Lohman, G.M. "NUMA-aware algorithms: the case of data shuffling." CIDR (2013).

[25] Kiefer, T., Schlegel, B., & Lehner, W. "Experimental Evaluation of NUMA Effects on Database Management Systems." BTW (2013).

[26] Salomie, T., Subasu, I.E., Giceva, J., & Alonso, G. (2011). "Database engines on multicores, why parallelize when you can distribute?" EuroSys '11.

[27] Dashti, M., Fedorova, A., Funston, J.R., Gaud, F., Lachaize, R., Lepers, B., Quéma, V., & Roth, M. (2013). "Traffic management: a holistic approach to memory placement on NUMA systems." ASPLOS '13.

[28] Gaud, F., Lepers, B., Funston, J.R., Dashti, M., Fedorova, A., Quéma, V., Lachaize, R., & Roth, M. (2015). "Challenges of memory management on modern NUMA systems." Communications of the ACM, 58, 59 - 66.

[29] SingleStore, Inc. "SingleStore Documentation." 2022. https://docs.singlestore.com/v7.3/introduction/documentation-overview/

[30] Greenplum Database. "Introduction to Greenplum." 2022. https://docs.greenplum.org/6-10/install_guide/preinstall_concepts.html

[31] Greenplum. "About this Documentation." 2022. https://docs.greenplum.org/6-12/common/gpdb-features.html

[32] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S.B., Jones, E.P., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., & Abadi, D.J. "H-store: a high-performance, distributed main memory transaction processing system." Proc. VLDB Endow., 1, 1496-1499.

[33] Kemper, A., & Neumann, T. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." 2011 IEEE 27th International Conference on Data Engineering, 195-206.

[34] Transaction Processing Performance Council (TPC). "TPC Benchmark H, (Decision Support) Standard Specification Revision 2.18.0." http://www.tpc.org/

[35] Boncz, P.A., Anadiotis, A.G., & Kläbe, S. (2017). JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. TPCTC. https://ir.cwi.nl/pub/27429/27429.pdf

[36] Kleen, A. (2005). "A numa api for linux." Novel Inc.

[37] Porobic, D., Pandis, I., Branco, M., Tözün, P., & Ailamaki, A. (2012). "OLTP on Hardware Islands." Proc. VLDB Endow., 5, 1447-1458.

[38] Hwang, J., Ramakrishnan, K.K., & Wood, T. (2015). "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms." IEEE Transactions on Network and Service Management, 12, 34-47.

[39] Balkesen, C., Teubner, J., Alonso, G., & Özsu, M.T. (2013). "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware." 2013 IEEE 29th International Conference on Data Engineering (ICDE), 362-373.

[40] Teubner, J., & Müller, R. (2011). "How soccer players would do stream joins." SIGMOD '11.

[41] Giceva, J., Alonso, G., Roscoe, T., & Harris, T.L. (2014). "Deployment of Query Plans on Multicores." Proc. VLDB Endow., 8, 233-244.

[42] Yang, M., Huang, W., & Chen, J. (2019). "Resource-Oriented Partitioning for Multiprocessor Systems with Shared Resources." IEEE Transactions on Computers, 68, 882-898.

[43] Psaroudakis, I., Scheuer, T., May, N., Sellami, A., & Ailamaki, A. (2015). "Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement." Proc. VLDB Endow., 8, 1442-1453.

[44] Kissinger, T., Kiefer, T., Schlegel, B., Habich, D., Molka, D., & Lehner, W. (2014). "ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload." ADMS@VLDB.

[45] Makreshanski, D., Giceva, J., Barthels, C., & Alonso, G. (2017). "BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications." Proceedings of the 2017 ACM International Conference on Management of Data.

[46] Bermejo, B., Guerrero, C., Lera, I., & Juiz, C. (2016). "Cloud Resource Management to Improve Energy Efficiency Based on Local Nodes Optimizations." ANT/SEIT.

[47] Geronimo, G.A., Werner, J., Westphall, C.M., & Defenti, L. (2013). "Provisioning and Resource Allocation for Green Clouds."

[48] Srikantaiah, S., Kansal, A., & Zhao, F. (2008). “Energy aware consolidation for cloud computing.“ CLUSTER 2008.

[49] Pawar, C.S., & Wagh, R.B. (2012). “Priority based dynamic resource allocation in Cloud computing with modified waiting queue.” 2013 International Conference on Intelligent Systems and Signal Processing (ISSP), 311-316.

[50] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K.K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., & Bao, X. (2017). “Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases.” Proceedings of the 2017 ACM International Conference on Management of Data.

[51] Dageville, B., Cruanes, T., Zukowski, M., Antonov, V.N., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., Lee, A.W., Motivala, A., Munir, A., Pelley, S., Povinec, P., Rahn, G., Triantafyllis, S., & Unterbrunner, P. (2016). “The Snowflake Elastic Data Warehouse.” Proceedings of the 2016 International Conference on Management of Data.

[52] Baumann, A., Barham, P., Dagand, P., Harris, T.L., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., & Singhania, A. (2009). “The multikernel: a new OS architecture for scalable multicore systems.” SOSP ’09.

[53] Wentzlaff, D., & Agarwal, A. (2009). “Factored operating systems (fos): the case for a scalable operating system for multicores.” ACM SIGOPS Oper. Syst. Rev., 43, 76-85.

[54] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, M.F., Morris, R.T., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., & Zhang, Z. (2008). “Corey: An Operating System for Many Cores.” OSDI.

