Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement

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

Main-memory column-stores are called to efficiently use modern non-uniform memory access (NUMA) architectures to service concurrent clients on big data. The efficient usage of NUMA architectures depends on the data placement and scheduling strategy of the column-store. Most column-stores choose a static strategy that involves partitioning all data across the NUMA architecture, and employing a stealing-based task scheduler. In this paper, we implement different strategies for data placement and task scheduling for the case of concurrent scans. We compare these strategies with an extensive sensitivity analysis. Our most significant findings include that unnecessary partitioning can hurt throughput by up to 70%, and that stealing memory-intensive tasks can hurt throughput by up to 58%. Based on our analysis, we envision a design that adapts the data placement and task scheduling strategy to the workload.

1. INTRODUCTION

Contemporary analytical workloads are characterized by massive data and high concurrency, with hundreds of clients [31]. The key comparative criterion for analytical relational database management systems (DBMS) is their efficiency in servicing numerous clients on big data. For this reason, many analytical DBMS, e.g., SAP HANA [14] or Oracle [18] (see Section 3 for additional examples), opt for a main-memory column-store instead of a disk-based row-store typically employed for OLTP workloads [32]. The column-wise in-memory representation minimizes the amount of data to read, as only the necessary columns are accessed. Data is typically compressed, e.g., using dictionary encoding, and processed in parallel using SIMD and multiple CPU cores [14].

Main-memory column-stores need to efficiently exploit the increasing amount of DRAM and multi-core processors. Processor vendors are scaling up hardware by interconnecting sockets of multi-core processors, each with its own memory controller and memory [5]. Memory is decentralized, forming a non-uniform memory access (NUMA) architecture.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

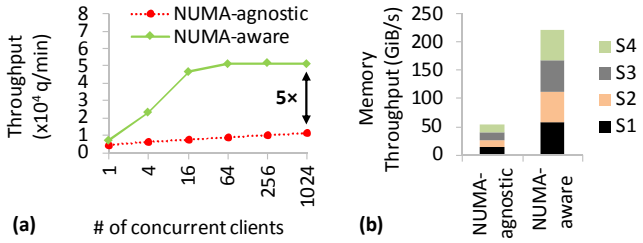

Figure 1: (a) Impact of NUMA. (b) Memory throughput of the sockets for the case of 1024 clients.

NUMA introduces new performance challenges, as communication costs vary across sockets [10, 27], and the bandwidth of the interconnect links is an additional bottleneck to be considered (see Section 2). The column-store needs to become NUMA-aware by handling the placement of its data structures across sockets, and scheduling the execution of queries onto the sockets. Figure 1a shows the performance difference between a NUMA-agnostic and a NUMA-aware column-store as they evaluate an increasing number of analytical clients on a machine with four sockets (see Section 6 for more details). In this scenario, NUMA-awareness significantly improves throughput, by up to 5×. By avoiding inter-socket communication, the memory bandwidth of the sockets can be fully utilized, as shown in Figure 1b.

In the literature, there has been a recent wave of related work for NUMA-aware analytical DBMS. Their majority employs a static strategy for data placement and scheduling. For example, HyPer [23] and ERIS [17] partition all data across sockets and parallelize queries with a task scheduler. The task scheduler consists of a pool of worker threads. Each worker processes local tasks or steals tasks from other workers. The trade-offs between different data placement and task scheduling strategies have not been yet fully analyzed.

Contributions. In this paper, we describe and implement data placement and task scheduling strategies for concurrent scans in main-memory column-stores. Through a sensitivity analysis, based on a commercial column-store (SAP HANA), we identify the trade-offs for each strategy under various workload parameters. Our main contributions are:

- We present a novel partitioning scheme for dictionary-encoded columns, supporting quick repartitioning, for skewed scan-intensive workloads (see Section 4).
- We show that unnecessary partitioning for highly concurrent memory-intensive workloads can hurt through-

put by up to 70% in comparison to not partitioning (see Section 6.1). Hot data in skewed workloads should be partitioned until socket utilization is balanced.

- We show that stealing memory-intensive tasks can hurt throughput by up to 58% (see Section 6.2). The task scheduler needs to support tasks that can prevent being stolen by another socket (see Section 5).
- Based on the implications of our sensitivity analysis, we envision a design that can adapt the task scheduling and data placement (by moving and partitioning hot data) strategy to the workload at hand (see Section 7).

2. BACKGROUND

In this section, we give a quick overview of different NUMA architectures (which we use in our experiments), and memory allocation facilities in the operating system (OS).

NUMA architectures. Figure 2 shows a NUMA server with 4 sockets of 15-core Intel Xeon E7-4880 v2 2.50GHz (Ivybridge-EX) processors. Each core has two hardware threads, a 32KiB L1, and a 256 KiB L2 cache. The cores in a socket share a 37.5MB L3 cache. Each socket shares 2 memory controllers (MC) [1], configured in "independent" mode for the highest throughput [2]. Each MC supports 2 Intel SMI interfaces [2]. Each SMI supports 2 memory channels (buses), with up to 3 DDR3 DIMM attached on each channel. Our configuration has one 16GiB DDR3 1600MHz DIMM per channel. The sockets are interconnected to enable accessing remote memory of another socket. Each socket has 3 Intel QPI (QuickPath Interconnect). Each QPI has a 16GiB/s bandwidth that supports data requests and the cache coherence protocol [1]. The majority of NUMA architectures today are cache coherent (ccNUMA). The interconnect topology, the interconnect protocol, and the cache coherence protocol are specific to each system and vendor.

NUMA introduces new considerations for software performance, in comparison to UMA [10]: (a) accesses to remote memory are slower than local memory, (b) the bandwidth of a MC can be separately saturated, and (c) the bandwidth of an interconnect can be separately saturated. To understand how these factors vary, we show in Table 1 local and inter-socket latencies, and peak memory bandwidths for 3 machines (measured with Intel Memory Latency Checker v2).

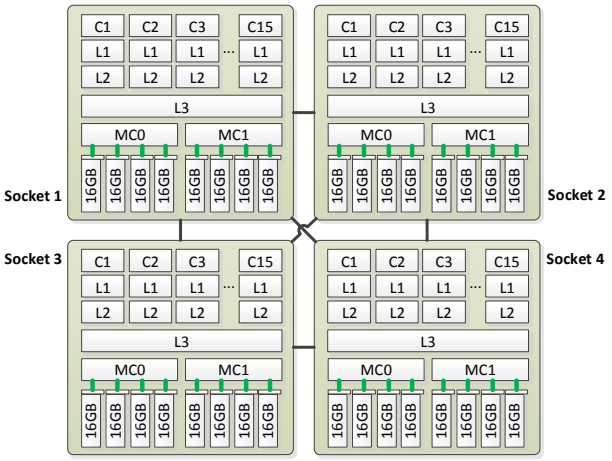

Figure 2: 4-socket server with Ivybridge-EX CPU.

Table 1: Local and inter-socket idle latencies, and peak memory bandwidths of three different servers.

Statistic	4×Ivybridge-EX	32×Ivybridge-EX	8×Westmere-EX
Local latency	150 ns	112 ns	163 ns
1 hop latency	240 ns	193 ns	195 ns
Max hops latency	240 ns	500 ns	245 ns
Local B/W	65 GiB/s	47.5 GiB/s	19.3 GiB/s
1 hop B/W	8.8 GiB/s	11.8 GiB/s	10.3 GiB/s
Max hops B/W	8.8 GiB/s	9.8 GiB/s	4.6 GiB/s
Total local B/W	260 GiB/s	1530 GiB/s	96.2 GiB/s

The first is the one of Figure 2, the second is a rack-scale SGI UV 300 system with 32 sockets of Intel Xeon E7-8890 v2 2.80GHz (Ivybridge-EX) processors, and the third consists of 2 IBM x3950 X5 boxes with a total of 8 sockets of Intel Xeon E7-8870 2.40GHz (Westmere-EX) processors.

While the 4-socket machine is fully interconnected, the topologies of the 8-socket and 32-socket machines have multiple hops. The max hop latency on the 4-socket and 8-socket machines is more than 30% slower than the local latency, and around 5× slower on the 32-socket machine. The inter-socket bandwidth decreases by an order of magnitude with multiple hops. Also, the total local bandwidth of the 8-socket machine is not the aggregated local bandwidth of each socket, due to its broadcast-based snooping cache coherence protocol, that stresses the interconnects. Ivybridge-EX processors use directory-based cache coherence.

