Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores

Iraklis Psaroudakis* ‡ Tobias Scheuer‡ Norman May‡
Abdelkader Sellami‡ Anastasia Ailamaki*

*EPFL, Lausanne, Switzerland
{first-name.last-name}@epfl.ch

‡SAP SE, Walldorf, Germany
{first-name.last-name}@sap.com

ABSTRACT

Non-uniform memory access (NUMA) architectures pose numerous performance challenges for main-memory column-stores in scaling up analytics on modern multi-socket multi-core servers. A NUMA-aware execution engine needs a strategy for data placement and task scheduling that prefers fast local memory accesses over remote memory accesses, and avoids an imbalance of resource utilization, both CPU and memory bandwidth, across sockets. State-of-the-art systems typically use a static strategy that always partitions data across sockets, and always allows inter-socket task stealing.

In this paper, we show that adapting data placement and task stealing to the workload can improve throughput by up to a factor of 4 compared to a static approach. We focus on highly concurrent workloads dominated by operators working on a single table or table group (copartitioned tables). Our adaptive data placement algorithm tracks the resource utilization of tasks, partitions of tables and table groups, and sockets. When a utilization imbalance across sockets is detected, the algorithm corrects it by moving or repartitioning tables. Also, inter-socket task stealing is dynamically disabled for memory-intensive tasks that could otherwise hurt performance.

1. INTRODUCTION

Processor vendors are scaling up modern servers by interconnecting multiple sockets in a single shared-memory system. Each socket has a memory controller and multiple cores attached, introducing new performance challenges for software. There are non-uniform memory access (NUMA) latencies across the system. The resources, either CPU or memory bandwidth, of a single socket, as well as the bandwidth of a single interconnect link, are additional bottlenecks to be considered. Contemporary main-memory column-store database management systems (DBMS), such as SAP HANA [11] or Oracle [23], need to tackle the challenges of data placement and scheduling in order to scale up on modern NUMA hardware and efficiently service highly concurrent big data analytics.

In order to balance utilization across sockets, state-of-the-art systems [16, 20] partition data across sockets and employ task scheduling with inter-socket task stealing. Our previous analysis of concurrent NUMA-aware scans [28] showed that such a static strategy for data placement and task scheduling is not always the most performant one. We showed that unnecessary partitioning can incur an overhead and that stealing memory-intensive tasks can hurt overall performance, depending on the workload.

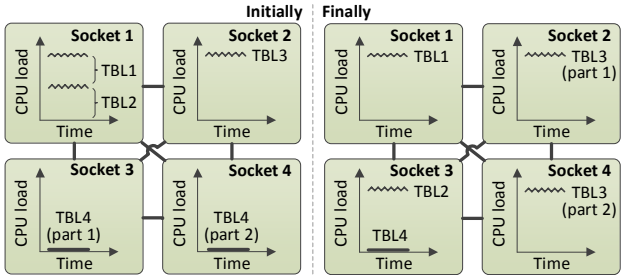

Figure 1: A conceptual example of our adaptive data placement.

In this paper, we show that the implications for adaptivity apply to additional NUMA-aware operators, focusing on aggregations and equi-joins (see Section 4). We attempt to solve the open problem of adapting data placement and task scheduling to the workload at runtime, with the aim to balance resource utilization across sockets. We target highly concurrent workloads dominated by operators working on a single table or table group (copartitioned tables).

Our proposed design relies on tracking the history of CPU and memory bandwidth utilization at three levels (see Section 5): (a) tasks, (b) partitions of tables and table groups, and (c) sockets. When the execution engine detects a utilization imbalance across sockets, it either moves or repartitions tables in order to fix the imbalance (see Section 6). Moreover, it also finds cold partitioned tables to consolidate and disallows inter-socket stealing of memory-intensive tasks that would hurt performance (see Section 7).

Figure 1 shows a conceptual example of the most significant aspects of our adaptive techniques. The server has four fully interconnected sockets. The workload consists of numerous concurrent memory-intensive scans on three tables, which are initially placed on two of the server's sockets. Task stealing is disallowed due to the memory intensity of the scans. Two of the sockets are fully utilized, and their memory bandwidth is saturated, while the remaining two sockets are idle. Our adaptive data placement detects the utilization imbalance, and takes actions to fix it. It moves table *TBL2* to socket 3, partitions *TBL3* across sockets 2 and 4, and finally merges the unutilized parts of *TBL4*. Socket utilization becomes balanced. The total memory throughput is 2x higher than initially, improving the workload's throughput by 2x (see Section 8.1).

Contributions. In this paper, we present adaptive NUMA-aware techniques for main-memory column-stores. We adapt data placement and inter-socket task stealing to workloads dominated by operators working on a single table or table group (copartitioned

Proceedings of the VLDB Endowment, Vol. 10, No. 2

tables), at run-time. Our implementation and experiments are based on a commercial column-store (SAP HANA). Our contributions are:

- An adaptive data placement strategy that can improve throughput by up to 2x in comparison to a static strategy that always partitions data across all sockets. We present an adaptive heuristic algorithm that moves and repartitions tables at run-time in order to balance the utilization across sockets.
- Adapting inter-socket task stealing to the memory intensity of tasks, improving throughput by 1.1x–4x in comparison to a static strategy that always allows inter-socket stealing.
- To adapt data placement and task stealing, complete knowledge of the system's utilization is needed. We present a design that tracks the utilization history at the levels of (a) tasks, (b) partitions of tables and table groups, and (c) sockets.

2. BACKGROUND

In this section, we give a brief overview of NUMA-awareness, data placement and task scheduling in main-memory column-stores.

NUMA. Processor vendors scale up modern machines with a non-uniform memory access (NUMA) architecture. Figure 2a shows an example of a 4-socket server. Each socket has a 15-core Intel Xeon E7-4880v2 CPU. Each core has its own L1 and L2 caches, and a socket's cores share a L3 cache. Eight 16 GB DIMM are attached to each socket. The sockets are interconnected to exchange data requests and support cache coherence. In this example, each socket has 3 QPI links, forming a fully-interconnected topology (maximum of 1 hop across sockets). The topology, the interconnects, and the cache coherence protocol are specific to each system.

NUMA-awareness. Since memory is distributed, new performance challenges arise for software: (a) accesses to remote memory can be up to 5x slower than local memory, (b) the bandwidth of a socket and an interconnect can be additional bottlenecks, and (c) the bandwidth of an interconnect can be up to 7x lower than the bandwidth of a socket [7, 28]. Due to the lack of knowledge about inter-socket routing or the cache coherence, a NUMA-aware application attempts to solve the above challenges in a simple way: optimizing for local memory accesses instead of remote accesses, and avoiding unnecessary centralized bandwidth bottlenecks.

Memory management in the operating system. The OS organizes memory with (typically) 4 KB pages [17]. The physical location of a virtual memory page, which an application has allocated, is decided upon the first page fault. In Linux, the default "first-touch" policy attempts to allocate physical memory for a virtual page from the socket where the thread is running. Linux provides NUMA-aware functions for an application, such as `mbind` or `move_pages`, to set and get the physical location of its allocated virtual memory.

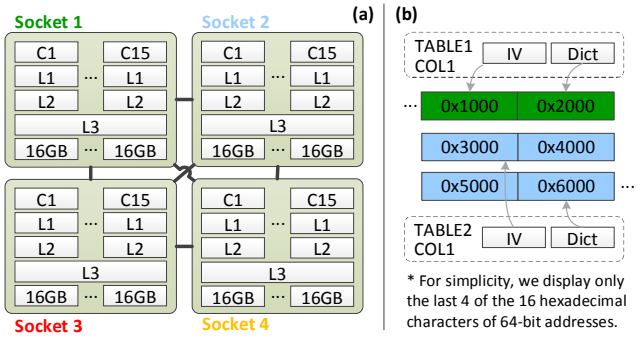

Figure 2: (a) 4-socket server. (b) Example of the physical location of the virtual memory of two tables on a 4-socket server.

Main-memory column-stores. The data of a column can be stored sequentially in a vector in main-memory [11, 19, 20, 23]. Compression techniques, such as dictionary encoding, are typically used to reduce the amount of memory and potentially speed up processing [21]. A generic dictionary-encoded column is composed of an integer vector, called ***indexvector (IV)*** (naming can be different), that stores ***value IDs (vid)***, and the ***dictionary*** vector, that stores the sorted unique real values of the value IDs [28].

Figure 2b shows an example of the physical location of the virtual memory of two tables with one column each. Assuming the same data type and page-aligned allocations, the example hints that Table 2 has around triple number of rows than Table 1 with a similar number of unique dictionary values. The DBMS has used OS NUMA-aware functions to place Table 1 on Socket 1 and Table 2 on Socket 2.

Data placement. An entire table can be placed on one socket as in Figure 2b, or can be physically partitioned across several sockets [1, 23]. By using, e.g., hash, range, or round-robin partitioning, we can define multiple ***table parts (TBP)*** [28]. All TBP share the same set of columns, but each column in a table part has its own IV and dictionary. As such, a table part can be entirely placed on one socket.

Task scheduling. With task scheduling, operations are encapsulated in tasks, which are put into task queues, and pools of worker threads are used to process them. The task scheduler can reflect the NUMA topology of a machine [16, 20, 27, 29]. In our previous work, we detailed our NUMA-aware task scheduler [27, 28]. We showed how stealing and a concurrency "hint" can help to saturate CPU resources without unnecessary scheduling overhead and that stealing memory-intensive tasks can hurt performance. In this work, we adapt task stealing to a task's memory intensity (see Section 7).

3. RELATED WORK

We organize related work by static NUMA-aware solutions, adaptive solutions, black-box solutions, and work in distributed systems.

