The Art of Efficient In-memory Query Processing on NUMA Systems: a Systematic Approach

Puya Memarzia [1], Suprio Ray [2], Virendra C. Bhavsar [3]
Faculty of Computer Science, University of New Brunswick, Canada. Email: {[1]pmemarzi, [2]sray, [3]bhavsar}@unb.ca

***Abstract*—Data analytics systems commonly utilize in-memory query processing techniques to achieve better throughput and lower latency. Modern computers increasingly rely on Non-Uniform Memory Access (NUMA) architectures to achieve scalability. A key drawback of NUMA architectures is that many existing software solutions are not aware of the underlying NUMA topology and thus do not take full advantage of the hardware. Modern operating systems are designed to provide basic support for NUMA systems. However, default system configurations are typically sub-optimal for large data analytics applications. Additionally, rewriting the application from the ground up is not always feasible.**

In this work, we evaluate a variety of strategies that aim to accelerate memory-intensive data analytics workloads on NUMA systems. Our findings indicate that the operating system default configurations can be detrimental to query performance. We analyze the impact of different memory allocators, memory placement strategies, thread placement, and kernel-level load balancing and memory management mechanisms. With extensive experimental evaluation, we demonstrate that the methodical application of these techniques can be used to obtain significant speedups in four commonplace in-memory query processing tasks, on three different hardware architectures. Furthermore, we show that these strategies can improve the performance of five popular database systems running a TPC-H workload. Lastly, we summarize our findings in a decision flowchart for practitioners.

I. Introduction

The digital world is producing large volumes of data at increasingly higher rates. The breadth of applications that depend on fast and efficient data processing has grown dramatically. Main memory query processing systems have been increasingly utilized to satisfy the growing demands of the data analytics industry [1]. As hardware moves toward greater parallelism and scalability, taking advantage of the hardware's full potential remains a key challenge for these systems.

NUMA architectures are pervasive in multi-socket and in-memory rack-scale systems, as well as a growing range of CPUs with on-chip NUMA. It is clear that NUMA is ubiquitous and is here to stay, and that software needs to evolve and keep pace with these changes. Although these advances have opened a path toward greater performance, the burden of efficiently leveraging the hardware mostly falls on developers.

NUMA systems include a wide range of CPU architectures, topologies, and interconnect technologies. As such, there is no standard for what a NUMA system's topology looks like. Due to the variety of NUMA topologies and applications, fine-tuning an algorithm to a single machine configuration will not necessarily deliver better performance for other machines. Furthermore, achieving optimal performance on different system configurations can be costly and time-consuming. As a result, we were motivated to pursue strategies that can improve performance across-the-board without code tuning.

In an effort to provide a general solution that speeds up applications on NUMA systems, some researchers have proposed using NUMA schedulers that co-exist with the operating system (OS). These schedulers monitor running applications in real-time and attempt to improve performance by migrating threads and memory pages to address load balancing issues [2]–[4]. However, some of these approaches are not architecture or OS independent. For instance, Carrefour [5] requires an AMD CPU that is based on the K10 architecture, in addition to a modified OS kernel. Moreover, researchers have argued that these schedulers may not be beneficial for multi-threaded in-memory query processing [6].

Lately, researchers have started to pay attention to the issues affecting query performance on NUMA systems. These researchers have favored a more application-oriented approach that involves algorithmic tweaks to the application's source code, particularly in the context of query processing engines. Among these works, some are static solutions that attempted to make query operators NUMA-aware [7], [8]. Others are dynamic solutions that focused on work allocation to threads using work-stealing [9], data placement [10], [11], and task scheduling with adaptive data repartitioning [12]. These approaches can be costly and time-consuming to implement, and incorporating these solutions in commercial database engines will take time. Regardless, our work is orthogonal to these efforts, as we explore application-agnostic approaches to improve query performance.

Main memory query processing systems leverage data parallelism on large sets of memory-resident data, thus diminishing the influence of disk I/O. However, applications that are not NUMA-aware do not fully utilize the hardware's potential [10]. Furthermore, rewriting the application is not always an option. Solving this problem without extensively modifying the code requires tools and tuning strategies that are application-agnostic. In this work, we evaluate the viability and impact of several key parameters (shown in Table IV) that aim to achieve this. We demonstrate that significant performance gains can be achieved by managing dynamic memory allocators, thread placement and scheduling, memory placement policies, indexing, and the OS configuration. In this context, the impact and role of memory allocators have been under-appreciated and overlooked by researchers. We center our investigation around five different memory-intensive query workloads (shown in Table I) that prominently feature joins and aggregations, arguably two of the most popular and computationally expensive workloads used in data analytics. We selected the open-source MonetDB, PostgreSQL, MySQL, and Quickstep database systems, as well as a commercial database system DBMSx for evaluation. These systems were selected due to their significantly divergent architectures as well as their popularity.

An important finding from our research is that the default (out-of-the-box) OS environment can be surprisingly sub-optimal for high-performance query processing. For instance, the default Linux memory allocator *ptmalloc* can significantly lag behind other alternatives. Furthermore, with extensive experimental evaluation, we demonstrate that it is possible to systematically utilize *application-agnostic* (or black-box) approaches to obtain speedups on a variety of data analytics workloads. We show that a hash join workload can achieve a 3× speedup on Machine C (see machine topologies in Figure 1 and specifications in Table II), by replacing the memory allocator. This speedup can be further improved to 20× by optimizing the memory placement policy and modifying the OS configuration. We also show that our findings apply to other hardware configurations, by evaluating the experiments on three machines with different hardware architectures and NUMA topologies. Lastly, we show how database system performance can be improved by systematically modifying the default OS configuration and overriding the memory allocator. For example, we demonstrate that MonetDB's query latency in the TPC-H workload can be reduced by up to 43%.

The main contributions of this paper are as follows:

- Categorization and analysis of strategies to improve application performance on NUMA systems
- The first study on NUMA systems (to our knowledge) that explores the *combined impact* of different memory allocators, thread and memory placement policies, and OS-level configurations, on different analytics workloads
- Extensive experimental evaluation, involving different workloads, indexes and database systems on different machine architectures and topologies, with profiling and performance counters, and microbenchmarks
- A decision flowchart (Figure 10) to help practitioners speed up query processing on NUMA systems with minimal code modifications

The paper is organized as follows: we provide some background on the problem and the workloads in Section II. In Section III we discuss the strategies for improving query performance on NUMA systems. We present our setup and experimental results in Section IV. Finally, we discuss related work in Section V and conclude the paper in Section VI.

II. Background

A NUMA system is divided into several NUMA nodes. Each node consists of one or more processors and their local memory resources. Multiple NUMA nodes are linked together using an interconnect to form a NUMA topology. The topology of our machines is shown in Figure 1. A local memory access involves data that resides on the same node, whereas accessing data on any other node is considered a remote access. Remote data travels over the interconnect, and may need to hop through one or more nodes to reach its destination. Consequently, remote memory access is slower.