The lack of knowledge and control about inter-socket routing or the cache coherence, hinders the ability to handle performance optimization similarly to a distributed network. NUMA-awareness is typically achieved in a simple way: optimizing for faster local accesses instead of remote accesses, and avoiding unnecessary centralized bandwidth bottlenecks.

OS memory allocation facilities. The OS organizes physical memory into fixed-sized (typically 4KiB) pages [19]. When an application requests memory, the OS allocates new virtual memory pages. Application libraries are responsible for organizing smaller allocations. Typically, virtual memory is not immediately backed by physical memory. On the first page fault, the OS actually allocates physical memory for that page. In a UMA system, performance is not affected by the physical location of virtual memory. In a NUMA system, however, the performance of a NUMA-agnostic application can degrade substantially (see Figure 1).

A moderate solution is to use ***interleaving***, distributing pages in a round-robin fashion across all sockets. This avoids worst-case performance scenarios with unnecessary centralized bandwidth bottlenecks, and averages memory latencies. It involves, however, a lot of remote memory accesses.

A NUMA-aware application should control and track the physical location of its virtual memory [19]. A simple strategy is to use the default ***first-touch policy***: on the first page fault, the OS allocates physical memory from the local socket (unless it is exhausted). For stronger control, additional facilities are provided. In Linux, e.g., `move_pages` can be used to query and move already touched virtual memory. We use these facilities in our data placement strategies.

3. RELATED WORK

NUMA-aware DBMS need to address two major dimensions: (a) data placement, and (b) scheduling tasks across sockets. We begin by mentioning how related work handles these di-

mensions, continue with NUMA-aware database operators, and finish with black-box approaches for NUMA-awareness.

Data placement. Many DBMS that do not mention advanced NUMA capabilities, indirectly rely on the first-touch policy, e.g., Vectorwise [36], IBM DB2 BLU [30], and the column-store of Microsoft SQL Server [21].

A few research prototypes explicitly control data placement. HyPer [23] follows a static strategy: it chunks all objects, and distributes the chunks uniformly over the sockets. ERIS [17] employs range partitioning and assigns each partition to a worker thread. ERIS dynamically adapts the sizes of the partitions to battle workload skewness. This is similar to what we propose. Moreover, we show that unnecessary partitioning has a significant overhead on large-scale NUMA topologies. Partitioning should be used only for hot data, and the granularity should be adjusted judiciously.

ATraPos [27] also uses dynamic repartitioning for OLTP workloads, to avoid transactions crossing partitions and avoid inter-partition synchronization. ATraPos optimizes for the latency of transactions, while we focus on the memory bandwidth and CPU utilization of all sockets. In contrast to HyPer, ERIS, and ATraPos, we analyze data placement strategies for dictionary-encoded columns as well.

Task scheduling. Task scheduling is an indirection layer between the OS and the execution engine. Operations are encapsulated in tasks, and a pool of worker threads is used to process them. In our previous work, we showed how task scheduling can be used in a NUMA-agnostic main-memory column-store to efficiently process mixed OLTP and OLAP workloads [28, 29]. We showed how stealing and a flexible concurrency level can help to saturate CPU resources without oversubscribing them, and how a concurrency hint can be used to adjust the task granularity of analytical partitionable operations to avoid unnecessary scheduling overhead. In this work, we make our task scheduler NUMA-aware.

Since our previous work, task scheduling has been prominent in NUMA-related work. In HyPer, each worker processes local data chunks through a whole pipeline of operators [23]. HyPer uses task stealing. We show that stealing should be avoided for memory-intensive tasks such as scans.

ERIS uses a worker per core, which processes tasks targeting a particular data partition [17]. ERIS uses the whole system, and it is not clear how this design can handle full query execution plans, intra- and inter-operator parallelism. Our NUMA-aware task scheduler handles all the workload's generic tasks. ATraPos's data-oriented execution model uses a worker per partition, similar to ERIS [27].

In the realm of commercial DBMS, IBM DB2 BLU processes data in chunks so that they fit in the CPU cache [30]. Each worker processes one chunk at a time and can steal chunks. Since NUMA-aware data placement is not specifically mentioned, chunks may not be local to the worker thread. The vanilla version of Vectorwise [36] uses static parallelism during query execution. A recent thesis [16] describes how to break query plans into stages and parallelize them using a task scheduler. Stealing from remote sockets is allowed based on the priority of unscheduled tasks and the contention of the sockets. In this paper, we show that stealing should be prevented for memory-intensive tasks.

NUMA-aware operators. There is related work on standalone operators as well. E.g., Albutiu et al [6] show that prefetching can hide the latency of remote accesses, constructing a competitive sort-merge join. Hash-joins, however, are shown to be superior [8, 20]. Yinan et al [25] optimize data shuffling on a fully-interconnected NUMA topology. Most related work, however, optimize for low concurrency, using the whole machine. In this work, we handle numerous concurrent analytical operations such as scans.

Black-box approaches. DINO [10], Carrefour [13], or the new automatic NUMA balancing of Linux, monitor performance metrics to predict an application's behavior. They attempt to improve performance by moving threads and data to balance cache load, opt for local memory accesses, and avoid bandwidth bottlenecks. Results for DBMS, however, are typically sub-optimal. A more promising approach is presented by Giceva et al [15] for DBMS evaluating a predefined global query plan. In the end, the DBMS needs to become NUMA-aware itself to optimize performance.

4. DATA PLACEMENT STRATEGIES

We start this section by describing the basic data structures in a main-memory column-store. We continue by outlining data placement strategies for them. Finally, we describe how we keep metadata information about the data placements.

4.1 Main-memory column-stores

The data of an uncompressed column can be stored sequentially in a vector in main-memory [14, 18, 21, 23]. Compression techniques are typically employed to reduce the amount of consumed memory, and potentially speed up processing. The simplest and most common compression is dictionary encoding [24]. In Figure 3, we show the data structures that compose a column in a generic column-store (naming can be different), along with an optional index.

The ***indexvector (IV)*** is an integer vector of dictionary-encoded values, called ***value identifiers (vid)***. The ***position (pos)*** relates to the row in the relation/table. The ***dictionary*** stores the sorted real values of vid. Fixed-width values are stored inline, while variable-length ones may require, e.g., using pointers or prefix-compressed storage for strings. A value lookup in the dictionary can be done with binary search, but one can also implement predicates like less-or-equal directly on the vid. The IV can be further compressed using bit-compression, i.e., using the least number of bits (called ***bitcase***) to store the vid. Vectorization enables the efficient implementation of scans, including their predicates, using SIMD. In this paper, we use scans implemented with SSE instructions on bit-compressed IV [33].

An optional ***index (IX)*** can speed up low selectivity lookups without scanning the IV. The simplest IX consists of two vectors. Each position of the first correlates to a vid,

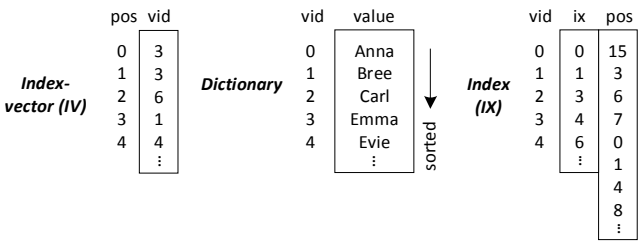

Figure 3: Example of a dictionary-encoded column.

and holds an index towards the second. The second vector holds the, possibly multiple, positions of a vid in the IV.

4.2 Data placement of columns

In this section, we propose and analyze the trade-offs of three data placement strategies, shown in Figure 4. In Table 2, we summarize which workload properties best fit each data placement and a few of their key characteristics.

Round-robin (RR). The simplest data placement is placing a whole column on a single socket. This means that queries wishing to scan this column, or do index lookups, should run and parallelize within that socket, to keep memory accesses local. A simple way to exploit this data placement for multiple columns, is to place each column on a different socket in a round-robin way.

As we show in our experiments, RR is not performant for low concurrency, because a query cannot utilize the whole machine, or for skewed workloads, because placing more than one hot column on a socket creates a hotspot on that socket. Additionally, our evaluation shows that for high concurrency, query latencies suffer a high variation, in comparison to the following partitioning strategies.