Static solutions. Most DBMS not mentioning advanced NUMA optimizations indirectly rely on the static first-touch policy for data placement, e.g., Vectorwise [39], Microsoft SQL Server's column-store [19], or IBM DB2 BLU [30]. In a recent thesis describing how to parallelize query plans in Vectorwise with task scheduling [14], inter-socket stealing is allowed based on task priorities and the contention of sockets. In this work, we show that stealing should not be allowed for memory-intensive tasks. Oracle's distributed manager decides the NUMA location of columnar data when the topology changes [23], but not when the workload changes. HyPer [20] chunks all data, and statically distributes them uniformly over the sockets, while inter-socket stealing is always enabled.

There is also related work on NUMA-aware standalone operators. Albutiu et al [5] construct a NUMA-aware sort-merge join. Hash-joins, however, are shown to be superior [6, 18]. Yinan et al [22] optimize data shuffling. Most related work, however, optimize for low concurrency with a static data placement using all sockets of the server. In this work, we optimize for highly concurrent workloads, with a data placement that adapts to the workload.

Adaptive solutions. Two state-of-the-art research prototypes use an adaptive NUMA-aware solution for data placement: ERIS [16] and ATraPos [26]. ERIS is a storage manager that employs adaptive range partitioning, and each partition is assigned to a worker thread. While ERIS targets storage operations, we target analytical workloads consisting of numerous operators. In addition, we show in this paper

how partitioning involves an overhead that can be avoided altogether depending on the workload, in which case intermediate results can also be local to process within a socket. ATraPos uses dynamic repartitioning for OLTP workloads, to avoid transactions crossing partitions and avoid inter-partition synchronization. While ATraPos optimizes for the latency of transactions, we focus on optimizing the throughput of analytical workloads by balancing socket utilization.

Black-box approaches. These approaches monitor performance metrics to predict an application's behavior and periodically move threads and data to balance cache load, optimize for local memory accesses, and avoid bandwidth bottlenecks. Examples include DINO [7], Carrefour [8], or the new automatic NUMA balancing of Linux. Results for DBMS, however, are typically sub-optimal. Giceva et al. [12] employ a DBMS-focused black-box static approach to characterize and group the shared operators of a predefined global query plan, and place them on a NUMA server with the main aim of improving overall energy efficiency. In this work, we also employ a DBMS-focused black-box approach, but geared towards adapting data placement and task scheduling at run-time.

Distributed systems. It is long known that the data placement problem in distributed systems is NP-hard [10]. The input is a characterization of the data and a workload. We refer to [25] for a discussion of solution methods. The most advanced method we are aware of relies on a reduction to a graph partitioning problem which is passed to a heuristic solver that may need minutes to run [13].

Similar solvers are employed in several distributed DBMS tools. Examples include the SAP Data Distribution Optimizer (DDO) [2], the physical database design advisor in DB2 [31], the database tuning advisor in MS SQL Server [4], and Oracle's distribution manager [23]. The produced data placement is static but the solver can be triggered again manually or automatically after a change in the network topology. The data placement, however, does not adapt automatically to new workload characteristics. In our experience workloads are rarely completely predictable, and there is a tendency towards highly concurrent workloads generated by several applications. Our aim is to adapt the data placement across NUMA nodes to the workload at run-time. The aforementioned solvers cannot quickly adapt to a changing workload and cannot be immediately applied to our dynamic setting. Our heuristic algorithm, however, considers only a few alternative placements, and can quickly adapt to the workload.

Partitioning specifications and table groups. Our adaptive data placement uses two notions found in automated distributed setups [1, 2]. First, only tables with a defined ***partitioning specification*** (e.g., hash partitioning on a column) are automatically repartitioned. Second, ***table groups (TG)*** can be defined to recognize associated tables used by multiple-input operators. We track utilization at the level of TG, instead of tables (see Section 5). When our adaptive data placement decides to move or repartition a TG across sockets, it does so for all the tables of the TG. Equi-joins on tables of a TG that are partitioned over the joined columns (copartitioned tables), can be executed mostly locally at the sockets with the collocated table parts [1]. TG and partitioning specifications can be defined manually by the administrator, suggested by one of the aforementioned solvers for a workload, or given for popular workloads, e.g., SAP BW [1].

4. NUMA-AWARE OPERATORS: THE NEED FOR ADAPTIVITY

In our previous work [28], we analyzed NUMA-aware scans. As a reminder, a scan has two phases: (a) a memory-intensive phase that scans the bit-compressed IV (with SIMD instructions [36]) for qualifying rows for a given predicate, and (b) a more CPU-intensive (with less memory throughput) phase that materializes the output values corresponding to the *vid* of the qualifying rows by consulting the dictionary. The first phase is parallelized with multiple tasks, and the second is parallelized in case of large results. Each phase is repeated concurrently for all TBP. Task scheduling is NUMA-aware, by queuing each task to the socket where its associated TBP is.

For scans, we identified the overhead of unnecessary partitioning and of stealing memory-intensive tasks, showing the need for adaptive data placement and task stealing. For this paper, we make more operators NUMA-aware, focusing on aggregations and equi-joins. Here, we briefly describe their NUMA-aware implementation, and then show that the same need for adaptivity applies to them as well.

Aggregations. An aggregation uses the scan's first phase to find the qualifying rows, and then is parallelized with multiple tasks, where each task executes two phases: (a) aggregating using a local ***hash table (HT)***, and (b) merging the local HT to a set of disjoint result HT [33]. The reason for the two phases is that they are interchanged potentially multiple times in order to avoid large local HT that do not fit in the processor's last level cache (LLC) [24, 33]. We note that we focus on decomposable associative aggregation functions.

If there are multiple TBP, an additional phase precedes, that creates a global dictionary (used in the subsequent aggregation phases) by matching the *vid* of the qualifying rows of the group-by column from each TBP. Tasks are scheduled in a NUMA-aware fashion similar to the scans. Although local HT are placed on a task's socket, the global dictionary is accessed by all tasks, and is placed on one of the sockets of the involved TBP. Each disjoint result HT is placed on one of the involved sockets in a round-robin manner. During the merge phase, a task tries to merge a result HT that lies on its socket, but necessarily accesses the local HT that can lie on multiple sockets.

Equi-joins. The NUMA-aware implementation of equi-joins is similar to their distributed implementation [15, 32]. Consider an example of an equi-join of two tables with a selection predicate on the first table. The first table is searched to find qualifying rows for the predicate, and the corresponding *vid* of the join column. A global dictionary is employed, similar to partitioned aggregations, to map the *vid* of the join columns between the tables. This is done by consulting the join columns' dictionaries. The join column of the second table is searched for the qualifying *vid* and rows. A reduced set of *vid* is then used to filter rows of the first table whose *vid* did not occur in the second table. With the help of the global dictionary, the second table is searched for the matched rows. The matched rows from both tables are joined, with the help of the global dictionary, to produce the final result. All steps are parallelized with multiple tasks, which are scheduled in a NUMA-aware fashion, best when the tables are on the same socket. If they are on different sockets, the tasks incur remote accesses when mapping the *vid* of the join columns, and during the final result production.

If there are multiple TBP for the tables, the aforementioned steps are done by visiting all involved TBP to map their *vid* using the global dictionary, and match the qualifying rows to produce the final result. The mapping of the *vid* is more expensive to compute, since each TBP has its own dictionary. Remote accesses can be increased considerably if the TBP reside on multiple sockets. There is one case when partitioning does not incur additional overhead for mapping *vid*, and remote accesses are largely avoided. If the tables are partitioned on the joined column, the partitioning specifications of the tables match, and each pair of TBP (of the two tables) reside on the same socket, the copartitioned equi-join can be parallelized and executed locally on the sockets where the pairs of TBP reside. Our adaptive data placement exploits this special case when the administrator has defined TG that contains the joined tables, and

partitioning specifications on the joined column. Joins on other columns, or between tables outside a TG, are not guaranteed to have mostly local accesses. This is similar to distributed joins with partitioned tables [1].

The overhead of partitioning and stealing. Here, we experimentally show the overhead of unnecessary partitioning and stealing for both aggregations and joins, pointing to the necessity for adaptivity. Partitioning should be used in skewed workloads to balance used data across sockets. Stealing should be used for CPU-intensive tasks to balance CPU load across sockets, but not for memory-intensive tasks that would overwhelm an already saturated socket.

Figure 3 shows the throughput (TP) of concurrent aggregations or joins under different cases of selectivity, available tables, task stealing, and partitioning (TBP/table). See Section 8 for more details on our methodology, dataset, and query types. Aggregations use query type b: each query picks a random table out of the available tables and aggregates a column with a group-by. Joins use a slightly modified version of query type c: a query picks a random pair of tables (either TBL1-TBL2, TBL3-TBL4 etc.) to join on their COL1 instead of the ID primary key (PK) column, with a filter predicate as well. In the case of 1 TBP/table, the table pairs are placed in a round-robin way around the sockets. In the case of 4 TBP/table, tables are partitioned on the PK, and the TBP of a table pair are collocated on the same socket. This configuration can show the worst overhead of partitioning. The server used is the one of Figure 2a. All cases saturate CPU load, apart from the case of 1 (or 2 for joins) table with 1 TBP/table without stealing, which saturates one socket.

Low selectivity aggregations are dominated by IV scans. The workload is memory-intensive, as can be seen by the high memory TP. As far as stealing is concerned, it helps saturate the CPU load (for the case of 1 table with 1 TBP), but it does not improve the TP as the workload is memory-intensive and stealing is unnecessary. In fact, it can hurt TP by up to 15% (see case of 8 tables with 4 TBP vs. not stealing). As far as partitioning is concerned, it can greatly help improve the memory TP, improving the TP by up to 3x in case the workload is skewed (see case of 1 table with 1 TBP vs. 4 TBP). But when partitioning is unnecessary, it has an overhead of up to 25% (see case of 8 tables with 1 TBP vs. 4 TBP).