TABLE I: Experiment Workloads

Workload	SQL Equivalent
W1) Holistic Aggregation (Hashtable-based) [14]	`SELECT groupkey, MEDIAN(val) FROM records GROUP BY groupkey;`
W2) Distributive Aggregation (Hashtable-based) [14]	`SELECT groupkey, COUNT(val) FROM records GROUP BY groupkey;`
W3) Hash Join [15] **W4)** Index Nested Loop Join (*ART* [16], *Masstree* [17], *B+tree* [18], *Skip List* [19])	`SELECT * FROM table1 INNER JOIN table2 ON table1.pk = table2.fk;`
W5) TPC-H [20]	22 analytical queries (Q_1, Q_2, ... , Q_{22})

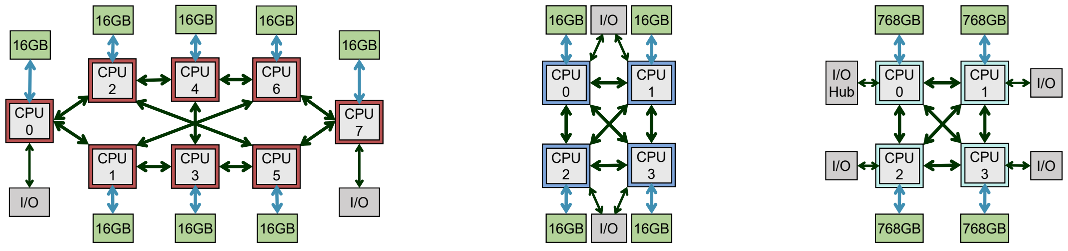
In addition to remote memory access, contention is another possible cause of sub-optimal performance on NUMA systems. Due to the memory wall [13], modern CPUs are capable of generating memory requests at a very high rate, which can easily saturate the interconnect or memory controller bandwidth [3]. Lastly, the abundance of hardware threads in NUMA systems presents a challenge in terms of scalability, particularly in scenarios with many concurrent memory allocation requests. In Section III, we explore strategies which can be used to mitigate these issues.

A. Experiment Workloads

Our goal is to analyze the effects of NUMA on query processing workloads, and show effective strategies to gain speedups in these workloads. We have selected five workloads, shown in Table I, to represent a variety of data operations that are common in data analytics and decision support systems. The implementation of these workloads is described in more detail in Section IV-B. We now provide some background on the experiment workloads.

Joins and aggregations are ubiquitous, essential data processing primitives used in many different applications. When used for in-memory query processing, they are notably demanding on cache and memory. Joins and aggregations are integral components in analytical queries and are frequently used in popular database benchmarks, such as TPC-H [20]. Although we do not evaluate transactional workloads such as TPC-C, we note that processing many concurrent transactions in-memory is also taxing on the cache and memory.

A typical aggregation workload involves grouping tuples by a designated grouping column and then applying an aggregate function to each group. Aggregate functions are divided into three categories: distributive, algebraic, and holistic. Distributive functions, such as the *Count* function used in W2 (see Table I), can be decomposed and processed in a distributed manner. This means that the input can be split up, processed, and recombined to produce the final result. Algebraic functions combine two or more distributive functions. For instance,



(a) Machine A (b) Machine B (c) Machine C

Fig. 1: Machine NUMA Topologies (machine specifications in Table II)

Average can be broken down into two distributive functions: *Count* and *Sum*. Holistic aggregate functions, such as the *Median* function used in W1, are computed by analyzing the entire input at once. Although approximation can be used to accelerate Holistic aggregation, accurate results require the processing of all input tuples for each group. As a result, these aggregate functions are typically more expensive in terms of computing resources. W3 represents a hash join query. As described in [15], the query joins two tables with a size ratio of 1:16, which is designed to mimic common decision support systems. The join is performed by building a hash table on the smaller table and probing the larger table for matching keys. W4 is an index nested loop join using the same dataset as W3. The main difference between W3 and W4 is that W3 builds an ad hoc hash table to perform the join, whereas W4 uses a pre-built in-memory index that accelerates lookups to one of the join relations. W5 is a database system workload, using the well-known queries and datasets from the TPC-H benchmark [20]. We evaluate W5 on five database systems: MonetDB [21], PostgreSQL [22], MySQL [23], DBMSx, and Quickstep [24]. In order to analyze query performance under memory-bound (rather than I/O-bound) situations, we configure the databases to use large buffer caches where applicable. Furthermore, we measure multiple warm runs for each query.

III. Improving Query Performance on NUMA Systems

Achieving good performance on NUMA systems involves careful consideration of thread placement, memory management, and load balancing. We explore application-agnostic strategies that can be applied to the data analytics application in either a black box manner, or with minimal tweaks to the code. Some strategies are exclusive to NUMA systems, whereas others may also yield benefits on uniform memory access (UMA) systems. These strategies consist of: overriding the memory allocator, defining a thread placement and affinity scheme, using a memory placement policy, and changing the operating system configuration. In this section, we describe these strategies and outline the options used for each one.

A. Dynamic Memory Allocators

Dynamic memory allocators track and manage dynamic memory during the lifetime of an application. The performance impact of memory allocators is often overlooked in favor of exploring ways to tweak the application's algorithms. It can be argued that this makes them one of the most under-appreciated system components. Both UMA and NUMA systems can benefit from faster or more efficient memory allocators. However, the potential is greater on NUMA systems, as the performance penalties caused by inefficient memory or cache behavior can be significantly higher. Key allocator attributes include allocation speed, fragmentation, and concurrency. Most developers use the default memory allocation functions to allocate or deallocate memory (*malloc/new* and *free/delete*) and trust that their library will perform these operations efficiently. In recent years, with the growing popularity of multi-threaded applications, there has been a renewed interest in memory allocators, and several alternative allocators have been proposed. Earlier iterations of *malloc* used a single lock resulting in serialized access to the global memory pool. Although recent *malloc* implementations provide support for multi-threaded scalability, there are now several competing memory allocators that aim for faster performance and reduced contention and memory consumption overhead. We evaluate the following allocators: *ptmalloc*, *jemalloc*, *tcmalloc*, *Hoard*, *tbbmalloc*, *mcmalloc*, and *supermalloc*.

1) ptmalloc (pthreads malloc): The standard memory allocator that ships with most Linux distributions. *ptmalloc* aims to attain a balance between speed, portability, and space-efficiency. It supports multi-threaded applications by employing multiple mutexes to synchronize and protect access to its data structures. The downside of this approach is the possibility of lock contention on the mutexes. In order to mitigate this issue, *ptmalloc* creates additional regions of memory (arenas) whenever contention is detected. Allocated memory can never move between arenas. *ptmalloc* employs a per-thread cache for small allocations. This helps to further reduce lock contention by skipping access to the memory arenas when possible.

2) jemalloc (Jason Evans malloc) [25]: First appearing as an SMP-aware memory allocator for the FreeBSD operating system, *jemalloc* was later expanded and adapted for use as a general purpose memory allocator. When a thread requests memory from *jemalloc* for the first time, it is assigned a memory allocation arena. Arena assignments for multi-threaded applications follow a round-robin order. In order to further improve performance, this allocator also uses thread-specific caches, which allows some allocation operations to completely avoid arena synchronization. Lock-free radix trees

TABLE II: Machine Specifications

System	**Machine A**	**Machine B**	**Machine C**
CPUs/ Model	8×Opteron 8220	4×Xeon E7520	4×Xeon E7-4850 v4
CPU Frequency	2.8GHz	2.1GHz	2.1GHz
Cores/Threads	16/16	16/32	32/64
Last Level Cache	2MB	18MB	40MB
4KB TLB Capacity	L1:32×4KB L2:512×4KB	L1:64×4KB L2:512×4KB	L1:64×4KB L2:1536×4KB
2MB TLB Capacity	L1:8×2MB -	L1:32×2MB -	L1:32×2MB L2:1536×2MB
NUMA Nodes	8	4	4
NUMA Topology	Twisted Ladder	Fully Connected	Fully Connected
Relative NUMA Node Memory Latency	Local: 1.0 1 hop: 1.2 2 hop: 1.4 3 hop: 1.6	Local: 1.0 1 hop: 1.1	Local: 1.0 1 hop: 2.1
Interconnect Bandwidth	2GT/s	4.8GT/s	8GT/s
Memory Capacity	16GB/node 128GB Total	16GB/node 64GB Total	768GB/node 3TB Total
Memory Clock	800MHz	1600MHz	2400MHz
Operating System	Ubuntu 16.04	Ubuntu 18.04	CentOS 7.5
Linux Kernel	4.4	4.15	3.10

track allocations across all arenas. *jemalloc* attempts to reduce memory fragmentation by packing allocations into contiguous blocks of memory and re-using the first available low address. This allocator maintains allocation arenas on a per-CPU basis and associates threads with their parent CPU's arena. We use *jemalloc* version 5.1.0 for our experiments.