Indexvector partitioning (IVP). To overcome the aforementioned negative implications of RR, we present a novel data placement that partitions the IV across the sockets. This can happen quickly and transparently, by using, e.g., `move_pages` in Linux, to change the physical location of the involved pages without affecting virtual memory addresses. A scan can be parallelized within the socket of each part, potentially using all sockets.

The disadvantage of IVP is that there is no clear choice how to place the dictionary or the IX. Unless the column has sorted values, the ordering of the vid in the IV does not follow the same ordering as the vid of the dictionary and the IX. Thus, we interleave them across the sockets, in order to average out the latency of memory accesses during materialization (converting qualifying vid to real values from the dictionary – see Section 5.2) and during index lookups.

As we show in the experiments, the disadvantage of IVP results in high selectivity scans and index lookups suffering decreased performance in comparison to the other data placements. Although a high selectivity scan can scan the parts of the IV locally, the dominating materialization phase involves potentially numerous remote accesses to the interleaved memory of the dictionary. Similarly, index lookups suffer from remote accesses to the interleaved index.

Furthermore, both IVP and the following physical partitioning suffer from overhead when a column is partitioned across a high number of sockets. The reason is that each

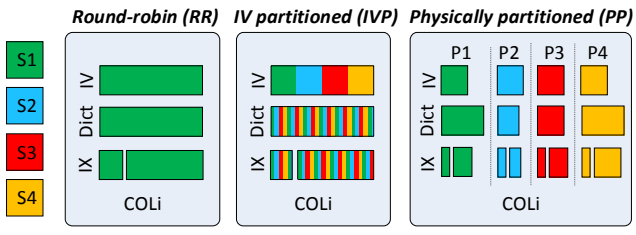

Figure 4: Different data placements of a dictionary-encoded column, with an index, on 4 sockets.

Table 2: Workload properties best fitted for each data placement, and a few key characteristics.

Data placement	Concurrency	Selectivities	Workload distrib.	Latency distrib.	Memory consumed	Readjustment	Large-scale overhead
RR	High	All	Uniform	Unfair	Normal	Quick	Low
IVP	All	Low (w/o index) & medium	Uniform & skewed	Fair	Normal	Quick	High
PP	All	All	Uniform & skewed	Fair	High	Slow	High

operation needs to be split into all partitions. As we show in our experiments, unnecessary partitioning can have a significant overhead. It should be used only for hot columns when the workload is skewed.

Physical partitioning (PP). To overcome the limitations of IVP, we can opt for an explicit physical partitioning of the table. PP can use a hash function on a set of columns, a range partitioning on a column, or a simple round-robin scheme [4, 18]. All columns are split into the ***parts*** defined by the PP. PP is useful for improving the performance through pruning, i.e., skipping a part if it is excluded by the query predicate, and for moving parts in a distributed environment (see, e.g., SAP HANA [4] and Oracle [18]). In this paper, we use PP to place each part on a different socket. Since we wish to evaluate NUMA aspects, we avoid exploiting the pruning capability in our sensitivity analysis.

The advantage of PP is that each part of a column can be allocated on a single socket. A scan is split into each part. The materialization phase for each part takes the qualifying vid of the scan and uses the local dictionary to materialize the real values. In contrast to IVP, PP is thus performant for high-selectivity queries and index lookups as well.

The disadvantages of PP are two-fold. First, PP is heavyweight and time-consuming to perform or repartition. The DBMS needs to recreate the components of all parts of the columns. For this reason, IVP is preferable for workloads that are intensive on scanning the IV, since IVP is faster to perform, and can be applied to a subset of the table's columns. The second disadvantage of PP is its potentially increased memory consumption. Although it results in non-intersecting IV across the parts of a column, the dictionaries and the IX of multiple parts may have common values. For large data types, e.g., strings, this can be expensive. The only case where this does not occur is if the column has values sorted according to the PP partitioning scheme.

Other data placements. We note that the aforementioned data placements are not exhaustive. For example, one can interleave columns across a subset of sockets. Or, one can replicate some or all components of a column on a few sockets, at the expense of memory. Replication is an orthogonal issue. The three basic data placements we describe are a set of realistic choices. More importantly, our experiments show how their basic differences affect the performance of different workloads, and motivate a design that adapts the data placement to the workload at hand.

4.3 Tracking and moving memory

We need a way to expose a column's data placement. When scheduling scans, e.g., we need to be able to query the physical location of a column, the location of a component of a

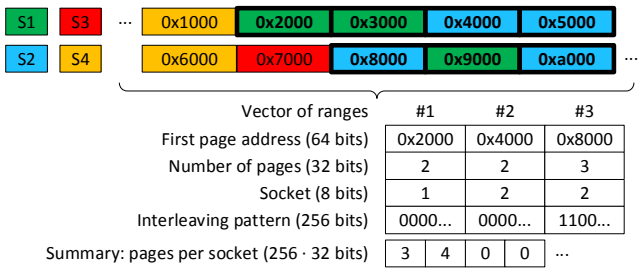

Figure 5: Example of a PSM after adding the virtual memory ranges we wish to track (bold lines).

column, and the location of a specific position in the vector of a component. To satisfy these requirements, we design a novel class, ***Page Socket Mapping (PSM)***, that summarizes the physical location of virtual address ranges.

In Figure 5, we show an exemplary PSM. The figure depicts a piece of virtual memory consisting of ten 4KiB pages. Each box includes the base address of each page.[1] The color signifies the socket where the page is physically allocated. Assume that we wish to keep metadata about the physical location of virtual address ranges `[0x2000, 0x6000)` and `[0x8000, 0xb000)`. This example can represent a tiny column, without an index, placed with IVP, where the first range holds the IV, partitioned across sockets S1 and S2, and the second range holds the interleaved dictionary.

The PSM maintains an internal vector of ***ranges***. Each range consists of a virtual page address, the number of subsequent pages, and the socket where they are physically allocated. If the range is interleaved, the interleaving pattern signifies the participating sockets, and then the socket number denotes the starting socket. The ranges are sorted by the virtual address of their base page. We choose a vector of ranges to optimize for reading the PSM instead of changing it. Looking up the physical location of a pointer includes a quick binary search on the ranges' first pages, and, in case the range is interleaved, following the interleaving pattern. Furthermore, we maintain another vector, which keeps a summary of the number of pages on each socket.

When we add virtual address ranges to the PSM, it maps them to page boundaries, checks which pages are not already included, and calls `move_pages` on Linux, not to move them but to find out their physical location. The algorithm goes through their physical locations, collapsing contiguous pages on the same socket into a new range for the internal vector. It detects an interleaving pattern when every other page is allocated on a different socket, following the same recurring pattern. When the pattern breaks, the new interleaved range is inserted in the internal vector, and the algorithm continues. The summary vector is updated accordingly.

PSM objects support additional useful functionality. We can remove memory ranges, add or subtract another PSM, ask for a subset of the metadata in a new PSM, and get the socket where the majority of the pages are. More importantly, we can also move a range to another socket or interleave it. The PSM uses `move_pages` to move the range, and update the internal information appropriately.

The space used by a PSM depends on the number of stored ranges. For the indicative sizes of Figure 5, we assume that a range can contain a maximum of 2^{32} pages (or 16TiB for 4KiB pages), and that a machine can have up to 256 sockets. The size of a PSM is $360 \cdot r + 8192$ bits, where r is the number of stored ranges. Let us examine the size of the metadata for a column on a 32-socket machine. We intend to attach a PSM to the IV, dictionary, and IX of a column, so that the scheduler can query the physical location of any component.

If a column is placed wholly on a socket, then $r = 1$ for the IV and the dictionary, and $r = 2$ for the IX (contains 2 vectors). The metadata is 26016 bits, or 3KiB. If a column is placed with IVP across all sockets, then $r = 32$ for the IV, $r = 1$ for the interleaved dictionary, and $r = 2$ for the interleaved IX. The metadata is 37176 bits, or 5KiB. If a column is physically partitioned, with 32 parts, each part is wholly placed on a socket. The metadata is around 102KiB. The size of the metadata is not large, compared to the typical sizes of columns (at least several MiB). We note that one can decrease the space substantially by losing some accuracy and the capability of querying specific virtual addresses: not storing ranges, and keeping only the summary vector.

[1]For simplicity, we display the last 4 hexadecimal characters of the 64-bit addresses instead of all 16 characters.