As we detected in our previous work for scans [28], the partitioning overhead involves at least a scheduling overhead that depends on the implementation, due to a query needing to visit all TBP on multiple sockets. Irrespective of the implementation, an additional overhead lies in the second scan phase when outputting the real values into a single result set, or, in case of partitioned sub-results, when fetching the sub-results from multiple sockets. In our experiments, we wish to consider the NUMA, and not the network, effect so the second scan phase outputs a single result set and we ignore fetching. Another overhead is the need of a global dictionary, plus the remote accesses during the merge phase. Remote accesses during the merge phase are necessary irrespective of the implementation [24, 37], and depend on the number of groups (see Section 8.3 for a relevant experiment).

High selectivity aggregations are dominated by the aforementioned aggregation phases. Due to the random accesses and hashing involved, the workload is more CPU-intensive without achieving a high memory TP. Stealing is now helpful. Stolen tasks are CPU-intensive and do not run the risk of overwhelming the remote memory controller or the interconnects. It can help saturate CPU load and improve TP by up to 2.5x (see case of 1 table with 1 TBP). The implications about partitioning are the same: it can help when the workload is skewed, otherwise it has an overhead (see case of 8 tables with 1 TBP vs. 4 TBP). The reasons for the overhead are the same as in the case of low selectivity.

Joins are also CPU-intensive and stealing helps. The overhead

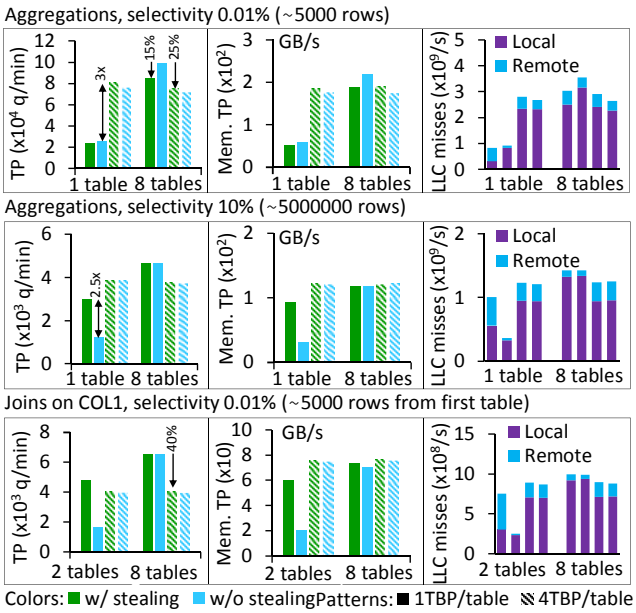

Figure 3: 256 clients issuing aggregation or join queries on a 4-socket server, under different cases of selectivity, available tables, stealing, and partitioning. Each LLC misses result belongs to the corresponding case of the left-hand side graphs.

of unnecessary partitioning, however, is aggravated for joins that are not copartitioned. It reaches up to 40% (see case of 8 tables with 1 TBP vs. 4 TBP). The overhead is due to mapping the *vid* of all involved TBP of both tables, and due to accessing TBP on all sockets for producing the final results. This overhead does not apply to copartitioned joins or unpartitioned tables.

To sum up, partitioning should be used only when necessary to balance utilization, and stealing should not be used for memory-intensive tasks. In the next sections, we detail how we track utilization and how our adaptive data placement algorithm works. Afterwards, we detail our adaptive task stealing strategy.

5. TRACKING RESOURCE UTILIZATION

We track resource utilization (CPU load and memory throughput) across three levels: (a) tasks, (b) partitions of tables and table groups, and (c) sockets. Figure 4 depicts our monitoring infrastructure with an example of concurrent low selectivity scans (dominated by the scan's memory-intensive first phase that scans the IV).

Task classes. We organize tasks into classes that have similar functionality and memory throughput. We use a different class for the tasks of the first phase of a scan (IV scan), for the tasks of the second phase of a scan (materialization), for the aggregation tasks that interchange between the two aggregation phases, and for every step of a join (see Section 4). For each class, we track the observed memory throughput with a single exponential moving average.

Scheduling. When a task is scheduled on its intended socket, where its associated TBP is placed, we aggregate its utilization on the level of the TBP and the socket. We atomically add the average memory throughput of the task's class to the memory throughput consumed by the TBP. We also atomically increment the CPU load (number of threads) utilized by the TBP and the socket. When the task either finishes or blocks in a synchronization primitive, we atomically subtract the memory TP we previously added and decrement the CPU load. With this method, we can keep track of the current local CPU load and the estimated local memory throughput of TBP

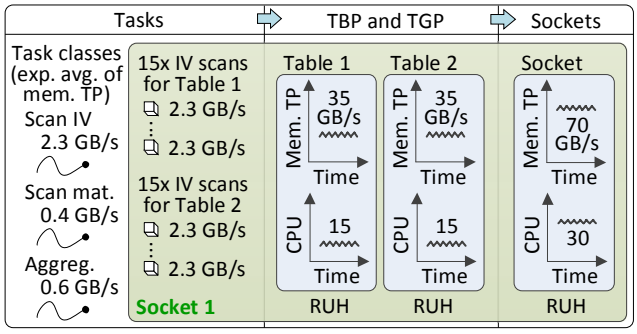

Figure 4: Tracking the resource utilization of tasks, TBP and TGP, and sockets. Exemplary values shown for concurrent scans on two tables on socket 1 of our 4-socket server.

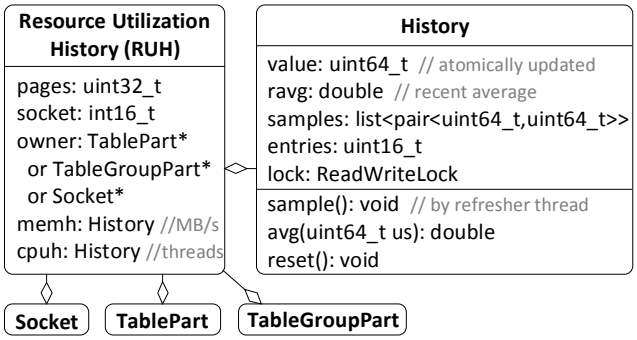

Figure 5: Each table part (TBP), table group part (TGP), and socket is associated with a resource utilization history (RUH).

and sockets. We ignore on purpose stolen tasks. The utilization of sockets that steal appear non-saturated in our metrics, as stealing is a temporary solution to balance CPU load until our adaptive data placement algorithm (see Section 6) fixes the imbalance of local utilization. In the example of Figure 4, we depict how the average memory TP of a task class is used when scheduling concurrent tasks to aggregate the utilization at the level of TBP and sockets.

Measuring memory throughput. A task class is needed to assign an estimated memory TP to a task when it is about to run. After the task ends, we calculate its consumed memory TP, and push back the value to the exponential moving average of its class with a low weight: classAverage = 0.9 ∗ classAverage + 0.1 ∗ newValue. Memory TP is calculated through H/W counters (integrating the Intel PCM tool [9]), and considered only for non-stolen tasks that have not blocked or moved to another core during execution. The formula is: memoryTP = $((\text{localLLC}_{\text{end}} - \text{localLLC}_{\text{start}}) * 64)/(\text{timestampUS}_{\text{end}} - \text{timestampUS}_{\text{start}})$. This calculates the accessed bytes by associating every local LLC miss with a cache line (64 bytes) retrieval. Dividing by the task's duration, gives the average memory TP in MB/s. We ignore on purpose remote LLC, since we do not wish to track remote memory TP in the task classes.

Table group parts (TGP). If a TG is defined for a group of tables, tasks do not aggregate their utilization at the level of TBP, but at the level of ***table group parts (TGP)***. For example, if the tables are partitioned with three TBP each, then we keep three TGP. Each one of the TGP tracks the aggregated utilization of the corresponding TBP of the tables. We note that we place the TBP of a TGP on the same socket, thus a TGP is associated with a single socket. If a table of the TG does not have a partitioning specification, it is considered as a single TBP and placed on the first TGP.

Resource utilization histories (RUH). Every TBP, TGP, and socket, has a RUH that keeps track of the memory throughput consumed and the number of threads used. The components of a RUH are shown in Figure 5. It contains the number of memory pages occupied by the TBP or TGP or socket, a pointer to the owner TBP or TGP or socket, the socket where the owner is placed, and two ***history*** objects that capture the recent utilization for memory throughput (`memh`) and threads (`cpuh`). Specially for the pages of a TGP, we do not sum the pages of its TBP, but keep the maximum pages of any of its TBP. This is because we later use the pages to estimate the moving or partitioning time, and we can move/partition a TG's TBP in parallel.

A History object contains the absolute value that, as explained before, is atomically incremented by a task when it is scheduled and decremented when it blocks or is de-scheduled. The absolute value is periodically sampled into a list of pairs of timestamps and values. We maintain one background ***refresher thread*** per socket, that periodically calls `sample()` on the histories of its socket and of a subset of all TBP and TGP. `sample()` appends a pair with the current timestamp and value to the back of the samples list, and increments the entries. If the entries grow over a specified limit, it deletes the pair at the front of the list. In our current implementation, a refresher thread runs every 100 ms, and the limit of the samples list is set to 3000 entries, which means that it can reach up to around 47 KB, holding samples for roughly the last 5 min.

We note that tasks atomically modify the absolute values of the RUH of TBP or TGP only. A socket's absolute values are periodically calculated by the corresponding refresher thread by aggregating the absolute values of all TBP and TGP placed on the socket. This is to avoid numerous atomic operations at the level of a socket, since many TBP and TGP can be placed on it, and because we use a socket's utilization mostly for monitoring purposes. The adaptive data placement algorithm of Section 6 calculates the sockets' utilization by aggregating the utilization of TBP and TGP, in order to work on a single "snapshot" of how the sockets' utilization is composed by the involved TBP and TGP.