3) tcmalloc (thread-caching malloc) [26]: Developed by Google and included as part of the *gperftools* library, its goal is to provide faster memory allocations in memory-intensive multi-threaded applications. *tcmalloc* handles small allocations using thread-private caches that do not require locking. Large allocations use a central heap that is organized into contiguous groups of pages called "spans". Each span stores multiple allocations of a particular size class. However, applications that use many different size classes may waste memory due to under-utilization of the memory spans. The central heap uses fine-grained locking on a per-span basis. As a result, two threads requesting memory from the central heap can do so concurrently, as long as their requests fall in different class categories. We use *tcmalloc* from *gperftools* release 2.7.

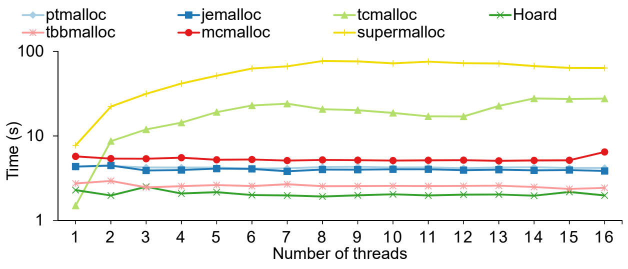
4) Hoard [27]: A standalone cross-platform allocator replacement designed specifically for multi-threaded applications, *Hoard*'s main design goals are to provide memory efficiency, reduce allocation contention, and prevent false sharing. At its core, *Hoard* consists of a global heap (the "hoard") that is protected by a lock and accessible by all threads, as well as per-thread heaps that are mapped to each thread using a hash function. *Hoard* uses heuristics to detect temporal locality and fill cache lines with objects that were allocated by the same thread, thus reducing false sharing. We evaluate *Hoard* version 3.13 in our experiments.

5) tbbmalloc: The *tbbmalloc* allocator is included as part of the *Intel Thread Building Blocks* (*TBB*) library [28]. This allocator pursues better performance and scalability for multi-threaded applications, and generally considers increased memory consumption as an acceptable tradeoff. Allocations in *tbbmalloc* are supported by per-thread memory pools. If the allocating thread is the owner of the target memory pool, no locking is required. If the target pool belongs to a different thread then the request is placed in a synchronized linked list, and the owner of the pool will allocate the object. We used version 2019 Update 4 of the *TBB* library for our experiments.

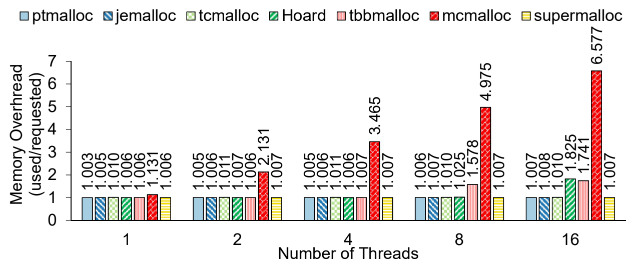
6) supermalloc [29]: This *malloc* replacement synchronizes concurrent memory allocation requests using hardware transactional memory (HTM) if available, and falls back to *pthread* mutexes if HTM is not available. *supermalloc* prefetches all necessary data while waiting to acquire a lock in order to minimize the amount of time spent in the critical section. It uses homogeneous chunks of objects for allocations smaller than 1MB, and supports larger objects using operating system primitives. Given a pointer to an object, its corresponding chunk is tracked using a lookup table. This lookup table is implemented as a large 512MB array, which takes advantage of the fact that most of its virtual memory will not be committed to physical memory by the OS. For our experiments, we use the latest publicly released source code, which was last updated in October 2017.

7) mcmalloc (many-core malloc) [30]: This allocator focuses on mitigating multi-threaded lock contention by reducing calls to kernel space, dynamically adjusting the memory pool structures, and using fine-grained locking. Similar to other allocators, *mcmalloc* uses a global and local (per-thread) memory pool layout. It monitors allocation requests, and dynamically splits its global memory pool into two categories: frequently used memory chunk sizes, and infrequently used memory chunk sizes. Dedicated homogeneous memory pools are created to support frequently used chunk sizes. Infrequent memory chunk sizes are handled using size-segregated memory pools. *mcmalloc* reduces system calls by batching multiple chunk allocations together. We use the latest *mcmalloc* source code, which was updated in March 2018.

8) Memory Allocator Microbenchmark: We now describe a multi-threaded microbenchmark that we use to gain insight on the relative performance of these memory allocators. Our goal is to answer the question: how well do these allocators scale up on a NUMA machine? The microbenchmark simulates a memory-intensive workload with multiple threads utilizing the allocator at the same time. Each thread completes 100 million memory operations, consisting of allocating memory and writing to it, or reading an existing item and then deallocating it. The distribution of allocation sizes is inversely proportional to the size class (smaller allocations are more frequent). We use two metrics to compare the allocators: execution time and memory allocation overhead. The execution time gives an idea of how fast an allocator is, as well as its efficiency when being used in a NUMA system



(a) Multi-threaded Scalability



(b) Memory Consumption Overhead

Fig. 2: Memory Allocator Microbenchmark - Machine A

by concurrent threads. In Figure 2a, we vary the number of threads in order to see how each allocator behaves under contention. The results show that *tcmalloc* provides the fastest single-threaded performance, but immediately falls behind the competition once the number of threads is increased. *Hoard* and *tbbmalloc* show good scalability and outperform the other allocators by a considerable margin. In Figure 2b, we show each allocator's overhead. This is calculated by measuring the amount of memory allocated by the OS (as maximum resident set size), and dividing it by the amount of memory that was requested by the microbenchmark. This experiment shows considerably higher memory overhead for *mcmalloc* as the number of threads increases. *Hoard* and *tbbmalloc* are slightly more memory-hungry than the other allocators, which highlights *jemalloc* as a low memory overhead alternative with decent performance. We omit *supermalloc* and *mcmalloc* from subsequent experiments due to their poor performance in terms of scalability and memory overhead respectively.

B. Thread Placement and Scheduling

Defining an efficient thread placement strategy is a well-known and essential step toward obtaining better performance on NUMA systems. By default, the kernel thread scheduler is free to migrate threads created by the program between all available processors. The reasons for doing so include power efficiency and balancing the heat output of different processors. This behavior is not ideal for large data analytics applications and may result in significantly reduced query throughput. The thread migrations slow down the program due to cache invalidation, as well as a likelihood of moving threads away from their data. The combination of cache invalidation, loss of locality, and non-deterministic behavior of the OS scheduler, can result in fluctuating runtimes (as shown in Figure 3 with 16 threads). Binding threads to processor cores can solve this issue by preventing the OS from migrating threads. However, deciding how to place the threads requires careful consideration of the topology and workload.