5. NUMA-AWARE TASK SCHEDULING

We begin this section by describing our NUMA-aware task scheduler. We then continue with outlining the different task scheduling strategies for concurrent scans, considering also the data placements of Section 4.

5.1 Task scheduler infrastructure

For a summary of task scheduling and our previous work, see Section 3. In this paper, we extend our task scheduler for NUMA-aware workloads. Tasks need to be able to choose the socket on which to run, and the task scheduler to reflect the topology of the machine. The design of our task scheduler is shown in Figure 6. Upon initialization, the scheduler divides each socket into one or more ***thread groups (TG)***. Small topologies are assigned one TG per socket, while larger topologies are assigned a couple TG per socket. Hyperthreads (HT) are grouped in the same TG. Figure 6 depicts the TG for a socket of the 4-socket machine we described in Section 1. The main purpose of multiple TG per socket is to decrease potential synchronization contention for the contained task priority queues and worker threads.

Inside a thread group. Each TG contains two priority queues for tasks. The first has tasks that can be stolen by other sockets. The second has tasks that have a ***hard affinity*** and can only be stolen by worker threads of TG of the same socket. As our experimental evaluation shows, supporting bound tasks is essential for memory-intensive workloads. The priority queues are protected by a lock. Lock-free

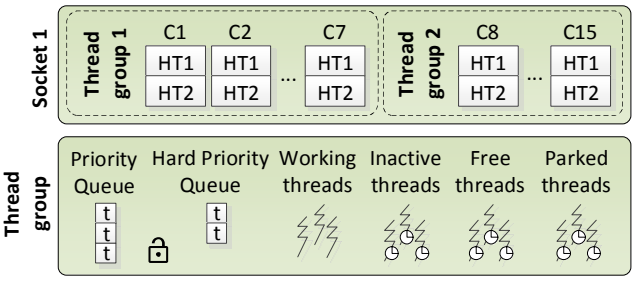

Figure 6: NUMA-aware task scheduler design.

implementations for approximate priority queues [7] can be employed for cases of numerous short-lived tasks where synchronization becomes a bottleneck. This is not the case for our experiments, mainly due to the concurrency hint (of our previous work [28]) that dynamically adjusts the task granularity of analytical operations, including scans, depending on the workload concurrency.

Each TG maintains a number of worker threads, which are distinguished to working, inactive, free, and parked. Working threads are those currently handling a task. Inactive threads are those currently sleeping in the kernel due to a synchronization primitive, while handling a task. Free threads wait on a semaphore to be woken up by the scheduler to get a newly inserted task. Parked threads, similarly to free threads, wait on a semaphore to be woken up in cases when there are no free threads. The difference between free and parked threads is that free threads wake up periodically to check for tasks, and the number of free threads cannot pass the number of H/W contexts of the TG.

The task scheduler strives to keep the number of working threads equal to the number of H/W contexts of each TG. In cases where tasks may block and become inactive, the scheduler wakes up free or parked threads to correct the number of working threads [28].

Main loop of a worker thread. The worker firstly checks that it is allowed to run, by checking that the number of working threads is not larger than the number of the H/W contexts of the TG. If it is, it goes to free mode, if the number of free threads is less than the H/W contexts of the TG, else it goes to park. If it is allowed to run, it peeks in the two priority queues to get the element with the highest priority. If there are no tasks in the TG, it attempts to steal a task from the priority queues of the other TG of the same socket. If there are no tasks, it goes around the TG of all sockets, stealing tasks (not from the hard priority queues). If the worker thread finally has a task, it executes it, and loops again. If no task is finally found, it goes to free mode.

Watchdog. We use a watchdog thread to speed up task processing [28]. It wakes up periodically to gather statistics and goes around all TG to check their number of working threads. If a TG is not saturated, but has tasks, it signals more free or unparked worker threads, or even creates new worker threads. If a TG is saturated, but has more tasks, it also monitors that, in order to wake up free threads in other TG that can potentially steal these tasks.

Task priorities. In this work, we do not set the user-defined priority of tasks [35]. Internally, however, the scheduler augments the user-defined priority of a task with more information. One crucial part is the time the related SQL statement was issued. The older the timestamp, the higher the priority of related tasks. For our experimental evaluation, this means that the tasks generated during the execution of a query are handled more or less at the same time.

Task affinities. A task can have an affinity for a socket, in which case it is inserted in the priority queue of one of the TG of the socket. Additionally, the task can specify a flag for hard affinity so that it is inserted into the hard priority queue. In case of no affinity, the task is inserted into the priority queue of the TG where the caller thread is running (for potentially better cache affinity).

By default, for NUMA-agnostic workloads, we do not bind worker threads to the H/W contexts of their TG. This is because the OS scheduler is sometimes better in balancing threads to improve the performance. We bind a worker thread to the H/W contexts of their TG only when it is about to handle a task with an affinity. And while the next task also has an affinity, the thread continues to be bound, otherwise it unbinds itself before running a task without an affinity. This gives us the flexibility to compare the OS scheduler, by not assigning affinities to tasks, against NUMA-aware scheduling by assigning affinities to tasks.

5.2 NUMA-aware scheduling of scans

To achieve NUMA-awareness, tasks need to consult the PSM of the data they intend to process to define their affinity. In Figure 7, we show the execution phases of a query selecting data from a single column, assuming the column is placed using IVP: (a) finding the qualifying matches, and (b) materializing the output. Next, we describe these phases.

Finding the qualifying matches. Depending on the predicate's estimated selectivity, the optimizer may either scan the IV, or perform a few lookups in the index (if available). For both cases, the query first needs to encode its predicate with vid. For a simple range predicate, the boundaries are replaced with the corresponding vid. If the predicate is more complex, a list of qualifying vid is built and used during the scan or the index lookups.

In the case of a scan, it is parallelized by splitting the IV into a number of ranges and issuing a task per range. The task granularity is defined by the concurrency hint [28], to avoid too many tasks under cases of high concurrency, but also opt for maximum parallelism under low concurrency. In the case of IVP, as in the example, we round up the number of tasks to a multiple of the partitions, so that tasks have a range wholly in one partition. We define a task's affinity by consulting the PSM of the IV for the task's range.

In the case of index lookups, the operation is not parallelized. For each qualifying vid, the IX is looked up to find the qualifying positions of the IV. The affinity of the single task is defined as the location of the IX. If it is interleaved, as in the example with IVP, we do not define an affinity.

The qualifying matches can be stored in two potential formats. For high selectivities, a bitvector format is preferred where each bit signifies if the relevant position is selected. For low selectivities, a vector of qualifying IV positions is built. Both formats consume little space, and we do not track their location on memory with a PSM.

Output materialization. Since we know the number of the qualifying matches, we allocate the whole output vector.

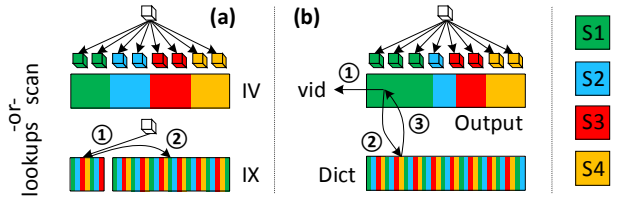

Figure 7: Task scheduling (a) for a scan or index lookups to find qualifying matches, and (b) for the output materialization, for an IVP-placed column.

Similar to the scan, we parallelize materialization by splitting the output into ranges and issuing a task per range. A task, for each qualifying position of its range, finds the relevant vid through the IV. Then it finds the real value by consulting the dictionary, and finally writes it to the output.

Because different partitions of the IV may produce more or less qualifying matches, the output may have unbalanced partitions. To define task affinities, we need a short preprocessing. Going through all qualifying matches to figure out the exact boundaries is costly. Thus, we divide the output vector length by the number of H/W contexts of the machine, to make fixed-sized regions, and find the location of their boundaries by consulting the PSM of the IV. We coalesce contiguous regions on the same socket, to make up the final list of partitions (visualized in Figure 7). For each partition, we issue a correspondingly weighted number of tasks with the affinity of that partition's socket, taking care that the number of tasks does not surpass the concurrency hint, and that each partition has at least one task.