The adaptive data placement algorithm makes heavy usage of the `avg(uint64_t us)` function of the RUH, which runs the sample list to find the first entry that is not older than the given microseconds, and starts averaging the entries up to the last entry (each entry is given a weight equal to the microseconds passed since the previous entry), finally returning the average utilization. If the given microseconds would require an entry older than the first of entry of the samples list, we assume that the value of the utilization during that period is equal to the value of the first entry.

Due to synchronization issues with the refresher threads, a read-write lock is used by both the refresher threads (which write) and the adaptive data placement algorithm (which reads). There are minimal synchronization issues due to the periodicity of the threads, and due to the fact that every refresher thread processes a different socket and a different subset of TBP and TGP.

Finally, we also keep a shortcut average value for the last microseconds that correspond to the period with which the adaptive data placement algorithm runs. At the end of every `sample()` call, the recent average (`ravg`) is updated by calling `avg()` with the period of the adaptive data placement algorithm. The algorithm can use this value immediately, without having to calculate it with `avg()`.

6. ADAPTIVE DATA PLACEMENT

First, we describe the abstract workflow of our adaptive data placement algorithm. Then we gradually delve into algorithmic details.

6.1 Abstract Workflow

The main component of our adaptive data placement is the ***Data Placer (DP)*** thread. Figure 6 shows its abstract workflow. The first time tables are loaded into memory, they can be placed across sockets in a round-robin manner. DP runs periodically in the background to monitor the workload, and automatically takes care to either move or repartition tables to fix a utilization imbalance.

DP focuses on balancing CPU utilization, under the constraint of not creating a memory bandwidth bottleneck. We remind that we refer to local-only utilization as tracked in Section 5. We balance the utilization between sockets with saturated CPU resources and colder sockets, by moving or partitioning tables. This strategy allows tables that were previously on saturated sockets to potentially increase their utilization using free threads on colder sockets (due to intra-query parallelism), increasing the total system utilization.

At every period, DP gets a snapshot of the active RUH (of TBP and TGP) and their recent utilization. It then calculates their eligibility for moving and partitioning, depending on whether their past utilization has been stable. Then, DP sorts the RUH within each socket by their recent utilization, aggregating them as well to calculate the recent utilization of the sockets. Afterwards, DP calculates the CPU utilization imbalance between all pairs of sockets, and sorts the pairs.

For every pair of sockets, DP investigates whether a new placement can reduce the imbalance. DP proceeds only if the imbalance is over a threshold, and if the hot socket is saturated. If the hot socket is not saturated, the TBP and TGP cannot increase their utilization by exploiting free threads on the cold socket. DP iterates the RUH of the hot socket, and examines whether moving or partitioning an eligible RUH's owner (the corresponding TBP or TGP) reduces the imbalance. If additionally it does not create a memory bandwidth bottleneck, DP proceeds to move or partition the RUH's owner.

The outlined steps in Figure 6 are first executed while considering moving an eligible RUH's owner. If none can be moved across all socket pairs, we repeat the steps considering partitioning an eligible RUH's owner. This ensures we first prefer moving over partitioning, to avoid any unnecessary overhead of partitioning (see Section 4).

In case DP did not move or partition a RUH's owner, it goes on to see if there are any cold partitioned tables to merge. This optional step can give the opportunity for cold tables previously partitioned to not suffer the partitioning overhead in case they are again utilized in the future (see Section 8.3 for a relevant experiment).

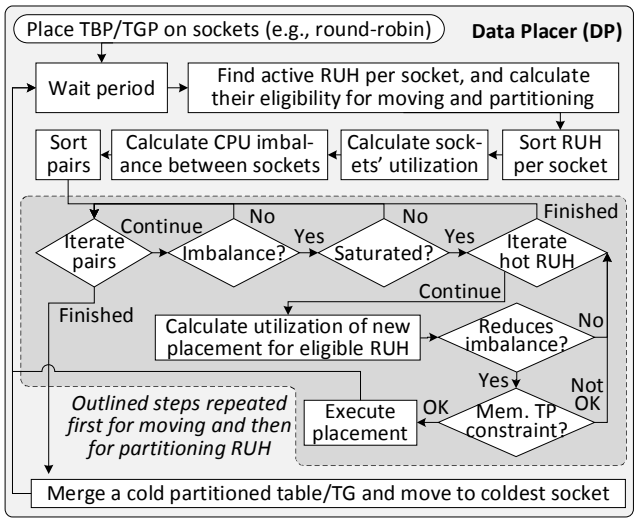

Figure 6: Abstract workflow of the Data Placer (DP).

Next, we detail how the eligibility of RUH is calculated (see Section 6.2), how we reduce the utilization imbalance by moving or partitioning (see Section 6.3), and finally how all pieces are put together in DP's algorithm (see Section 6.4). The parameters used in our algorithms are summarized in Table 1. The exemplary values are used for our experiments. They are not absolute and can be modified to fit other systems, implementations and use cases.

Table 1: Configurable parameters used in our algorithms, along with the values we use in our experiments.

Symbol	Description	Our value
c_p	Period of the Data Placer (DP) algorithm	1 second
c_e	Eligibility threshold for the divergence between the past and recent utilization of a RUH	30%
c_i	Acceptable imbalance threshold between the utilization of a pair of sockets	40% of socket cores
c_s	Lower threshold for considering a socket's utilization saturated	70% of socket cores

We assume that each socket corresponds to a NUMA node [17], and that all sockets have the same number of H/W threads and maximum memory TP. This assumption is for typical NUMA servers with the same processors, and the same number and type of DIMM.

6.2 Information and Eligibility of RUH

At every period, DP finds the RUH of all TBP and TGP, takes a snapshot of their recent utilization and calculates their eligibility to be moved or partitioned. For every RUH, this information is stored in an ***InfoRUH*** object, which is defined in Algorithm 1, and calculated through the function `calculateInfoRUH`.

The algorithm first stores the recent CPU and memory throughput utilization of the RUH (lines 2–3). The RUH is deemed active if its utilization is non-zero (line 4). DP continues to calculate the eligibility of the RUH for moving or partitioning. A RUH is deemed eligible if its average utilization in the past does not diverge much from its recent utilization. The amount of time we look in the past depends on the implementation of the move or partition operation.

For the time to look in the past in the case of moving, we first calculate the time required to move the RUH's owner (line 5), by multiplying its pages with the speed of moving (microseconds per page). The speed of moving and partitioning are calculated at start-up by moving or partitioning a simple mock-up table to another socket, without a concurrent workload. See Table 2 for the speeds of the machines we use in our experiments. The speeds are rough estimates. One can improve accuracy by specializing the speeds by socket, or the concurrent workload, or a table's characteristics such as the number of columns, data types, etc. However, we do not need to be precise, since the aim of our eligibility calculations is to disallow instant actions by DP and not delaying them for long.

In our implementation, queries need to wait while a TBP is moved. We use SAP HANA's functionality to unload a TBP from memory and reload it on the desired socket. We do not use Linux's `move_pages`, because it would mess up the statistics of SAP HANA's NUMA-aware memory allocators [35]. Due to queries waiting during the move, we double the time to look in the past (line 5). This is optional and simply prolongs the amount of time to look in the past. Conceptually, the additional time corresponds to the time required to "recover" the utilization which drops to zero during the move. We then calculate the average past utilization of both CPU and memory TP (lines 6–7). The RUH is eligible for moving if the past utilization is within a threshold of the recent utilization (line 8).

The algorithm then continues similarly for calculating the eligibility of partitioning (lines 9–14). There are three differences. First, we require that the RUH is also eligible for moving (line 9). This is to enforce the preference of DP to having first considered moving the

Algorithm 1 Calculate information and eligibility of a RUH

```
    struct InfoRUH:
        RUH;  // pointer to corresponding RUH object
        recentCpu;  // recent CPU utilization
        recentMem;  // recent memory throughput utilization
        isActive;  // whether the recent utilization is non-zero
        canMove;  // eligible for moving
        canPartition;  // eligible for partitioning
 1: function CALCULATEINFORUH(corresponding RUH)
 2:     recentCpu ← RUH.cpuh.ravg
 3:     recentMem ← RUH.memh.ravg
 4:     isActive ← (recentCpu > 0 and recentMem > 0)
 5:     usMove ← (RUH.pages * speed of moving) * 2
 6:     pastCpu ← RUH.CPUH.AVG(usMove)
 7:     pastMem ← RUH.MEMH.AVG(usMove)
 8:     canMove ← (| recentCpu - pastCpu | < c_e * recentCpu) and
                  (| recentMem - pastMem | < c_e * recentMem)
 9:     canPartition ← canMove and 2 * current partitions ≤ sockets
10:     if canPartition then
11:         usPartition ← usMove + (RUH.pages * speed of partitioning)
12:         pastCpu ← RUH.CPUH.AVG(usPartition)
13:         pastMem ← RUH.MEMH.AVG(usPartition)
14:         canPartition ← (| recentCpu - pastCpu | < c_e * recentCpu) and
                           (| recentMem - pastMem | < c_e * recentMem)
```

RUH's owner before considering partitioning it. Second, the amount of time to look in the past consists of partitioning plus the time required to move the new partitions to the correct sockets (line 11). We use SAP HANA's partitioning commands, which, contrary to the implementation of moving, creates the partitions in the background and allows queries [1]. Third, we limit the number of new partitions to the number of sockets to avoid excessive partitioning (line 9).