A thread placement strategy details the manner in which threads are assigned to processors. We explore two strategies for assigning thread affinity: *Dense* and *Sparse*. A *Dense* thread placement involves packing threads in as few processors as possible. The idea behind this approach is to minimize remote access distance and maximize resource sharing. In contrast, the *Sparse* strategy attempts to maximize memory bandwidth utilization by spreading the threads out among the

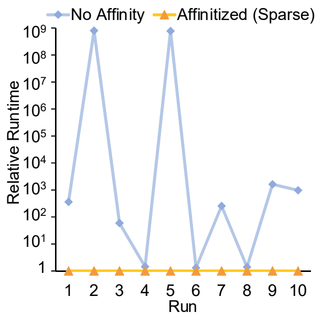


Fig. 3: OS thread scheduler behavior vs thread affinity strategy - Consecutive runs of W1 - Machine A

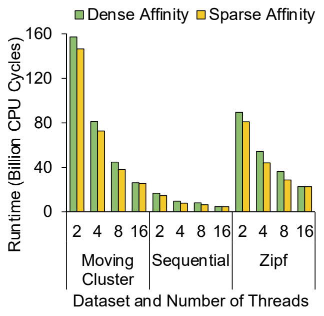


Fig. 4: Comparison of *Sparse* and *Dense* thread affinitization strategies - W1 - Machine A

processors. There are a variety of ways to implement and manage thread placement, depending on the level of access to the source code and the library used to provide multithreading. Applications built on OpenMP can use the *OMP_PROC_BIND* and *OMP_PLACES* environment variables in order to set a thread placement strategy.

To demonstrate the impact of affinitization, we evaluate workload W1 from Table I. The workload involves building a hash table with key-value pairs taken from a moving cluster distribution. Figure 3 depicts 10 consecutive runs of this workload on Machine A. The runtime number of the default configuration (*no affinity*) is expressed in relation to the *affinitized* configuration. The results highlight the inconsistency of the default OS behavior. In the best case, the *affinitized* configuration is several orders of magnitude faster, and the worst case runtime is still around 27% faster. In order to gain a better understanding of how each configuration affects the workload, we use the *perf* tool to measure several key metrics. The results, depicted in Table III, show that the operating system migrates the worker threads many times during the course of the workload. The *Sparse* affinity configuration prevents migration-induced cache invalidation, which in turn reduces cache misses. Furthermore, the affinitized configuration increases the ratio of local memory accesses.

In Figure 4 we evaluate the *Sparse* and *Dense* thread affinity strategies on workload W1, and vary the number of threads. We also vary the dataset (see Section IV-B) in order to ensure that the distribution of the data records is not the defining factor. The goal of this experiment is to determine if threads benefit from being on the same NUMA node against utilizing a greater number of the system's memory controllers. The

TABLE III: Profiling thread placement - W1 on Machine A - Default (managed by OS) vs Modified (*Sparse* policy)

Performance Metric	Default	Modified	Percent Change
Thread Migrations	33196	16	−99.95%
Cache Misses	1450M	972M	−32.95%
Local Memory Accesses	367M	374M	+2.06%
Remote Memory Accesses	159M	108M	−31.95%
Local Access Ratio	0.70	0.78	+10.77%

Sparse policy achieves better performance when the workload is not using all available hardware threads. This is due to the threads having access to additional memory bandwidth, which plays a major role in memory-intensive workloads. When all hardware threads are occupied, the two policies perform almost identically. Henceforth, we use the *Sparse* configuration (when applicable) for all our experiments.

C. Memory Placement Policies

Memory pages are not always accessed from the same threads that allocated them. Memory placement policies are used to control the location of memory pages in relation to the NUMA topology. As a general rule of thumb, data should be on the same node as the thread that processes it and sharing should be kept to a minimum. However, too much consolidation can lead to congestion of the interconnects and contention on the memory controllers. The *numactl* tool applies a memory placement policy to a process, which is then inherited by all its children (threads). We evaluate the following policies: *First Touch*, *Interleave*, *Localalloc*, and *Preferred*. We also use hardware counters to measure the ratio of local to total (local + remote) memory accesses.

Modern Linux systems employ a memory placement policy called *First Touch*. In *First Touch*, each memory page is allocated to the first node that performs a read or write operation on it. If the selected node does not have sufficient free memory, an adjacent node is used. This is the most popular memory placement policy and represents the default configuration for most Linux distributions. *Interleave* places memory pages on all NUMA nodes in a round-robin fashion. In some prior works, memory interleaving was used to spread a shared hash table across all available NUMA nodes [9], [31], [32]. In *Localalloc*, the memory pages are placed on the same NUMA node as the thread performing the allocation. The *Preferred* policy places all newly allocated memory pages on a single node that is selected by the user. This policy will fall back to using other nodes for allocation when the selected node has run out of free space and cannot fulfill the allocation.

D. Operating System Configuration

In this section, we outline two key operating system mechanisms that affect NUMA applications: Virtual Memory Page Management (Transparent Hugepages), and Load Balancing Schedulers (AutoNUMA). These mechanisms are enabled out-of-the-box on most Linux distributions.

1) Virtual Memory Page Management: OS memory management works at the virtual page level. Pages represent chunks of memory, and their size determines the granularity of which memory is tracked and managed. Most Linux systems use a default memory page size of 4KB in order to minimize wasted space. The CPU's TLB caches can only hold a limited number of page entries. When the page size is larger, each TLB entry spans a greater memory area. Although the TLB capacity is even smaller for large entries, the total volume of cached memory space is increased. As a result, larger page sizes may reduce the occurrence of TLB misses. *Transparent Hugepages* (*THP*) is an abstraction layer that automates the process of creating large memory pages from smaller pages. *THP* is not to be confused with *Hugepages*, which depends on the application explicitly interfacing with it and is usually disabled by default. We use the global *THP* toggles on our Linux machines to configure its behavior.

2) Automatic NUMA Load Balancing: There have been several projects to develop NUMA-aware schedulers that facilitate automatic load balancing. Among these projects, *Dino* [2] and *AsymSched* [4] do not provide any source code, and *Numad* [33] is designed for multi-process load balancing. *Carrefour* [3] provides public source code, but requires an AMD CPU based on the K10 architecture (with instruction-based sampling), as well as a modified operating system kernel. Consequently, we opted to evaluate the *AutoNUMA* scheduler, which is open-source and supports all hardware architectures. *AutoNUMA* was initially developed by Red Hat and later on merged with the Linux kernel. It attempts to maximize data and thread co-location by migrating memory pages and threads. *AutoNUMA* has two key limitations: 1) workloads that utilize data sharing can be mishandled due to the unnecessary migration of memory pages between nodes, 2) it does not factor in the cost of migration or contention, and thus aims to improve locality at any cost. *AutoNUMA* has received continuous updates, and is considered to be one of the most well-rounded kernel-based NUMA schedulers. We use the *numa_balancing* kernel parameter to toggle the scheduler.

IV. Evaluation

In this section we describe our setup and evaluate the effectiveness of our strategies. In Section IV-A we outline the hardware/software specifications of our machines. Section IV-B describes the datasets, implementations, and systems used. We analyze the impact of the OS configuration in Section IV-C. In Section IV-E we evaluate these techniques on database engines running TPC-H queries. We explore the effects of overriding the default system memory allocator in Section IV-D. Finally, we summarize our findings in Section IV-F.