[55] Hashem, I.A., Yaqoob, I., Anuar, N.B., Mokhtar, S., Gani, A.B., & Khan, S.U. (2015). “The rise of ”big data” on cloud computing: Review and open research issues.” Inf. Syst., 47, 98-115.

[56] Mishra, V., Benjamin, J.L., & Zervas, G.S. (2021). “MONet: heterogeneous Memory over Optical Network for large-scale data center resource disaggregation.” IEEE/OSA Journal of Optical Communications and Networking, 13, 126-139.

[57] Liu, W., Cai, J., Chen, Q.C., & Wang, Y. (2021). “DRL-R: Deep reinforcement learning approach for intelligent routing in software-defined data-center networks.” J. Netw. Comput. Appl., 177, 102865.

[58] Roozbeh, A. (2019). “Toward Next-generation Data Centers : Principles of Software-Defined “Hardware” Infrastructures and Resource Disaggregation.”

[59] Mars, J., Tang, L., Hundt, R., Skadron, K., & Soffa, M.L. (2011). “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations.” 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 248-259.

[60] Wang, L., & Ranjan, R. (2015). “Processing Distributed Internet of Things Data in Clouds.” IEEE Cloud Computing, 2, 76-80.

[61] Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., & Grot, B. (2014). “Scale-out NUMA.” Proceedings of the 19th international conference on Architectural support for programming languages and operating systems.

[62] Grolinger, K., Higashino, W.A., Tiwari, A., & Capretz, M.A. (2013). “Data management in cloud environments: NoSQL and NewSQL data stores.” Journal of Cloud Computing: Advances, Systems and Applications, 2, 1-24.

[63] Wu, J., Wang, J., & Zaniolo, C. (2022). “Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines.” Proceedings of the 2022 International Conference on Management of Data.

[64] Agarwal, S., Milner, H., Kleiner, A., Talwalkar, A.S., Jordan, M.I., Madden, S., Mozafari, B., & Stoica, I. (2014). “Knowing when you’re wrong: building fast and reliable approximate query processing systems.” Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.

[65] Awada, U., & Barker, A. (2017). “Improving Resource Efficiency of Container-Instance Clusters on Clouds.” 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 929-934.

[66] Bermejo, B., Filiposka, S., Juiz, C., Gómez, B., & Guerrero, C. (2017). “Improving the Energy Efficiency in Cloud Computing Data Centres Through Resource Allocation Techniques.” Research Advances in Cloud Computing.

[67] Moreno, I.S., Yang, R., Xu, J., & Wo, T. (2013). “Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement.” 2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS), 1-8.

[68] Zhang, Q., Cai, Y., Chen, X., Angel, S.G., Chen, A., Liu, V., & Loo, B.T. (2020). “Understanding the effect of data center resource disaggregation on production DBMSs.” Proceedings of the VLDB Endowment, 13, 1568 - 1581.

[69] Jiang, D., Pierre, G., & Chi, C. (2009). “EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications.” ICSOC/ServiceWave Workshops.

[70] Chai, L., Hartono, A., & Panda, D.K. (2006). “Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters.” 2006 IEEE International Conference on Cluster Computing, 1-10.

[71] Milic, U., Villa, O., Bolotin, E., Arunkumar, A., Ebrahimi, E., Jaleel, A., Ramírez, A., & Nellans, D.W. (2017). “Beyond the Socket: NUMA-Aware GPUs.” 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 123-135.

[72] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., & Tatbul, N. (2019). “Neo: A Learned Query Optimizer.” Proc. VLDB Endow., 12, 1705-1718.

[73] Wu, C., Jindal, A., Amizadeh, S., Patel, H., Le, W., Qiao, S., & Rao, S. (2018). “Towards a Learning Optimizer for Shared Clouds.” Proc. VLDB Endow., 12, 210-222.

[74] Lee, K., & Liu, L. (2013). “Efficient data partitioning model for heterogeneous graphs in the cloud.” 2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 1-12.

[75] Nehme, R.V., & Bruno, N. (2011). “Automated partitioning design in parallel database systems.” SIGMOD ’11.

[76] Dreseler, M., Boissier, M., Rabl, T., & Uflacker, M. (2020). “Quantifying TPC-H choke points and their optimizations.” Proceedings of the VLDB Endowment, 13, 1206 - 1220.

[77] Lübcke, A. (2017). Automated query interface for hybrid relational architectures.

[78] Bui, V.Q., Mvondo, D., Teabe, B., Jiokeng, K., Wapet, P.L., Tchana, A., Thomas, G., Hagimont, D., Muller, G., & Palma, N.D. (2019). “When eXtended Para - Virtualization (XPV) Meets NUMA.” Proceedings of the Fourteenth EuroSys Conference 2019.

[79] Pacheco, P.S. (2011). An Introduction to Parallel Programming.

[80] Engler, D.R., Kaashoek, M.F., & O’Toole, J.W. (1995). “Exokernel: an operating system architecture for application-level resource management.” Proceedings of the fifteenth ACM symposium on Operating systems principles.

[81] Accetta, M.J., Baron, R.V., Bolosky, W.J., Golub, D.B., Rashid, R.F., Tevanian, A., & Young, M. (1986). “Mach: A New Kernel Foundation for UNIX Development.” USENIX Summer.

[82] Jebalia, M., LETAIFA, A.B., Hamdi, M., & Tabbane, S. (2013). “A Comparative Study on Game Theoretic Approaches for Resource Allocation in Cloud Computing Architectures.” 2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 336-341.

[83] Dinesh, K., Poornima, G.R., & Kiruthika, K. (2012). “Efficient Resources Allocation for Different Jobs in Cloud.” International Journal of Computer Applications, 56, 30-35.