We note that in this work when we partition a TBP or TGP, we partition the corresponding table or TG into double their previous number of partitions. The reasons why we double the partitions are two. First, partitioning is more time-consuming than moving. Since we decide to partition, we can immediately have double number of partitions and give the algorithm more parts that can potentially be moved later. Second, repartitioning with a number of partitions that is a multiple of the previous number of partitions is fast since each existing partition can be split separately and concurrently [1].

As an illustrative example, Figure 7 depicts two RUH, one that is eligible for partitioning, and one that is not. Both RUH have similar recent utilizations. The utilization of the first one, however, is not stable in the past, and is thus ineligible for partitioning yet.

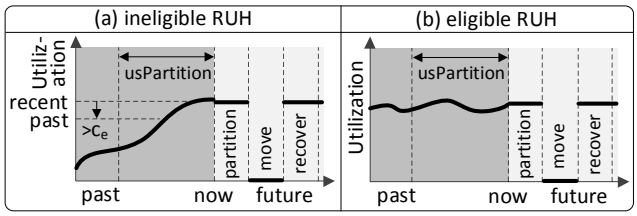

Figure 7: Conceptual examples of calculating the partitioning eligibility of (a) an ineligible RUH, and (b) an eligible RUH.

6.3 Reducing the Utilization Imbalance

The purpose of balancing the CPU utilization between sockets is to allow tables to increase their utilization by exploiting free threads on cold sockets when moved or partitioned out of saturated sockets. When considering moving or partitioning a tables, however, we assume the worst case that it does not increase its utilization. This allows us to be on the safe side when calculating the new utilization imbalance, and truly decrease it with every move or partition.

Algorithm 2 Reduce the utilization imbalance between two sockets

```
    struct InfoSocket:
        recentCpu;  // recent CPU utilization
        recentMem;  // recent memory throughput utilization
        infoRUH[];  // InfoRUH of TBP and TGP placed on the socket
 1: function REDUCEIMBALANCE(InfoSocket S1, InfoSocket S2, strategy)
 2:     S1 ← hotter socket of the two, according to recentCpu
 3:     S2 ← colder socket of the two, according to recentCpu
 4:     maxCpu ← H/W threads of a socket
 5:     maxMem ← Maximum memory throughput of a socket
 6:     imb ← S1.recentCpu - S2.recentCpu
 7:     if S1.recentCpu > c_s * maxCpu and imb > c_i * maxCpu then
 8:         for all S1.infoRUH do
 9:             if strategy = move and infoRUH.canMove then
10:                 S1'cpu ← S1.recentCpu - infoRUH.recentCpu
11:                 S2'cpu ← min(S2.recentCpu + infoRUH.recentCpu, maxCpu)
12:                 imb' ← | S1'cpu - S2'cpu |
13:                 S2'mem ← S2.recentMem + infoRUH.recentMem
14:                 if imb' < imb and S2'mem ≤ maxMem then
15:                     Move TBP or TGP to S2
16:                     return true
17:             else if strategy = partition and infoRUH.canPartition then
18:                 S2freeCpu ← maxCpu - S2.recentCpu
19:                 halfRUH ← min(infoRUH.recentCpu / 2, S2freeCpu)
20:                 S1'cpu ← S1.recentCpu - infoRUH.recentCpu + halfRUH
21:                 S2'cpu ← S2.recentCpu + halfRUH
22:                 imb' ← | S1'cpu - S2'cpu |
23:                 S2'mem ← S2.recentMem + (infoRUH.recentMem / 2)
24:                 if imb' < imb and S2'mem ≤ maxMem then
25:                     Partition table or TG into 2 · current partitions,
                        and move all new partitions to their original sockets,
                        apart from one partition of S1 which is moved to S2
26:                     return true
27:     return false
```

Let us denote the utilization imbalance between two sockets at some timestamp t_n as $imb(t_n)$. If our algorithm does nothing, the imbalance stays the same. If our algorithm moves or partitions a RUH's owner, the imbalance decreases: $imb(t_{n+1}) \leq imb(t_n)$. The sequence $imb(t_n)$ is monotonically decreasing with a lower bound of 0. According to the monotone convergence theorem, the sequence will converge. In our case, since we limit partitioning to a number of partitions capped by the number of sockets, the imbalance may converge to a non-zero value. Also, we set a lower threshold for the imbalance (see Table 1), below which DP does nothing. We note that even if tables increase their utilization after the new placement, the resulting imbalance cannot exceed the previous one.

The algorithm for reducing the utilization imbalance of a pair of sockets is presented in Algorithm 2. As mentioned, DP at every period calculates a snapshot of the recent utilization of the system's RUH by creating InfoRUH objects. It also aggregates the utilization of every InfoRUH to calculate the snapshot of the recent utilization of every socket. For every socket, this information is stored in an ***InfoSocket*** object, which is defined in Algorithm 2 as well.

Function `reduceImbalance` receives two InfoSocket objects and a strategy (move or partition). First, it discerns which socket is the hotter one and which is the colder one (lines 2–3). It then gets the cores and maximum memory throughput of a socket (lines 4–5). These are calculated once on a machine (see Section 8 and Table 2). The current imbalance is calculated (line 6). If the hot socket's utilization is over our saturation threshold and the imbalance is over our threshold (see Table 1), the function proceeds (line 7). It iterates the RUH of the hot socket (line 8), and examines whether the utilization imbalance can be reduced by either moving (lines 9–16) or partitioning an eligible RUH's owner (lines 17–26). Conceptual examples of the calculations are shown in Figure 8.

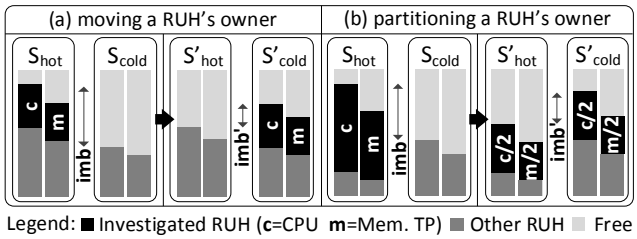

Figure 8: Calculating the utilization imbalance of a pair of sockets before and after (a) moving, and (b) partitioning.

In case of moving, the sockets' new CPU utilization is calculated (lines 10–11). For the cold socket, we cap the CPU utilization by the socket's cores (line 11). The new CPU imbalance is calculated (line 12). The new memory bandwidth of the cold socket is calculated (line 13), without capping it. If the new CPU imbalance is less than the original, and we do not create a new memory bandwidth bottleneck on the cold socket (line 14), we move the corresponding TBP or TGP to the cold socket (line 15), and return true (line 16). We note that we move all TBP of a TGP concurrently.

In case of partitioning, we calculate the cold socket's free threads (line 18). Each new partition's CPU utilization is half of the original utilization, capped by the cold socket's free threads (line 19). Then, we calculate the sockets' new CPU utilization (lines 20–21) and imbalance (line 22). The cold socket's new memory bandwidth is calculated (line 23), without capping it. If the new CPU imbalance is less than the original, and we do not create a new memory bandwidth bottleneck on the cold socket (line 24), we partition the RUH's owner (line 25), and return true (line 26). We note that we partition all tables of a TG concurrently. After partitioning, we move concurrently all new TBP or TGP to their original sockets, apart from one of the hot socket which is moved to the cold socket.

6.4 Data Placer

Algorithm 3 implements DP. After waiting for a period (line 2), DP goes through all TBP and TGP (line 3). For every one, it calculates its InfoRUH (line 5), and if it is active, DP adds it to the appropriate InfoSocket, aggregating its recent utilization as well to the socket (line 7). At this point, we have a snapshot of the recent utilization in the past period. DP then sorts the RUH of every socket by their recent CPU utilization (line 8). This makes intensely used RUH to be considered first for moving or partitioning.

DP then calculates the imbalance of every pair of sockets (lines 9–11). The pairs of sockets are sorted by their imbalance (line 12), so that we first examine the pair with the greatest imbalance. For all pairs (line 13), we try to reduce their imbalance by moving a RUH's owner (line 14). If nothing is moved, we try again to reduce their imbalance by partitioning a RUH's owner (lines 16–18). If DP moves or partitions a RUH's owner, it goes back to waiting.

If DP does not move or partition a RUH's owner, it attempts to merge cold partitioned data. It iterates through all partitioned tables and TG in the system catalog (line 19). DP checks whether a table or TG is cold by looking into its past utilization. The amount of time to look in the past is calculated (lines 20–22) as in Algorithm 1 for the case of partitioning, with the difference of summing the pages of all RUH (since merging creates a single partition). If the average past CPU and memory TP utilization of all RUH is zero (line 23), a background request is initiated (line 24) which merges the table, or all involved tables in case of a TG, and moves it to the coldest socket. We use a `set` internally to keep track of tables or TG undergoing merging in order to consider them ineligible for moving or partitioning until their merging completes.

Algorithm 3 Data Placer

```
1: while true do
2:   wait c_p
3:   InfoSocket[] ← initialize an InfoSocket object for every socket
4:   for all loaded TBP and TGP do
5:     InfoRUH ← CALCULATEINFORUH(RUH)
6:     if InfoRUH.isActive then
7:       Add and aggregate InfoRUH to the appropriate InfoSocket
8:   Sort the InfoRUH in every InfoSocket by their recentCpu
9:   list<tuple<InfoSocket, InfoSocket, imbalance>> ← empty list
10:  for all pairs of InfoSocket do
11:    Add new tuple(S1, S2, | S1.recentCpu - S2.recentCpu |) to list
12:  Sort list by imbalance
13:  for all pairs of InfoSocket in the sorted list do
14:    if REDUCEIMBALANCE(S1, S2, move) then
15:      goto line 2
16:  for all pairs of InfoSocket in the sorted list do
17:    if REDUCEIMBALANCE(S1, S2, partition) then
18:      goto line 2
19:  for all partitioned tables and TG in the catalog do
20:    pages ← sum pages of all RUH of table or TG
21:    usMove ← (pages * speed of moving) * 2
22:    usPartition ← usMove + (pages * speed of partitioning)
23:    if all RUH have 0 average utilization during last usPartition then
24:      Merge and move to the coldest socket in the background
```

7. ADAPTIVE TASK STEALING

As we showed in Section 4, stealing memory-intensive tasks can hurt overall performance. Here, we pinpoint the switching point when tasks become memory-intensive enough that they should not be stolen across sockets. The switching point depends on the hardware and the implementation. For this reason, we propose a calibration experiment that the DBMS can run once on a server to find the switching point. In order to fully control the memory intensity of tasks, we avoid the SQL layer of the DBMS, and use immediately the task scheduler on simple data structures.