A. Experimental Setup

We run our experiments on three different machines based on different architectures. This is done to ensure that the applicability of our findings is not biased to a particular system's characteristics. The NUMA topologies of these machines are depicted in Figure 1 and their specifications are

TABLE IV: Experiment Parameters (bolded = system defaults)

Parameter	**Values**
Experiment Workload	W1) Holistic Aggregation [14] W2) Distributive Aggregation [14] W3) Hash Join [15] W4) Index Nested Loop Join using: 1)ART [16], 2)Masstree [17], 3)B+tree [18], 4)Skip List [19] W5) TPC-H Queries (Q_1 to Q_{22}) [20]
Thread Placement Strategy	**None** (OS scheduler is free to migrate threads), Sparse, Dense
Memory Placement Policy	**First Touch**, Interleave, Localalloc, Preferred
Memory Allocator	**ptmalloc**, jemalloc, tcmalloc, Hoard, tbbmalloc
Dataset Distribution	Moving Cluster (default for W1), Sequential (default for W3 and W4), Zipfian (default for W2), TPC-H (W5)
Database System (W5)	MonetDB [21], PostgreSQL [22], MySQL [23], DBMSx, Quickstep [24]
OS Configuration	**AutoNUMA on**/off, **Transparent Hugepages (THP) on**/off
Hardware System	Machine A, Machine B, Machine C

outlined in Table II. We used *LIKWID* [34] to measure each system's relative memory access latencies, and the remainder of the specifications were obtained using product pages, spec sheets, and Linux system queries. Now we outline some of the key hardware specifications for each machine. Machine A is an eight socket AMD-based server with a total of 128GB of memory. As the only machine with eight NUMA nodes, machine A provides us with an opportunity to study NUMA effects on a larger scale. The twisted ladder topology shown in Figure 1a is designed to minimize inter-node latency with three *HyperTransport* interconnect links per node. As a result, Machine A has three remote memory access latencies, depending on number of hops between the source and the destination. Each node contains an AMD Opteron 8220 CPU running at 2.8GHz and 16GB of memory. Machine B is a quad-socket Intel server with four NUMA nodes and a total memory capacity of 64GB. The NUMA nodes are fully connected, and each node consists of an Intel Xeon E7520 CPU running at 1.87GHz with 16GB of memory. Lastly, Machine C contains four sockets populated with Intel Xeon E7-4850 v4 processors. Each processor constitutes a NUMA node with 768MB of memory, providing a total system memory capacity of 3TB. The NUMA nodes of this machine are fully connected.

Our experiments are coded in C++ and compiled using GCC 7.3.0 with the -O3 and -march=native flags. Likewise, all dynamic memory allocators and database systems are compiled from source. Machines B and C are owned and maintained by external parties and are based on different Linux distributions. The experiments are configured to utilize all available hardware threads on each machine.

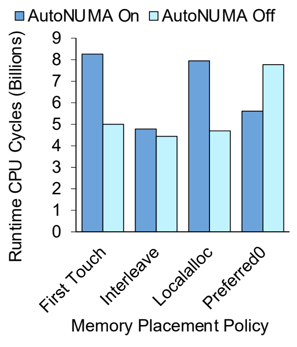
B. Datasets and Implementation Details

In this section, we outline the datasets and code used for the experiments. Unless otherwise noted, all workloads operate on datasets that are stored in memory resident data structures, hence avoiding any I/O bottlenecks.

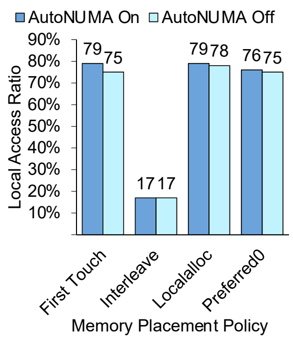
The aggregation workloads (W1 and W2) evaluate a typical hash-based aggregation query, based on a state-of-the-art concurrent hash table [35], which is implemented as a shared global hash table [14]. The datasets used for the aggregation workloads are based on three different data distributions: *Moving Cluster* (default), *Sequential*, and *Zipfian*. Each dataset consists of 100 million records with a group-by cardinality of one million. In the *Moving Cluster* dataset, the keys are chosen from a window that gradually slides. The *Moving Cluster* dataset provides a gradual shift in data locality that is similar to workloads encountered in streaming or spatial applications. In the *Sequential* dataset, we generate a series of segments that contain multiple number sequences. The number of segments is equal to the group-by cardinality, and the number of records in each segment is equal to the dataset size divided by the cardinality. This dataset mimics transactional data where the key incrementally increases. In the *Zipfian* dataset, the distribution of the keys approximates *Zipf's law*. We first generate a *Zipfian* sequence with the desired cardinality c and *Zipf* exponent $e = 0.5$. Then we take n random samples from this sequence to build n records. The *Zipfian* distribution is used to model many big data phenomena, such as word frequency, website traffic, and city population.

The join workloads (W3 and W4) evaluate a typical join query involving two tables. W3 is a non-partitioning hash join based on the code and dataset from [15]. The dataset contains two tables sized at 16M and 256M tuples, and is designed to simulate a decision support system. W4 is an index nested loop join that uses the same dataset as W3. We evaluated several in-memory indexes for this workload: *ART* [16], *Masstree* [17], *B+tree* [18], and *Skip List* [19]. *ART* [16] is based on the concept of a Radix tree. *Masstree* [17] is a hybrid index that uses a *trie* of *B+trees* to store keys. *B+tree* [18] is a cache-optimized in-memory $B^{+}tree$. *Skip List* is a canonical implementation of a *Skip List* [19].

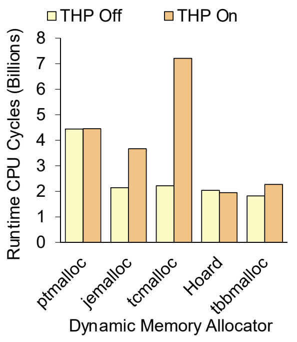
We use the TPC-H workload (W5) to investigate how our strategies can benefit database systems. This entails some limitations, as databases are complex systems with less flexibility compared to microbenchmarks and codelets. Although there are many available database systems that are TPC-H compliant, we note that comparing an extensive variety of systems is beyond the scope of this paper. We evaluate W5 on the MonetDB [21] (version 11.33.3), PostgreSQL [22] (version 11.4), MySQL [23] (version 8.0.17), DBMSx, Quickstep [24] (latest Github version as of October 2019) database systems. MonetDB is an open-source columnar store that uses memory mapped files with demand paging and multiple worker threads for its query processing. PostgreSQL is a widely-used open-source row store that supports intra-query parallelism using multiple worker processes and a shared memory pool for communication. We configured PostgreSQL with a 42GB buffer pool. MySQL is an open-source row store that remains highly popular. DBMSx is a commercial hybrid row/column-store with a parallel in-memory query execution engine. Quickstep is an open-source hybrid store with a focus on in-memory analytical workloads. W5 uses version 2.18 of the TPC-



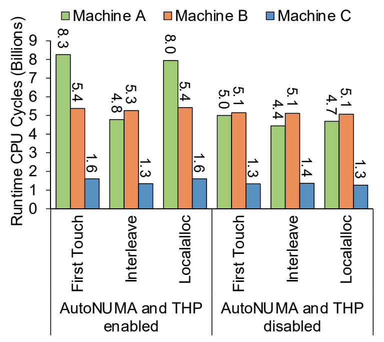
(a) *AutoNUMA* effect on execution time - Machine A



(b) Profiling effect of *AutoNUMA* on Local Access Ratio - Machine A



(c) Impact of *THP* on memory allocators - Machine A



(d) Combined effect of *AutoNUMA* and *THP* on different memory placement policies - variable machine

Fig. 5: Effect of *AutoNUMA* load balancing and *THP* page merging on memory placement policies and allocators - W1

H dataset specifications. We evaluate the impact of the OS configuration on each database system, using all 22 queries and a dataset scale factor of 20. Additionally, we use Queries 5 and 18 to show the impact of utilizing different memory allocators, as both queries involve a combination of joins and aggregations.