Figure 7 hints that we place the partitions to their corresponding socket. Unfortunately, allocating a new output vector in order to specify its location turns out to have a bad performance. Especially for high-selectivity concurrent scans, it involves numerous page faults with heavy synchronization in the OS. This is one reason why SAP HANA implements its own allocators [4] to re-use virtual memory. Furthermore, we note that using `move_pages` to move the partitions also runs into a similar problem in the OS. Thus, for concurrent workloads, re-using virtual memory for the output vectors, even if writes are remote accesses, is better than explicitly placing the pages of the output vectors.

Remaining data placements. Figure 7 describes how a scan is scheduled when the selected column is placed with IVP. In the case of RR, where a column is allocated on one socket, the same scheduling is involved, but without figuring out the boundaries of the partitions of the output vector.

In the case of PP, the phase of finding qualifying matches occurs once per part, concurrently. There is a single output vector, with a length equal to the sum of the lengths of the results of each part. The preprocessing phase of the materialization happens once, in a similar way as in the case of IVP, by considering that partitions of the IV are now separate IV. The materialization phase occurs once per part, concurrently, and each one knows the region of the single output vector where to write the real values.

6. EXPERIMENTAL EVALUATION

In this section, we present a sensitivity analysis of concurrent scans for different data placement and task scheduling strategies, under various workload parameters. We use a prototype built on SAP HANA (SP9), a commercial main-memory column-store DBMS. We extend the execution engine of SAP HANA with our new NUMA-aware data placements (see Section 4) and task scheduling (see Section 5).

For all experiments, we warm up the DBMS first. We make sure that all clients are admitted, and we disable query caching. In several cases, we present additional performance metrics gathered from Linux, SAP HANA, and H/W counters (using the Intel Performance Counter Monitor tool).

We generate a dataset with a large table, taking up 100GiB of a flat CSV file. It consists of 100 million rows, an ID integer column as the primary key, and 160 additional columns of random integers generated with a uniform distribution. We use bitcases 17 to 26 in a round-robin fashion for the 160 columns, to avoid scans with the same speed [33].

We use a Java application on a different machine to generate the workload. The clients connect and build a prepared statement for each column: `SELECT COLx FROM TBL WHERE COLx >= ? AND COLx <= ?`. We note that we experiment with queries selecting a single column for simplicity. The implications of our evaluation are relevant for queries that have a predicate on multiple columns or project multiple columns also. In the former case, the first phase of Figure 7 is repeated (in parallel) for each column, to find the qualifying positions. In the latter case, the materialization phase of Figure 7 is repeated (in parallel) for each column.

After the statements are prepared, each client continuously picks a prepared statement to execute. There are no thinking times. The client does not fetch the results, otherwise the network transfer would dominate. We focus on the time required to create the results (that could potentially be used as intermediate results for higher-level operators). We measure the total throughput (TP) over 2' and report the average throughput per minute. 2' are sufficient, since all TP results we present are at least one order of magnitude more than the number of clients. The additional performance metrics presented are averaged over the whole 2' period. Every data point and metric presented is an average of 3 iterations with a standard deviation of less than 10%.

The data placement strategies we compare are:

- ***Round-robin (RR)***. Each column is allocated on one socket, in a round-robin fashion.
- ***Indexvector partitioning (IVP)***. The IV of each column is partitioned equally across the sockets.
- ***Physical partitioning (PP)***. The table is physically partitioned according to ranges of the ID column. The number of equally-sized ranges is the number of the sockets. Each part is placed on a different socket.

The task scheduling strategies we compare are:

- ***OS***. We do not define task affinities, and we do not bind worker threads, leaving the scheduling to the OS.
- ***Target***. We define task affinities. Tasks may be stolen.
- ***Bound***. We define task affinities, and set the hard affinity flag for tasks. Inter-socket stealing is prevented.

The workload parameters we vary are:

- ***Concurrency***. The number of clients in the workload.
- ***Indexes***. Whether indexes can be used or not. In the majority of the experiments, we do not use indexes.
- ***Selectivity***. The selectivity of the range predicates.
- ***Column selection***. The probability that a column may be selected. Can either be uniform or skewed.
- ***Parallelism***. We can disable intra-query parallelism, to execute each query in one task. In the majority of the experiments, intra-query parallelism is enabled.

The main server we use is the 4-socket Ivybridge-EX of Table 1. In certain cases, we present results from the other two servers, to show how some implications change due to different hardware. On all servers, the OS is a 64-bit SMP Linux 3.0.101 (SUSE Enterprise Server 11 SP3).

6.1 Uniformly distributed workload

In this section, we evaluate a uniform workload, i.e., clients pick a column to query, randomly with uniform distribution.

6.1.1 Impact of scheduling

This experiment aims to show the largest performance difference between NUMA-agnostic and NUMA-aware execution. We use *RR* to place the columns on the 4-socket server. Intra-query parallelism is enabled, and the selectivity of the queries is low (0.001%). Indexes are not used, thus scans are used, and the workload is memory-intensive. The TP and relevant performance metrics are shown in Figure 8.

Performance metrics include the CPU load, the number of tasks, and the number of tasks stolen across sockets. For the case of 1024 clients, performance metrics include the last-level cache (LLC) load misses (local and remote), the memory throughput of each socket, the instructions per cycle (IPC), the total traffic through the QPI, and the total data (without the cache coherence) traffic through the QPI.

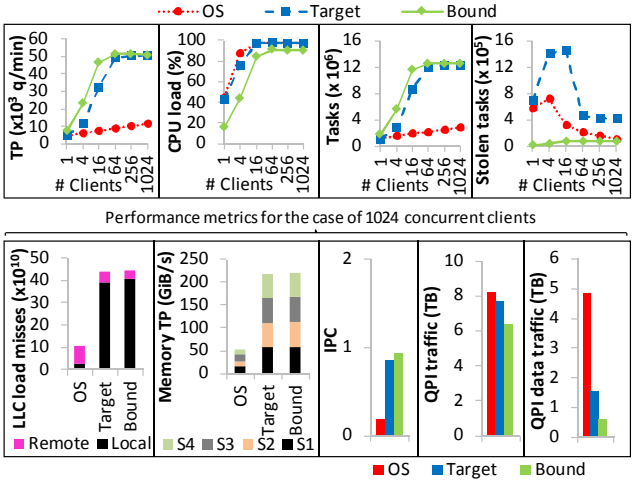

Figure 8: Evaluating the *OS*, *Target*, and *Bound* scheduling strategies, with *RR*-placed columns.

There is a 5× TP improvement with the *Target* and *Bound* strategies, over the *OS*, mainly due to the improved memory throughput. The LLC load misses, most of which are prefetched, are almost 5× more, and mostly local compared to the mostly remote misses of *OS*. The number of processed tasks is 5× higher, and IPC is also 5× higher due to faster memory accesses. QPI data traffic is reduced analogously, but cache coherence traffic is generated indirectly, even with local accesses, and cannot be avoided.

Overall, *Bound* achieves better throughput than *Target*. Although stealing improves CPU load, it hurts throughput for memory-intensive workloads due to the incurred remote accesses and stress on the QPI. We see this effect again later.

Implications. NUMA-awareness can significantly improve the performance of memory-intensive workloads. Memory-intensive tasks should be bound to the socket of their data.

6.1.2 Impact of the cache coherence protocol

Figure 9 shows the results of the previous experiment on the 8-socket Westmere-EX machine. *Bound* decreases the QPI data traffic, but the total traffic is increased, due to the broadcast-based cache coherence protocol (see Section 2). Due to the saturation of the QPI, we cannot fully exploit the memory bandwidth of all sockets simultaneously. Thus, *Bound* improves performance only by 2× compared to *OS*.

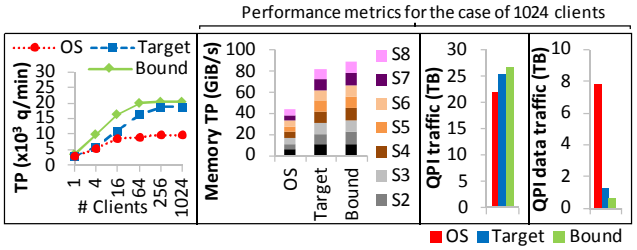

Figure 9: As Figure 8, on the 8-socket server.

Implications. H/W characteristics, such as the cache coherence protocol, can affect the NUMA impact we observe.

6.1.3 Impact of parallelism and data placement

We continue with the previous experiment, using *Bound*, but with different data placements, on the 4-socket machine. We trigger intra-query parallelism in this experiment, to show the effect of issuing a single task for each query versus multiple tasks that can potentially be distributed across the partitions of a column. The results are shown in Figure 10.