Calibration experiment. We place four 4 GB vectors of randomly generated `doubles` on each of the half sockets of the server. Thus, half of the sockets can have local accesses, while the rest will either steal remote tasks or stay idle. The workload consists of one client thread per vector. Each client continuously issues a query that sums the elements of its corresponding vector. We measure the total throughput (TP). The client parallelizes the query with a number of tasks equal to the number of hardware threads in the socket. The tasks are given equi-sized ranges of the vector to sum. The reason we use four vectors per socket, and one client thread per vector, is to have enough tasks to saturate the local socket and more tasks that can potentially be stolen by another socket (that does not have data).

In order to control the memory intensity of the summation, we raise each element of the vector to a varying power `n`: $sum = v_1^n + v_2^n + v_3^n \ldots$ We implement each task's summation using a for loop to raise its element to the desired power `n`. As we increase `n`, we increase the time spent in the for loop, thus increasing the CPU intensity.

Figure 9 shows the results of the calibration experiment for the three servers of our experiments (see Section 8). For all servers, disallowing stealing results in the best TP for lower values of `n`, since the summation is more memory-intensive. This is also shown by the system's memory TP, which almost saturates the memory bandwidth of half the sockets. It is also shown by the average memory TP of the task class (tasks belong to a single class in this experiment).

We note that disallowing stealing only achieves a 50% CPU load, since half of the sockets have data. But it is better for memory-intensive workloads than allowing stealing, achieving up to 4x better TP (on the 32-socket server). Stealing has 100% CPU load, but

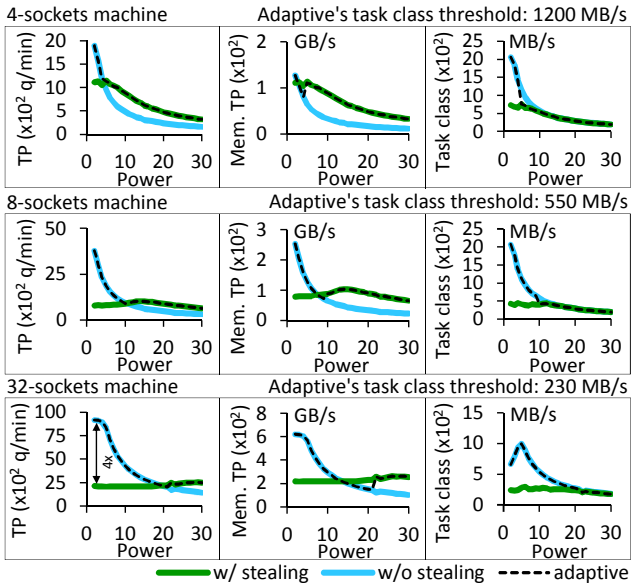

Figure 9: Calibration experiment for three NUMA servers.

saturates the interconnect network and overwhelms the already saturated remote memory controllers. The overwhelmed memory controllers achieve much lower overall memory TP (for both local and remote tasks) than the case of disallowing stealing.

As n increases, the workload becomes more CPU-intensive, and the memory TP of the system and the task class finally starts decreasing. In terms of CPI (cycles per instruction), e.g., on the 4-socket server for the case when stealing is disabled, it starts at 0.83 and gradually increases up to 1.25 (for n = 30). At a switching point, stealing becomes better, and remote tasks can be satisfied through the interconnects and the remote memory controllers sufficiently.

At the switching point, we mark the memory TP of the task class as the threshold for stealing vs. not stealing. Our adaptive task stealing uses this threshold at run-time. If the exponential average of a task class becomes higher than the threshold, stealing is disallowed for tasks of this task class. If the exponential average becomes lower than the threshold, stealing is allowed.

Figure 9 shows the threshold pinpointed for each server, and also shows the adaptive task stealing that uses this threshold. The adaptive line achieves the best throughput for all cases of power n, successfully allowing stealing at the switching point.

Finally, we note that the calibration experiment can be further extended to specialize the threshold for more cases of: different number of sockets having data, different CPU utilization per socket, etc. Our current calibration experiment is sufficient for our use cases and experiments, as it roughly finds out the switching point for the "middle" case where half of the server's sockets have active data.

8. EXPERIMENTAL EVALUATION

We first present our experimental configuration. Then, we present results of a custom benchmark and finally of the TPC-H benchmark.

Experimental configuration. We use a prototype built on SAP HANA (SPS11), a commercial main-memory column-store, with our NUMA-aware task scheduler and scans [28]. We add support for more NUMA-aware operators (see Section 4), track resource utilization (see Section 5), and employ our adaptive NUMA-aware data placement (see Section 6) and task stealing (see Section 7).

For all experiments, we warm up the DBMS, we admit all clients and disable result caching. LLC misses, CPU load, and memory TP are gathered from Linux and H/W counters (integrating Intel PCM [9]). The utilizations of RUH are the local-only utilizations we track (via the same H/W counters). The imbalance metric corresponds to the maximum imbalance between any two sockets, when calculated by DP. The imbalance fluctuates since the averaged sampled utilizations of RUH also fluctuate, but in general is decreased with a stable workload. Results shown with a timeline consist of a single run, while any data points (in previous sections as well) are averages of at least three iterations with a standard deviation <10%.

Table 2 shows characteristics of the servers we use. The first is the one of Figure 2a. The second is an 8-socket server. The third is a rack-scale 32-socket SGI UV 300H server. NUMA characteristics, such as local and inter-socket idle latencies and peak memory bandwidths, are measured with Intel Memory Latency Checker [34].

Table 2: Characteristics of the three NUMA servers we use.

Statistic \ Machine	4x15-core Intel Xeon E7-4880v2 at 2.50GHz	8x15-core Intel Xeon E7-8880v2 at 2.50GHz	32x18-core Intel Xeon E7-8890v3 at 2.50GHz
Memory per socket	128 GB	128 GB	512 GB
Local latency	108 ns	110 ns	120 ns
1 hop latency	170 ns	320 ns	320 ns
Max hops latency	170 ns	390 ns	590 ns
Local B/W	70 GB/s	70 GB/s	45.5 GB/s
1 hop B/W	12.5 GB/s	10.5 GB/s	15 GB/s
Max hops B/W	12.5 GB/s	9.5 GB/s	7.3 GB/s
Total local B/W	280 GB/s	570 GB/s	1363 GB/s
Stealing threshold	1200 MB/s	550 MB/s	230 MB/s
Move us/page	59	63	60
Partition us/page	109	123	129

Custom benchmark. Our dataset has 64 tables (`TBL1-64`). For each table we generate a CSV file of 50 million rows, around 3.2 GB, for a total of 204 GB files. Each table has an `ID` integer column (PK), 8 additional columns (`COL1-8`) of random integers (uniform distribution), and a partitioning specification (hash) on `ID`. The 8 columns have bitcases 17 to 24, so as to have different number of unique values. Each experiment mentions the initial table placement.

The workload is generated with a Java application on a different server. Clients continuously issue queries and we measure the total throughput (TP). At each experiment, we mention how many clients are used, which query type(s) they issue, which table(s) they target, and which selectivity they use. The possible query types are:

(a) `SELECT COLx FROM TBLy WHERE COLx >= ? AND COLx <= ?`. The client selects a random column from its target table. The query involves both scan phases mentioned in Section 4.

(b) `SELECT COL1, SUM(TO_DOUBLE(COLx)) FROM TBLy WHERE COLx >= ? AND COLx <= ? GROUP BY COL1`. The client selects a random column (`COL2-8`) from its target table to aggregate and group-by `COL1`. This query involves the aggregation phases mentioned in Section 4. We choose `COL1` for the group-by because it has the least number of unique values. We cast to double to avoid potential numeric overflow errors.

(c) `SELECT TBLz.COLx FROM TBLy, TBLz WHERE TBLy.ID = TBLz.ID AND TBLy.COLx >= ? AND TBLy.COLx <= ?`. The client joins two target tables on the `ID` column. A random column is selected to filter and project. This query involves the equi-join steps mentioned in Section 4.

Before each experiment begins, we let clients build a prepared statement for each query they can issue. There are no thinking times. The clients do not fetch results, in order to not let the network transfer dominate. Each TP value in a timeline corresponds to the slope of the achieved queries during the previous 30 seconds.

8.1 Adaptive Data Placement

The first experiment realizes the introductory example of Figure 1. TBL1 and TBL2 are placed on socket S1, TBL3 on S2, and TBL4 is partitioned across S3 and S4. Each of the tables TBL1–3 are targeted by 64 clients executing query (a) with a low selectivity (0.001%) for 5 minutes. Figure 10 shows the timelines of the throughput (TP), the utilization imbalance, and additional performance measurements that include H/W counters as well as our tracked utilization (RUH).

At the beginning, only S1 and S2 execute queries as shown by their RUH. Queries are dominated by the scan's first phase ("IV-Scan"). Tasks are memory-intensive as shown by the task class's memory TP, which is over the stealing threshold. That is why adaptive stealing disallows stealing, and most LLC misses are local. As shown by the tables' RUH, TBL1 and TBL2 share S1, while TBL3 fully utilizes S2.