The experimental parameters are shown in Table IV. We use the maximum number of hardware threads supported by each machine. In W1-W4, we measure the average workload execution time using the timer from [15]. In W5, we use the built-in query timing features of each database system.

C. *Operating System Configuration Experiments*

In this section, we evaluate three key OS mechanisms that affect NUMA behavior: NUMA Load Balancing (*AutoNUMA*), *Transparent Hugepages* (*THP*), and the system's memory placement policy. The experiments demonstrate each parameter's affect on query performance. We also examine how these variables are affected by other experiment parameters, such as hardware architecture, and the interaction between *THP* and memory allocators.

1) AutoNUMA Load Balancing Experiments: In Figures 5a and 5b, we evaluate W1 and toggle the state of *AutoNUMA* Load Balancing between *On* (the system default) and *Off*. The results in Figure 5a show that *AutoNUMA* slows down the runtime for the *First Touch*, *Interleave*, and *Localalloc* memory placement policies. In most cases, *AutoNUMA*'s overhead dominates any performance gained by migrating threads and memory pages. The best runtime is obtained by applying the *Interleave* policy and disabling *AutoNUMA*. If *AutoNUMA* is enabled, the best approach is to apply the *Interleave* policy, which may be useful for scenarios where *superuser* access is unavailable. We observed similar behavior for the other workloads and machines. *AutoNUMA* had a significantly detrimental effect on runtime. The best overall approach is to use memory interleaving and disable *AutoNUMA*. The *Local Access Ratio (LAR)* shown in Figure 5b specifies the ratio of memory accesses that were satisfied with local memory [3] compared to all memory accesses. *AutoNUMA* attempts to increase *LAR* without considering other costs, such as moving threads and memory, or memory controller contention. Due to this, the *First Touch* policy with *AutoNUMA* enabled (system default) is 86% slower than *Interleave* without *AutoNUMA*, despite a higher *LAR* measurement. In summary, we obtain significant speedups using a modified OS configuration, and note that *LAR* is not necessarily an accurate predictor of performance on NUMA systems.

2) Transparent Hugepages Experiments: Next we evaluate the effect of the *Transparent Hugepages* (*THP*) configuration, which automatically merges groups of 4KB memory pages into 2MB memory pages. As shown in Figure 5c, *THP*'s impact on the workload execution time ranges from detrimental in most cases to negligible in other cases. As *THP* alters the composition of the operating system's memory pages, support for *THP* within the memory allocators is the defining factor on whether it is detrimental to performance. *tcmalloc*, *jemalloc*, and *tbbmalloc* are currently not handling *THP* well. We hope that future versions of these memory allocators will rectify this issue out-of-the-box. Although most Linux distributions enable *THP* by default, our results indicate that it is better to disable *THP* for high performance data analytics.

3) Hardware Architecture Experiments: Here we show how the performance of data analytics applications running on different machines with different hardware architectures is affected by the memory placement strategies. For all machines, the default configuration uses the *First Touch* memory placement, and both *AutoNUMA* and *THP* are enabled. The results depicted in Figure 5d show that Machine A is slower than Machine B when both machines are using the default configuration. However, using the *Interleave* memory placement policy and disabling the operating system switches allows Machine A to outperform Machine B by up to 15%. Machine A shows the most significant improvement from operating system and memory placement policy changes, and the workload runtime is reduced by up to 46%. The runtime for Machine C is reduced by up to 21%. The performance improvement on Machine B is around 7%, which is fairly modest compared to the other machines. Although Machines B and C have a similar inter-socket topology, the relative local and remote memory access latencies are much closer in Machine B (see Table II). Henceforth, we keep *AutoNUMA* and *THP* disabled for our experiments, unless otherwise noted.

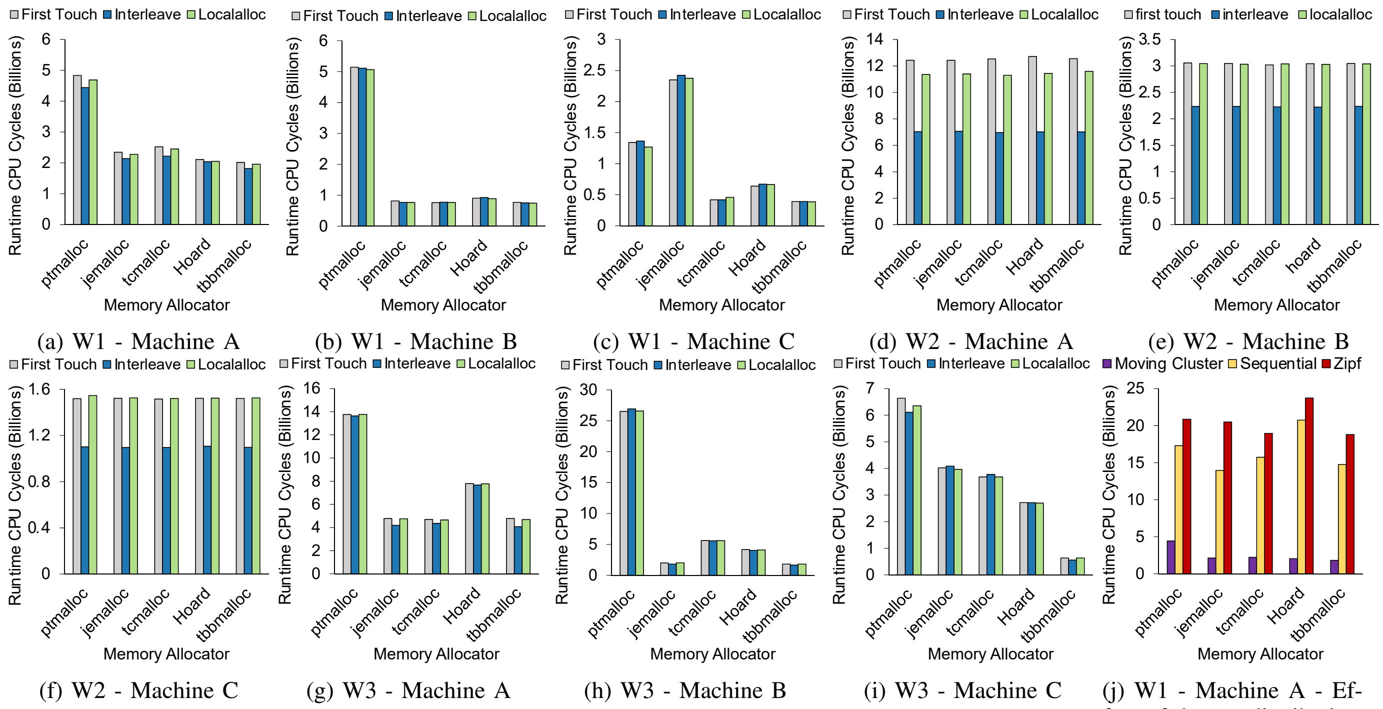

(a) W1 - Machine A (b) W1 - Machine B (c) W1 - Machine C (d) W2 - Machine A (e) W2 - Machine B (f) W2 - Machine C (g) W3 - Machine A (h) W3 - Machine B (i) W3 - Machine C (j) W1 - Machine A - Effect of dataset distribution