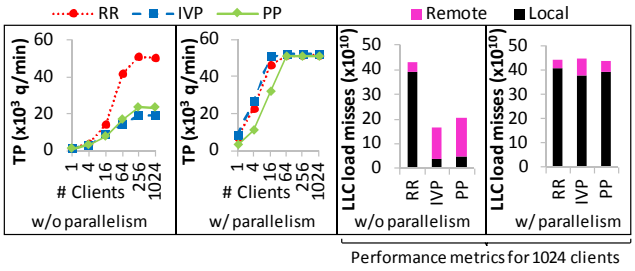

Figure 10: The effect of intra-query parallelism on the *RR*, *IVP*, and *PP* data placement strategies.

Intra-query parallelism is useful for low concurrency, since it can use more CPU resources and achieve better throughput. Also, parallelism is required when columns are partitioned. Otherwise, the single task has to access remotely the sockets of the remaining partitions. Parallelism, however, is not required for *RR* and high concurrency, since the single task is wholly local to the socket of a column. All parallel versions of the three data placements reach the same TP for high concurrency. *IVP* has slightly more remote accesses than *PP*, since the dictionary is interleaved.

Although the same throughput is reached with parallelism, there is a difference between the data placements. In Figure 11, we show violin plots of the query latency distributions. All have the same average latency. *RR*, however, is unfair, with more variance. *IVP* and *PP* have most latencies closer

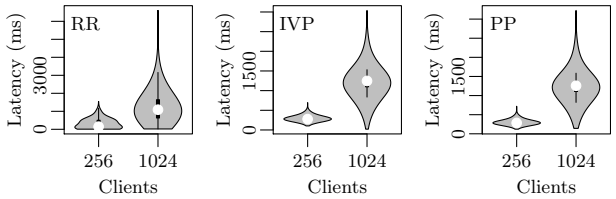

Figure 11: Violin plots of the latency distributions.

to the average. This is because in *RR*, queries queue up and execute on the socket level. With *IVP* and *PP*, each query parallelizes across all sockets, and because the tasks are prioritized according to the query's timestamp, they complete approximately in the order they were received.

Implications. Intra-query parallelism is required for partitioned data. Partitioning has a fair latency distribution.

6.1.4 Impact of scale on data placement

Although partitioning can achieve the same performance as *RR* on the 4-socket machine, this is not the case for large-scale machines. We evaluate the previous experiment with different partitioning granularities on the 32-socket machine. We increase the number of partitions up to 32, when each column is partitioned across all sockets. We take care to distribute the partitions in a round-robin manner around the sockets. For this reason, the case of one partition per column degenerates to *RR*. We show the results for *IVP* in Figure 12, for 1024 concurrent clients, and for all scheduling strategies. We note that the implications are similar for *PP*.

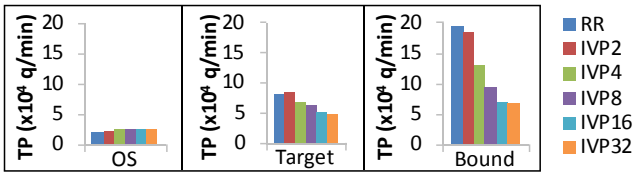

Figure 12: Combinations of scheduling strategies and *IVP* granularities, on the 32-socket machine.

Bound is the best, while *OS* is the worst and is not affected by the placement. *Target* achieves significantly less throughput than *Bound*, because on this machine there are much higher chances of stealing a memory-intensive task. Stealing stresses the remote memory controller and the interconnects due to increased traffic. For *RR*, *Target* has around 58% worse throughput than *Bound*. We revisit this effect later in Section 6.2.1.

An important implication is that as the number of partitions increases, the performance of *Target* and *Bound* decreases significantly. This is due to the overhead of parallelizing queries across numerous sockets. This overhead is unnecessary, since the workload is uniformly distributed and *RR* can already saturate the machine. As shown, *RR* and *IVP2* are comparable. Partitioning across more sockets, however, incurs overhead. Partitioning across all sockets decreases the throughput by around 70% compared to *RR*.

Interestingly, this implication is not true for all cases of concurrency. As shown in Figure 13, for low concurrency, partitioning either matches or surpasses *RR*. For high concurrency, however, partitioning proves worse than *RR*.

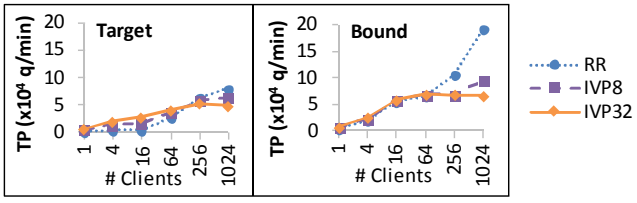

Figure 13: Scaling up the number of concurrent clients with different partitioning granularities.

Implications. Unnecessarily increasing the number of partitions has an overhead that depends on the scale of the NUMA topology and the concurrency of the workload.

6.1.5 Impact of selectivity

In this experiment, we vary the selectivity from 0.001% up to 10%. We enable indexes, evaluate 1024 clients, and use *RR* and *Bound*, on the 4-socket machine. We note that *Target* achieves similar results since the workload is uniform and the concurrency is high. The results are shown in Figure 14.

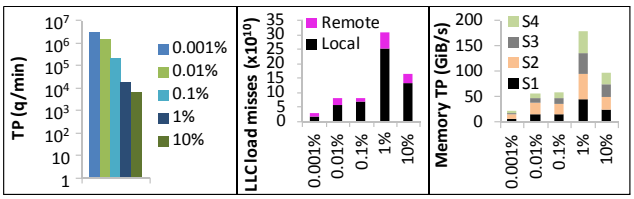

Figure 14: Evaluating different selectivities.

As expected, throughput drops as we increase the selectivity. The optimizer chooses to perform index lookups for selectivities 0.001%-0.1%, as implied by the low memory throughput and the number of LLC misses. For larger selectivities, it chooses scans. For selectivity 1%, scans dominate the execution, as is shown by the high memory throughput and the large number of LLC misses. The workload is more memory-intensive. For selectivity 10%, however, the materialization phase dominates the execution. Since the materialization consists of random accesses due to the dictionary, it is more CPU-intensive, and we observe less memory throughput and a lower number of LLC misses.

Implications. For a dictionary-encoded column, with an index, the selectivity changes the critical path of execution. It consists of CPU-intensive index lookups for low selectivities, memory-intensive scans for intermediate selectivities, and CPU-intensive materializations for high selectivities.

6.2 Skewed workload

In this section, we use a skewed workload. Clients have a 20% probability of choosing a random column from the first 80 columns of the dataset, and a 80% probability of choosing one from the remaining 80 columns.

6.2.1 Impact of stealing memory-intensive tasks

We perform the same experiment of Section 6.1.1. We use *RR*, and we intend to see the effect of scheduling strategies on performance. The results are shown in Figure 15.

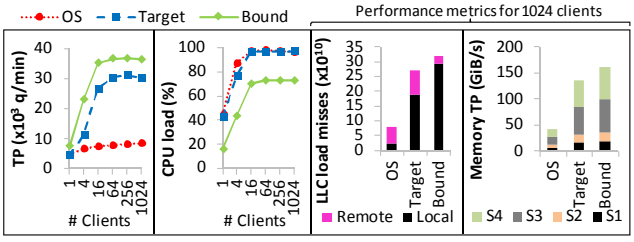

Figure 15: Evaluating the *OS*, *Target*, and *Bound* scheduling strategies, with *RR*-placed columns.

Bound still achieves the best throughput, even though it underutilizes the machine. As implied by the memory throughput, only two sockets contain the hot set of columns.

One would expect that *Target* achieves better throughput, since it utilizes more CPU resources. It decreases the throughput, however, by around 15%. This is because the two hot sockets are already saturated. Remote accesses to these sockets prevent some local accesses from queuing in the memory controllers fast. Also, increased traffic stresses the interconnects. We observe a similar effect in Section 6.1.4, where *Target* hurts throughput by around 58%.

Implications. Allowing stealing for memory-intensive tasks can decrease throughput by up to 58%.