DP recognizes the imbalance, but does not take action because the TBP are not yet eligible to be moved or partitioned. DP searches the catalog to find TBL4 which is partitioned and cold (thus not shown in Figure 10), and at 16 s starts a background request to merge it. The merge finishes at 64 s, and the single TBP is moved to S4 (a cold socket) at 106 s. The merge and move contribute to the small bump in the CPU load and memory TP of S3 and S4. Another reason for their increased CPU load is that their worker threads attempt to steal tasks from other sockets, but tasks are memory-intensive and cannot be stolen in this experiment. Since S3 and S4 do not process any queries, we do not account this busy CPU load in their RUH.

At 53 s (see markers on the timeline of the tables' RUH graphs), DP examines the pair of S1 and S3. It decides to fix their imbalance by moving TBL2, which has become eligible for moving, to S3. The move completes at 91 s. Overall TP and memory TP are increased.

Next, DP detects that there is still a utilization imbalance because 3 sockets are utilized and S4 is not (as shown by its RUH). At 108 s, DP examines the pair of S2 and S4. It decides to fix their imbalance by partitioning TBL3, which is eligible for partitioning, into two parts. At 174 s, partitioning completes, and the two TBP are moved to S2 and S4 concurrently, which completes at 190 s.

After that point, the imbalance is decreased within our threshold, and there are no more actions. In comparison to the beginning of the experiment, overall memory TP is 2x more, and TP is also 2x more.

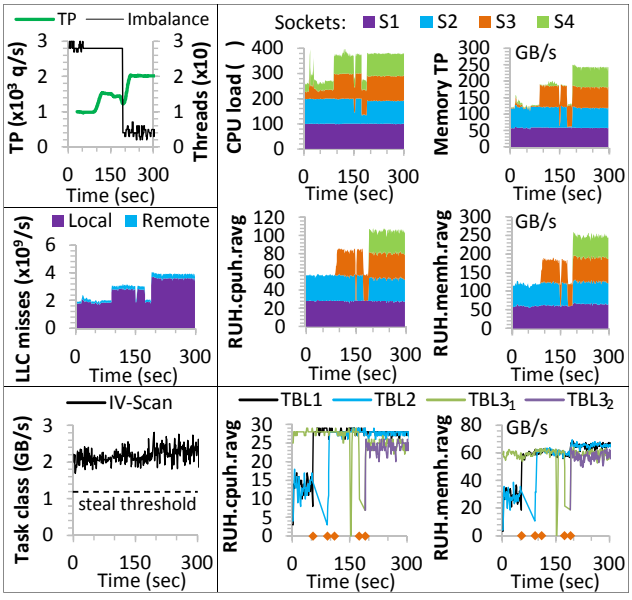

Figure 10: Introductory experiment showing the adaptive data placement of three active tables (4-socket server).

8.2 Adaptive Task Stealing

To show the effect of adaptive task stealing, we use scans of varying selectivity. We place TBL1 on S2 and TBL2 on S4. Adaptive placement is disabled. Each of the tables is targeted by 256 clients executing query (a) with the specified selectivity. Half of the sockets have local tasks, while the other half would need to steal. For each selectivity, we execute 5 min runs of: enabled stealing for all tasks, disabled stealing for all tasks, and adaptive stealing. We report each run's average TP. The results are shown in Figure 11.

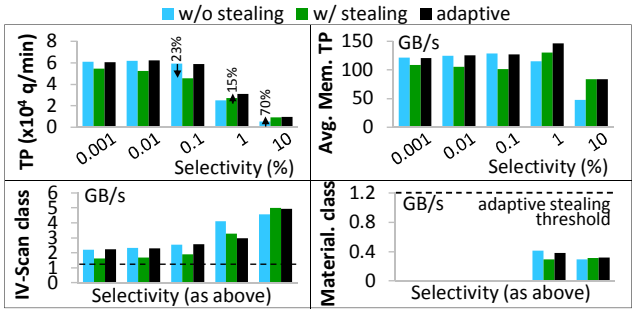

Figure 11: Experiment showing how adaptive task stealing disallows stealing of memory-intensive classes (4-socket server).

For low selectivities, the scan's first phase ("IV-scan") dominates. Tasks are memory-intensive and stealing hurts TP by up to 23% for the case of 0.1% selectivity. As selectivity increases, the scan's second phase (materialization) dominates and is parallelized. The fewer IV-scan tasks can utilize more memory bandwidth on their socket. The dominating materialization tasks are CPU-intensive, due to their random accesses to the dictionary, and thus stealing helps improve TP by up to 70% for the case of 10% selectivity. Adaptive stealing achieves the best TP of either stealing or not stealing in all cases of selectivity. It can also, e.g., for the case of 1% selectivity, further improve performance by 15%. This is due to disallowing stealing of IV-scan tasks, and allowing stealing of materialization tasks, instead of taking a static strategy for all task classes.

8.3 Partitioning Overhead

Here, we show how our adaptive data placement can avoid the overhead of unnecessary partitioning. We focus on aggregations, but the implications of unnecessary partitioning are similar for joins (as in Section 4). We initially partition TBL1–8 across all sockets of the 8-socket server. The experiment has three consecutive 5min phases. In the first, each table is targeted by 8 clients executing query (b) with a high selectivity (10%). The second phase has no activity. The third phase is the same as the first. Figure 12 shows the results.

During the second phase, all tables are merged. During the third phase, DP moves tables that happened to be merged on the same socket to balance utilization. TP reaches 1.7x of the TP of the first phase, because there is no partitioning overhead, and the server can be saturated with non-partitioned tables. This is also shown by the improved memory TP and local LLC misses.

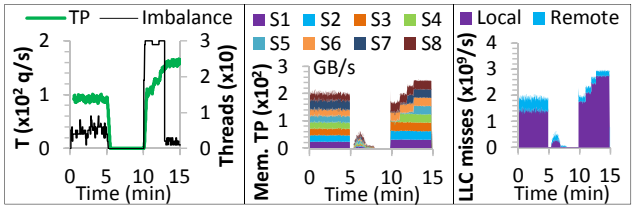

Figure 12: The overhead of partitioning (8-socket server).

Impact of groups. Next, we detail how the partitioning overhead for aggregations increases as the number of groups increases. We use TBL1-8 either partitioned (8 TBP/table) or non-partitioned (1 TBP/table), placed round-robin across the sockets. Adaptive placement is disabled. Each table is targeted by 8 clients issuing a variation of query (b) that selects COL8 with a high selectivity (10%), and groups-by a different column. As we group-by COL1 through COL7, the bitcase of the group-by column increases, and so the number of groups increases. Figure 13 shows the results. Each run is 5 minutes. We also include a case without a group-by, and show the percentage of remote LLC misses out of all LLC misses.

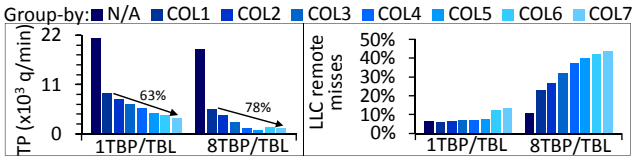

Figure 13: The impact of the number of groups (8-socket server)

For the case without grouping, both data placements perform similarly, with mostly local accesses. As the number of groups increases, TP drops since we need to work on more hash tables. For 1 TBP/table, merges happen locally to each socket, while for 8 TBP/table there is the the cost of the global dictionary and merges may need to access hash tables on 7 other sockets. For this reason, the drop in TP from COL1 to COL7 for 8 TBP/table is worse (78% drop) than for 1 TBP/table (63% drop). This is also reflected in the percentage of LLC remote misses. For several group-by cases, the TP of 1 TBP/table is more than 2x than the TP of 8 TBP/table.

8.4 Changing Workload

Here, we show a workload with three consecutive phases. We place 8 tables on each socket of the 8-socket server (all 64 tables are placed). Clients execute query (a) with low selectivity (0.001%). The first phase lasts 15 min, and only TBL56 (on S7) and TBL64 (on S8) are targeted by 512 clients each. The second phase lasts 5 min, and all 64 tables are targeted by 16 clients each. The third phase lasts 10 min, and only TBL1-4 (on S1) and TBL9-12 (on S2) are targeted by 128 clients each. Figure 14 shows the results.

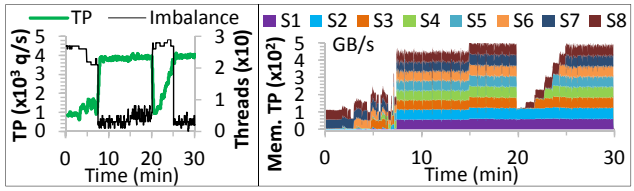

Figure 14: Three different workload phases (8-socket server).

During the first phase, DP gradually partitions the hot tables to fill all sockets and reach maximum TP. During the second phase, all tables become hot, TP stays at the maximum, and DP takes no actions. At the third phase, only two sockets are used, and DP gradually moves their hot tables to fill all sockets and reach maximum TP.

8.5 Workload Mix

Here, we have an initial complex placement, a stable workload mix, and show that DP gradually reaches a final stable state. We use TBL1-7, initially placed on 4 sockets of the 8-socket machine as shown in Figure 15. We use 128 clients, continuously issuing a random query out of the queries shown in Figure 15. We define a TG for TBL1-2 and another TG for TBL3-5, thus the queries' joins between these tables are copartitioned. Figure 15 shows the results.