Fig. 6: Comparison of memory allocators - variable workload, memory placement policy, and machine

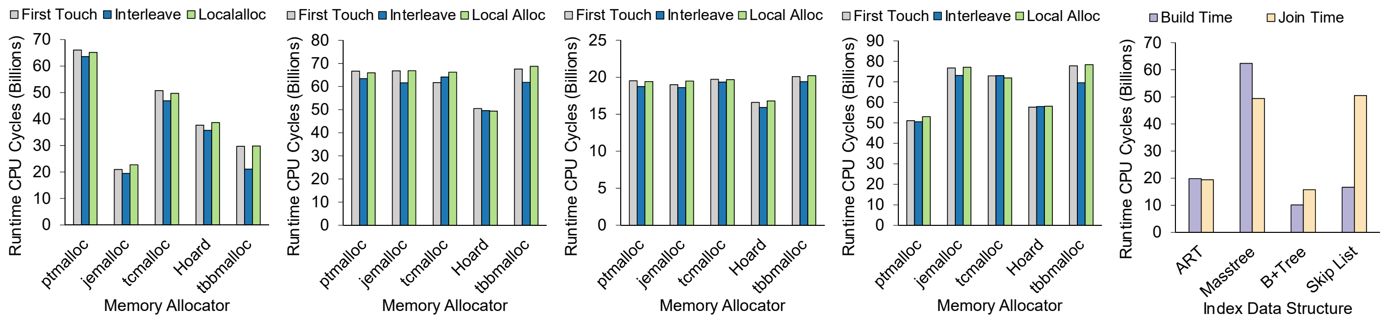
D. Memory Allocator Experiments

In Section III-A8, we used a memory allocator microbenchmark to show that there are significant differences in both multi-threaded scalability and memory consumption overhead. In this section, we explore the performance impact of overriding the system default memory allocator on four in-memory data analytics workloads.

1) Hashtable-based Experimental Workloads: In Figure 6, we show our results for the holistic aggregation (W1), distributive aggregation (W2), and hash join (W3) workloads running on each of the machines. In addition to the memory allocators, we vary the memory placement policies for each workload. The results show significant runtime reductions on all three machines, particularly when using *tbbmalloc* in conjunction with the *Interleave* memory placement policy. The holistic aggregation workload (W1) shown in Figure 6a to 6c extensively uses memory allocation during its runtime to store the tuples for each group and calculate their aggregate value. Utilizing *tbbmalloc* reduced the runtime of W1 by up to 62% on Machine A, 83% on Machine B, and 72% on Machine C, compared to the default allocator (*ptmalloc*). The results for the join query (W3) depicted in Figures 6g to 6i also show significant improvements, with *tbbmalloc* reducing workload execution time by 70% on Machine A, 94% on Machine B, and 92% on Machine C. The distributive aggregation query (W2) shown in Figure 6d to 6f speeds up by 44%, 27%, and 28% on Machines A, B, and C respectively. This speedup is almost entirely due to the *Interleave* memory placement policy. Although W2 is not allocation-heavy and does not gain much benefit from a faster memory allocator, it can still be accelerated using a more efficient memory placement policy.

2) Impact of Dataset Distribution: The performance of query workloads and memory allocators can be sensitive to the access patterns induced by the dataset distribution. The three tested datasets have the same number of records, but differ in the way the record keys are distributed (see Section IV-B for more information). In our previous experiments, we used the *Heavy Hitter* dataset as the default dataset for W1. In Figure 6j, we vary the dataset distribution to investigate its impact on different memory allocators. The results show that *tbbmalloc* continues to produce the largest speedups on both the *Zipf* and *Sequential* datasets. We also observe this trend on Machines B and C, but omit them due to space constraints.

3) Effect on In-memory Indexing: In W4, we investigate index nested loop join query processing with different in-memory indexes. The type of index used to accelerate the nested loop join workload (W4) plays a key role in determining its speed. We evaluate four in-memory indexes: *ART* [16], *Masstree* [17], *B+tree* [18], and *Skip List* [19]. As the index is pre-built, the workload is relatively light in terms of number of memory allocations during the join phase, hence factors such as scan/lookup times, materialization, and locality play a greater role. For each index, we vary the memory allocator and memory placement policy and measure the join time. The results, depicted in Figures 7a to 7c, show that runtime can be significantly improved for most of the tested indexes. In Figure 7a, we show that ART's join time can be substantially improved using the *jemalloc* or *tbbmalloc* allocators. A key characteristic of ART is that it uses variable node sizes and a variety of compression techniques for its trie, thus requesting a greater variety of size classes from the memory allocator, compared to the other allocators. In Figures 7b and



(a) ART Index - Join Times (b) Masstree Index - Join Times (c) B+tree index - Join Times (d) Skip List Index - Join Times (e) Index build and join times (best configuration)

Fig. 7: Index nested loop join workload (W4) - variable memory allocators and memory placements - Machine A

7c *Masstree* and *B+tree* show a notable improvement with the *Hoard* allocator. Both indexes rely on grouping many keys per node, which is favorable for *Hoard*'s straightforward global heap approach. *Skip List* breaks the trend as the only index that runs fastest with *ptmalloc*. Finally, we summarize the results in Figure 7e, which depicts each index's build and join times using their fastest configuration. The results show that we were able to speed up the two fastest indexes (*ART* and *B+tree*) despite their inherent lack of NUMA-awareness.

E. Database Engine Experiments

In this section, we evaluate the TPC-H workload (W5) on five database systems: MonetDB, PostgreSQL, MySQL, DBMSx, and Quickstep. Investigating NUMA strategies on database systems is more challenging compared to standalone in-memory microbenchmark workloads, as there is considerably more complexity involved in storing and loading the data and great care must be taken to ensure that disk I/O and caching do not skew the results. To ensure fair and consistent results, we clear the page cache using the *proc/sys/vm/drop_caches* command before running each query, disregard the first (cold) run, and measure the mean runtime for five additional runs. In a similar vein to our previous experiments, we evaluate the impact of the OS configuration, memory placement policies, and memory allocators. Due to an issue with PostgreSQL producing severely sub-optimal plans for queries 17 and 22, we evaluate modified versions of these two queries which use joins instead of nested queries. All other database systems run the original versions of queries 17 and 22. We used the following parameters to speed up W5: *First Touch* memory placement, *AutoNUMA* disabled, THP disabled (for all except DBMSx), and the *tbbmalloc* memory allocator. In Figure 8, we present the speedups obtained across all 22 TPC-H queries for each of the database systems. The results show that MonetDB's query latencies improved by up to 43%, with an average improvement of 14.5%. In comparison, the gains for PostgreSQL are less consistent. Query latency improved by up to 27.6%, but the average improvement is 3% and seven queries take slightly longer to complete. We believe these variances are due to PostgreSQL's rigid multi-process query processing approach, which sometimes opts to use only one worker process and thus fails to fully utilize the hardware. MySQL's query latency is reduced by up to 49% with an average reduction of 12%. Lastly, we observe that DBMSx query latency improved by up to 43% with an average of 21%. Lastly, Quickstep query latency speeds up by up to 40% and an average of 7%. All five database systems obtained speedups from modifying the default OS configuration.