6.2.2 Impact of partitioning

To battle skewness, apart from collocating hot and cold columns, one can partition hot columns. Figure 16 shows the results of the previous experiment using *Bound*, but evaluating the different data placements and partitioning types. *IVP* and *PP* achieve the best throughput as *RR* in the case of the uniform workload of Section 6.1.1. Skewness is smoothed out since queries are parallelized across all sockets.

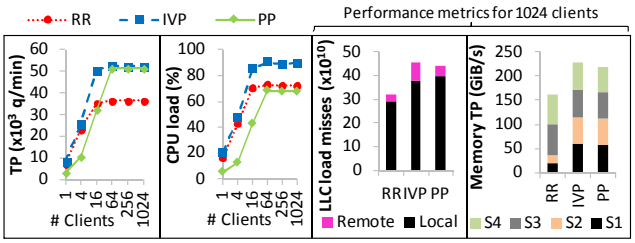

Figure 16: Evaluating the *RR*, *IVP*, and *PP* data placements, with the *Bound* scheduling strategy.

PP has less CPU load due to the concurrency hint being considered for every part, resulting in a smaller number of tasks than *IVP*. Even with more CPU load, however, *PP* cannot achieve a better throughput since the memory bandwidth is already saturated. Due to space limitations, we do not include *Target*. We note that it achieves a better throughput for low concurrency, and a similar throughput for high concurrency, since worker threads find local tasks and steal fewer tasks than in the case of *RR*.

Implications. Partitioning can significantly help in smoothing skewed memory-intensive workloads.

6.2.3 Impact of partitioning type

One needs to consider two things for choosing between *IVP* and *PP*. Firstly, *PP* is expensive to perform as it recreates columns. *PP* on this dataset takes around 18', compared to 4' for *IVP*, and consumes around 8% more memory because dictionaries contain recurrent values. Secondly, *IVP* interleaves the IX and the dictionary, which may be inefficient for index lookups and intensive materialization phases.

As a practical example, Figure 17 shows the results of the previous experiment with a high selectivity of 10%. Execution is dominated by the CPU-intensive materialization phase, which involves random accesses to the dictionary. *PP* is better, since it involves more local accesses. The throughput of *IVP* ultimately decreases due to remote accesses.

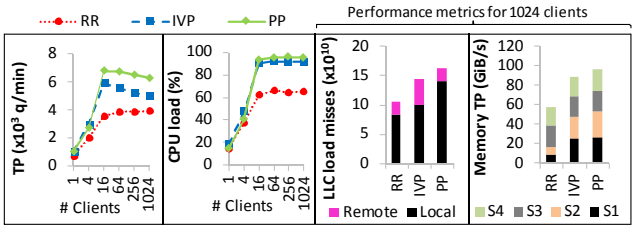

Figure 17: As Figure 16, with a high selectivity.

Implications. To battle the skewness of IV-intensive workloads, *IVP* is a quick solution. *PP* is slower to perform, but best for battling skewness in all workloads.

6.2.4 Impact of stealing CPU-intensive tasks

Stealing can be helpful, as long as tasks are CPU-intensive so that the incurred remote traffic does not stress the interconnects. Figure 18 shows the results of the previous experiment with high selectivities, but with the *Target* strategy. Stealing now does not hurt as in the case of Section 6.2.1.

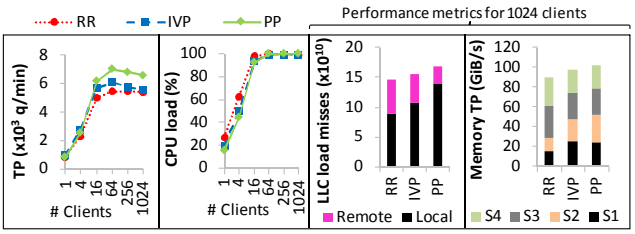

Figure 18: As Figure 17, with *Target*.

Stealing does not improve *IVP* and *PP* since they were already saturating CPU resources, but improves *RR*, which now has full CPU load and achieves the same throughput as *IVP*. Stealing incurs remote accesses, and both *RR* and *IVP* are still worse than *PP* which results in more local accesses.

Implications. Inter-socket stealing should be allowed for CPU-intensive tasks.

6.3 TPC-H and SAP BW-EML benchmarks

The implications of our analysis largely apply to the data placement of whole tables and to the scheduling of aggregations as well. We parallelize aggregations similarly to scans, and we define task affinities similarly (see Section 5.2). In this section, we show the impact of different strategies for placing tables and scheduling queries on the throughput of two benchmarks dominated by scans and aggregations.

For the first benchmark, we use TPC-H [12] with a scaling factor 100 (100GiB flat files). We measure the throughput (queries per hour) of TPC-H Q1 instances, with random parameters (see clause 2.4.1.3 [12]), which are continuously issued by 32 concurrent clients (who can saturate resources due to intra-query parallelism). The evaluation of TPC-H Q1 is dominated by aggregations on a single table (lineitem).

For the second benchmark, we measure the throughput of the reporting load of SAP BW-EML [3, 9].[2] BW-EML is representative of realistic SAP BW (business warehouse) industrial workloads. It uses an application server to host

[2]We refer the reader to an IBM redbook [11] for a more detailed introduction to BW-EML and sample statements.

BW users who query a database server. Throughput is measured in navigation steps (or queries) per hour. At the core of the data model, there are 3 *InfoCubes*, each modeling multidimensional data with an extended star schema [22]. The major part of the execution consists of queries which are dominated by scans and aggregations on the InfoCubes. Every presented measurement uses the maximum number of users that can be serviced without timeouts. Our dataset has 1 billion records (around 800GiB of flat files).

To evaluate BW-EML, we split the 32-socket rack-scale machine into two 16-socket ones, each hosting the application and the database server (running our prototype of SAP HANA) respectively. We use the database server for TPC-H as well. We evaluate the impact of different *PP* granularities, and the impact of *Target* and *Bound*, on the throughput. For each granularity, we distribute the table partitions in a round-robin manner around the sockets. Similar to Section 6.1.4, the case of one partition per table degenerates to *RR*. Due to legal reasons, throughput results are normalized to undisclosed constants c_1 and c_2, corresponding to the maximum observed throughput for TPC-H and BW-EML respectively. This does not hinder us from comparing the impact of the different data placement and scheduling strategies on the throughput. The results are shown in Figure 19.

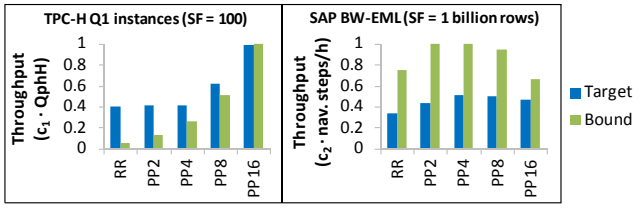

Figure 19: TPC-H and SAP BW-EML with different *PP* granularities, and different scheduling strategies.

TPC-H is severely skewed, since Q1 queries one table. Increasing the number of partitions improves performance as queries are gradually executed locally on more sockets. Our measurements show that Q1 is CPU-intensive as its execution is dominated by the multiplications of its aggregations. Due to this, *Target* is better than *Bound*. For *Bound*, increasing the number of partitions results in utilizing more sockets, finally matching the throughput of *Target*.

BW-EML, in contrast, has simpler aggregation expressions and is memory-intensive. Thus, *Bound* is better than *Target*, even if it underutilizes the machine (as in the case of *RR*), as stolen tasks incur remote accesses and stress the QPI. This is why the performance of *Target* remains low. Increasing the number of partitions up to 4 improves the performance of *Bound*, since the 3 InfoCubes are fully consuming the memory bandwidth of 12 sockets. Further partitioning, however, is unnecessary since the machine is saturated, and creates an overhead (as in Section 6.1.4 for *IVP*).

Implications. The implications are in line with our scan benchmarks. First, memory-intensive tasks should be bound, while CPU-intensive tasks not. Second, only hot data should be partitioned, up to the point that the CPU and memory bandwidth utilization is balanced across all sockets.

7. TOWARDS AN ADAPTIVE DESIGN

Our analysis shows that a main-memory column-store needs a task scheduling and data placement strategy that adapts to the workload. Considering the implications of our analysis, we envision an adaptive design. The design, shown in Figure 20, comprises of three components: (a) the catalog, (b) the task scheduler, and the (c) data placer.