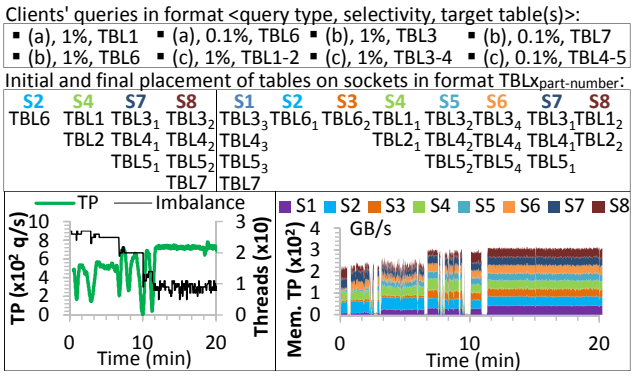

Figure 15: Balancing the utilization of a stable workload mix.

Initially, 4 sockets have memory TP. We note that the system CPU load is saturated, as there are CPU-intensive tasks that are stolen. DP moves and partitions tables to reach the placement shown in Figure 15 with a balanced utilization. All sockets finally have memory TP and mostly local accesses, improving TP by 44% vs. the initial TP. Since in this experiment all clients issue all queries, there are drops in TP while DP holds an exclusive lock for moving a table.

8.6 TPC-H Benchmark

Here, we show the TPC-H [3] benchmark with a scaling factor 30 (30 GB flat files). We use 512 clients, each continuously issuing a random query out of the 22 templates. The tables are initially placed on a single socket of the 32-socket server. We define a partitioning specification for all tables except nations and regions, and a TG for lineitems and orders so that they are copartitioned (on the orderkey columns). Figure 16 shows the results, which are normalized, due to legal reasons, with undisclosed constants, to the maximum observed values. This does not hinder us from showing DP's impact.

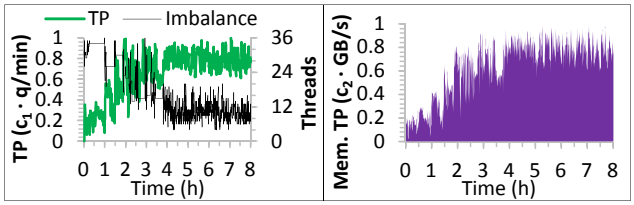

Figure 16: TPC-H throughput run (32-socket server).

Initially, only one socket has memory TP. We note that more sockets have CPU load, as there are CPU-intensive tasks that can be stolen. TP is only around 0.2. DP gradually moves and repartitions tables and the TG to balance utilization. Finally, all sockets have data, the TG of lineitems and orders has 32 partitions, customers have 4 partitions, partsupp has 8 partitions, while the remaining tables are not partitioned. All sockets have memory TP and mostly local accesses. TP reaches around 0.8. Assuming the initial TP corresponds to the typical case when an administrator simply loads the dataset, our adaptive data placement helps improve TP by 4x.

9. DISCUSSION AND OUTLOOK

Associated tables. We use predefined TG for handling multiple-input operators at run-time, focusing on copartitioned equi-joins. A further opportunity is an automated way of recognizing associated tables, the dominant join predicates, and forming TG, by tracking the workload at run-time. This requires the challenging adaptation of an offline solver for distributed data placement (see Section 3) or specially for copartitioned joins [38], for execution at run-time.

Priorities and fairness. Priorities and fairness are an orthogonal issue out of this paper's scope. We note that our task scheduler supports priorities and a degree of fairness (based on query submission time) [28]. In typical cases, if a workload has user-defined priorities, the prioritized tasks will highly contribute to the utilization of RUH which will be considered first by DP for moving or partitioning.

Task classes. Tasks in the same class should have similar memory throughput. We assume that classes are defined manually, as we do in Section 5 for our NUMA-aware operators. One can further specialize classes, e.g., by the involved predicates. As seen in Section 7, an aggregation's memory throughput can vary depending on the complexity of the predicate. Ideally, we need a way to classify complex predicates. This is left out of the paper's scope. For our implementation and experiments, we use rather typical predicates and the defined task classes can sufficiently capture the memory intensity of our NUMA-aware operators' different phases.

Unit of data placement. Our unit of data placement is a row-wise partition of a table. An alternative would be column-wise partitioning, i.e., placing a table's columns on different sockets. In such a case, a global dictionary (see Section 4) is not needed, but queries referencing columns on multiple sockets incur a lot of remote accesses. For this work, we assume that the organization of associated columns into tables is left to the administrator.

Balancing memory throughput. We balance primarily the CPU utilization under the constraint of not creating memory bandwidth bottlenecks. This is to allow newly placed data to potentially increase their utilization. Since we balance local-only CPU utilization, this can indirectly balance memory TP as well as shown in many of our experiments. This is not guaranteed, however. One may wish DP to continue balancing memory TP after CPU utilization is balanced, under the constraint of not increasing the CPU imbalance. DP's possible actions can be extended to consider exchanging TBP or TGP between sockets. We have found only a few cases where balancing memory TP is required to slightly improve IPC and TP.

10. CONCLUSIONS

In this paper, we show that main-memory column-stores should not employ a static strategy of always partitioning data across all sockets, and always allowing inter-socket task stealing. We show that unnecessary partitioning involves an overhead of up to 2x in comparison to not partitioning. For this reason, we develop an adaptive data placement algorithm that can track a utilization imbalance across sockets, and can move or repartition tables at run-time to fix the imbalance. Also, we show that inter-socket stealing of memory-intensive tasks can hurt throughput by up to 4x in comparison to not stealing. For this reason, we develop an adaptive technique that disallows stealing at run-time for tasks whose memory intensity exceeds a fixed threshold for a NUMA server.

Acknowledgements. This project has received funding from SAP SE, Walldorf, Germany. We thank the members of the SAP HANA team for their support and feedback.

11. REFERENCES

[1] SAP HANA Platform SPS 11 Administration Guide, Dec. 2015. http://help.sap.com/hana_platform.
[2] SAP HANA Data Distribution Optimizer Administration Guide, Mar. 2016. http://help.sap.com/hana_options_dwf.
[3] TPC Benchmark H Rev. 2.17.1, 2016. http://www.tpc.org/.
[4] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, pp. 1110–1121, 2004.
[5] M. Albutiu et al. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*, 5(10):1064–1075, 2012.
[6] C. Balkesen et al. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1):85–96, 2013.
[7] S. Blagodurov et al. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIX*, 2011.
[8] M. Dashti et al. Traffic management: A holistic approach to memory placement on NUMA systems. In *ASPLOS*, pp. 381–394, 2013.
[9] R. Dementiev et al. Intel Performance Counter Monitor, Mar. 2016. https://software.intel.com/articles/intel-performance-counter-monitor.
[10] K. P. Eswaran. Placement of records in a file and file allocation in a computer network. In *Information Processing*, pp. 304–307, 1974.
[11] F. Färber et al. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.
[12] J. Giceva et al. Deployment of Query Plans on Multicores. *PVLDB*, 8(3):233–244, 2014.
[13] L. Golab et al. Distributed data placement to minimize communication costs via graph partitioning. In *SSDBM*, pp. 20:1–20:12, 2014.
[14] T. Gubner. Achieving many-core scalability in Vectorwise. 2014. Master's thesis. TU Ilmenau.
[15] G. Hill and A. Ross. Reducing outer joins. *The VLDB Journal*, 18(3):599–610, Aug. 2008.
[16] T. Kissinger et al. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workloads. *ADMS*, pp. 74–85, 2014.
[17] C. Lameter et al. NUMA (Non-Uniform Memory Access): An Overview. *ACM Queue*, 11(7):40:40–40:51, 2013.
[18] H. Lang et al. Massively Parallel NUMA-aware Hash Joins. In *IMDM*, pp. 1–12, 2013.
[19] P.-A. Larson et al. Enhancements to SQL server column stores. In *SIGMOD*, pp. 1159–1168, 2013.
[20] V. Leis et al. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*, pp. 743–754, 2014.
[21] C. Lemke et al. Speeding up queries in column stores: a case for compression. In *DaWaK*, pp. 117–129, 2010.
[22] Y. Li et al. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.
[23] N. Mukherjee et al. Distributed Architecture of Oracle Database In-memory. In *PVLDB*, volume 8, pp. 1630–1641, 2015.
[24] I. Müller et al. Cache-Efficient Aggregation: Hashing Is Sorting. In *SIGMOD*, pp. 1123–1136, 2015.
[25] M. T. Özsu et al. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.
[26] D. Porobic et al. ATraPos: Adaptive transaction processing on hardware Islands. In *ICDE*, pp. 688–699, 2014.
[27] I. Psaroudakis et al. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *ADMS*, pp. 36–45, 2013.
[28] I. Psaroudakis et al. Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *PVLDB*, 8(12):1442–1453, 2015.
[29] I. Psaroudakis et al. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *TPCTC*, pp. 97–112, 2015.
[30] V. Raman et al. DB2 with BLU Acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.
[31] J. Rao et al. Automating Physical Database Design in a Parallel Database. In *SIGMOD*, pp. 558–569, 2002.
[32] O. Steinau et al. Method for calculating distributed joins in main memory with minimal communicaton overhead. US Patent App. 11/018,697.
[33] F. Transier et al. Aggregation in parallel computation environments with shared memory, 2012. US Patent App. 12/978,194.
[34] V. Viswanathan et al. Intel Memory Latency Checker v3.0, Mar. 2016. https://software.intel.com/articles/intelr-memory-latency-checker.
[35] M. Wagle et al. NUMA-Aware Memory Management with In-Memory Databases. In *TPCTC*, pp. 45–60, 2015.
[36] T. Willhalm et al. Vectorizing database column scans with complex predicates. In *ADMS*, pp. 1–12, 2013.
[37] Y. Ye et al. Scalable aggregation on multicore processors. *DaMoN*, 2011.
[38] E. Zamanian et al. Locality-aware Partitioning in Parallel Database Systems. In *SIGMOD*, pp. 17–30, 2015.
[39] M. Zukowski et al. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.