Catalog. The catalog holds information about the tables, their columns, and whether a table is physically partitioned (see Section 4). The second table of the figure, e.g., has two physical partitions. Through the catalog, the PSM of the components (indexvector, dictionary, index) of any column can be accessed. Task creators can consult the PSM to define a socket affinity for their tasks.

Task scheduler. The task scheduler needs to be NUMA-aware by supporting task affinities (see Section 5). We envision that task creators assign estimated performance metrics to tasks, e.g., their memory bandwidth consumption. A black-box approach using H/W counters can be employed to find performance metrics (see Section 3). Task creators can set the hard affinity flag of a task based on whether its performance metrics indicate a memory-intensive task.

Data placer. The data placer initially places data items (tables or columns) across the sockets with RR. Afterwards, it continuously runs the workflow of Figure 20 to balance the CPU and memory bandwidth utilization of all sockets by moving or repartitioning data items. The utilization can be measured with H/W counters.

If the utilization across sockets is unbalanced, the data placer finds the hottest, most utilized, socket. By examining its active tasks, and their performance metrics, it discerns the hottest data item. If the hottest data item is not dominating the utilization of the socket, the data placer moves it to the coldest socket. If it is dominating the utilization of the socket, it increases its number of partitions, either with IVP, if the data item services tasks that mostly scan the IV of the columns, or PP otherwise (see Section 4). The new partition is moved to the coldest socket.

In case the socket utilization is balanced, the data placer iterates over the catalog to find partitioned data. For each partitioned data item, it decides if it is cold by examining if any active tasks are processing it. If a partitioned data item is cold, the data placer decreases its number of partitions.

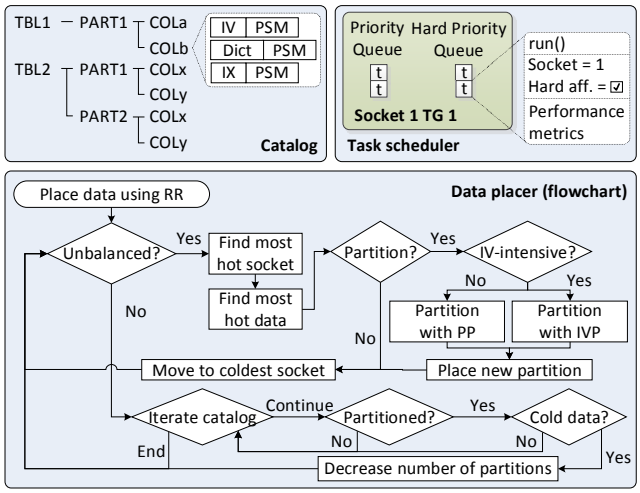

Figure 20: Our envisioned adaptive design.

8. DISCUSSION AND OUTLOOK

Applicability. To support our strategies for data placement and task scheduling, a main-memory column-store can be augmented with two functionalities. First, adopt PSM in order to realize the RR, IVP, and PP data placements (see Section 4). Second, the task scheduler needs to support a socket affinity and a hard affinity for tasks (see Section 5).

Compression. Most column-stores use dictionary encoding such as Microsoft SQL Server [21], IBM DB2 BLU [30], Oracle [18], and MonetDB [26]. Our analysis can also extend to columns without a dictionary. In this case, the output of a scan is written directly without a dictionary. Our novel IVP can be as efficient as PP for high selectivities as well.

Moreover, our scans are implemented using SSE over bit-compressed IV [33]. IV can be further compressed using, e.g., run-length or prefix encoding, and scans can use the new AVX2 instructions [34]. Different compression forms, however, do not change the basic implications for placing data and scheduling tasks. Decompression may modify the CPU- and memory-intensity of tasks. In this case, our envisioned adaptive design can accommodate them.

Additional operators. We are working on extending our analysis and our envisioned design to incorporate more complex operators, such as joins. We intend to use similar data placement and task scheduling strategies. What we need to consider additionally for, e.g., joins, is the placement of the data structures used internally in the operator, and placing correlated data on the same socket or on nearby sockets.

9. CONCLUSIONS

In this paper, we show that main-memory column-stores should depart from a static data placement and task scheduling strategy on NUMA machines, towards a strategy that adapts to the workload. We describe and implement various strategies for concurrent scans. For task scheduling, we distinguish between *Target*, and *Bound*. For data placement, we distinguish between *RR*, our novel *IVP*, and *PP*. Our extensive sensitivity analysis of the strategies, under various workload parameters, shows that (a) stealing memory-intensive tasks can hurt throughput by up to 58%, and that (b) unnecessary partitioning can hurt throughput by up to 70%. Based on our analysis, we envision an adaptive design that balances the utilization of all sockets. Partitioning should be used only for hot data in case of skewed workloads, and the number of partitions should be increased up to the point that utilization across sockets is balanced.

10. REFERENCES

[1] Intel Xeon Processor E7 Family Uncore Performance Monitoring, 2011. http://www.intel.com/.

[2] Intel Xeon Processor E7 v2 2800/4800/8800 - Datasheet - Vol. 2, 2014. http://www.intel.com/.

[3] SAP BW Enhanced Mixed Load benchmark, 2015. http://global.sap.com/campaigns/benchmark/.

[4] SAP HANA Platform SQL and System Views Reference, 2015. http://help.sap.com.

[5] A. Ailamaki et al. How to stop under-utilization and love multicores. In *SIGMOD*, pp. 189–192, 2014.

[6] M.-C. Albutiu et al. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *VLDB*, 5(10):1064–1075, 2012.

[7] D. Alistarh et al. The SprayList: A Scalable Relaxed Priority Queue. In *PPoPP*, pp. 11–20, 2015.

[8] C. Balkesen et al. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *VLDB*, 7(1):85–96, 2013.

[9] T. Becker et al. Searching for the Best System Configuration to Fit Your BW?, Nov. 2012. http://scn.sap.com/docs/DOC-33705.

[10] S. Blagodurov et al. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIXATC*, pp. 557–558, 2011.

[11] W. Chen et al. *Architecting and Deploying DB2 with BLU Acceleration*. IBM Redbooks, 2014.

[12] TPC-H Benchmark Rev. 2.17.1. http://www.tpc.org.

[13] M. Dashti et al. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ASPLOS*, pp. 381–394, 2013.

[14] F. Färber et al. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[15] J. Giceva et al. Deployment of Query Plans on Multicores. *VLDB*, 8(3):233–244, 2014.

[16] T. Gubner. Achieving Many-Core Scalability in Vectorwise. Master's thesis, TU Ilmenau, 2014.

[17] T. Kissinger et al. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workloads. In *ADMS*, pp. 74–85, 2014.

[18] T. Lahiri et al. Oracle Database In-Memory: A Dual Format In-Memory Database. In *ICDE*, 2015.

[19] C. Lameter et al. NUMA (Non-Uniform Memory Access): An Overview. *ACM Queue*, 11(7):40, 2013.

[20] H. Lang et al. Massively Parallel NUMA-aware Hash Joins. In *IMDM*, 2013.

[21] P.-A. Larson et al. Enhancements to SQL server column stores. In *SIGMOD*, pp. 1159–1168, 2013.

[22] T. Legler et al. Data Mining with the SAP NetWeaver BI Accelerator. In *VLDB*, pp. 1059–1068, 2006.

[23] V. Leis et al. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*, pp. 743–754, 2014.

[24] C. Lemke et al. Speeding up queries in column stores: a case for compression. In *DaWaK*, pp. 117–129, 2010.

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

[26] S. Idreos et al. MonetDB: Two decades of research in column-oriented database architectures. *Data Engineering*, page 40, 2012.

[27] D. Porobic et al. ATraPos: Adaptive transaction processing on hardware Islands. In *ICDE*, pp. 688–699, 2014.

[28] I. Psaroudakis et al. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *ADMS*, pp. 36–45, 2013.

[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. *VLDB*, 6(11):1080–1091, 2013.

[31] P. Russom. Best Practices Report: High-Performance Data Warehousing, 2012. http://tdwi.org/.

[32] M. Stonebraker et al. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pp. 1150–1160, 2007.

[33] T. Willhalm et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. In *VLDB*, volume 2, pp. 385–394, 2009.

[34] T. Willhalm et al. Vectorizing database column scans with complex predicates. In *ADMS*, pp. 1–12, 2013.

[35] F. Wolf et al. Extending Database Task Schedulers for Multi-threaded Application Code. In *SSDBM*, 2015, to appear.

[36] M. Zukowski et al. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.