Next we investigate the effect of memory allocator overriding on MonetDB. To do so, we select queries 5 and 18 due to their usage of both joins and aggregation. The results, shown in Figure 9a, indicate that *tbbmalloc* can provide an average query latency reduction of up to 11% for Query 5, and 20% for Query 18, compared to *ptmalloc*. As with other memory allocator experiments, we measure the average of five runs.

F. Summary

The strategies explored in this paper, when carefully applied, can significantly speed up query processing workloads without the need for source code modification. The effectiveness and applicability of these strategies to a workload depend on several factors. Figure 10 shows a strategic plan for practitioners. The flowchart outlines a systematic guide to improving performance on NUMA systems, along with some general recommendations. We base these recommendations on our extensive experimental evaluation using multiple machine architectures and workloads. Starting with thread management, we showed that thread affinitization can be critical for NUMA systems, but more importantly how a *Sparse* placement approach can maximize performance in situations that are memory-bandwidth-bound. We then showed that the default OS configuration can have a significant detrimental effect on query performance. The overhead of *AutoNUMA* and *THP* was demonstrated to be too costly for high performance data analytics workloads. Although *superuser* privileges are required to modify *AutoNUMA* and *THP*, we observed that optimizing the memory placement policy (such as using *Interleave*) can mostly mitigate their negative impact. We also investigated dynamic memory allocators using a microbenchmark. The microbenchmark results showed that there are considerable differences between the allocators, both in terms of scalability and efficiency. In our evaluation, we demonstrated that these differences translate into real gains in analytical query processing workloads, although the performance gains depend on the

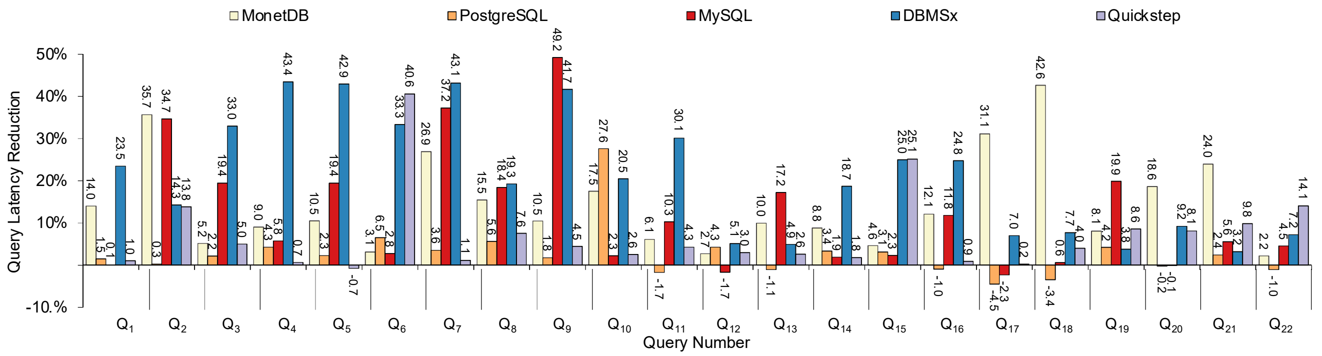
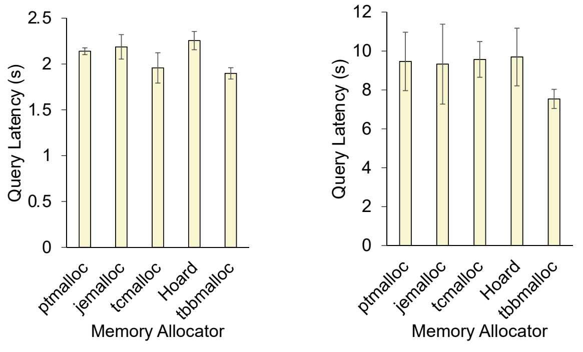


Fig. 8: 22 TPC-H queries (W5) scale factor 20 - Query latency reduction - Variable database systems - Machine A



(a) TPC-H Q_5 (b) TPC-H Q_{18}

Fig. 9: Effect of memory allocator on TPC-H query latency - MonetDB - Machine A

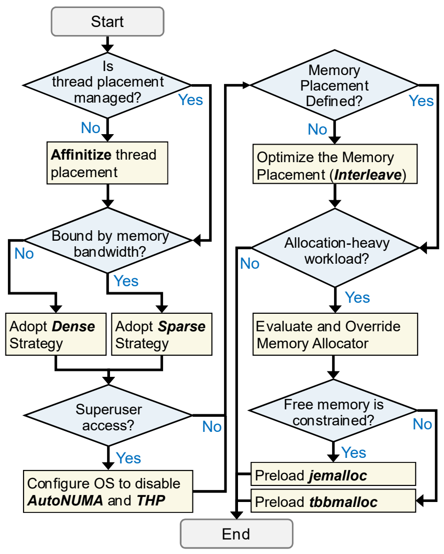

Start
Is thread placement managed?
Yes
No
Affinitize thread placement
Bound by memory bandwidth?
No
Yes
Adopt Dense Strategy
Adopt Sparse Strategy
Superuser access?
No
Yes
Configure OS to disable AutoNUMA and THP
Memory Placement Defined?
Yes
No
Optimize the Memory Placement (Interleave)
Allocation-heavy workload?
No
Yes
Evaluate and Override Memory Allocator
Free memory is constrained?
No
Yes
Preload jemalloc
Preload tbbmalloc
End

Fig. 10: Application-agnostic Decision Flowchart

way the workloads allocate memory. For example, allocation-heavy workloads, such as the hash join (W3) benefited the most, whereas the index nested loop join (W4) exhibited smaller gains due to the prebuilt index. Although we have shown that *tbbmalloc* frequently outperformed its competitors on different machines and workloads, we recommend experimentation with new/updated memory allocators before selecting a solution.

V. Related Work

The rising demand for high performance parallel computing has motivated many works on leveraging NUMA architectures. We now outline existing research that is relevant to our work.

In [36], Kiefer et al. evaluated the performance impact of NUMA effects on multiple independent instances of the MySQL database system. Popov et al. [37] explored the combined effect of thread and page placement using supercomputing benchmarks running on NUMA systems. They observed that co-optimizing thread and memory page placement can provide significant speedups. Durner et al. [38] explored the performance impact of dynamic memory allocators on a database system running TPC-DS. The authors obtained significant speedups utilizing *jemalloc* and *tbbmalloc*, which agrees with our findings. In this paper, we evaluate a broader and newer range of allocators, and additional NUMA parameters, indexes, datasets, databases, and workloads.

Some prior work has pursued automatic load balancing approaches that can improve NUMA system performance in an application-agnostic manner. These approaches generally focus on improving performance by altering the process and/or memory placement. Some examples include *Dino* [2], *Carrefour* [3], *AsymSched* [4], *Numad* [33], and *AutoNUMA* [33]. These schedulers have been shown to improve performance in some cases, particularly on systems running multiple independent processes. However, some researchers have claimed that these schedulers do not provide much benefit for multi-threaded query processing applications [6], [7].

A different approach involves either extensively modifying or completely replacing the OS. This is done with the goal of providing a custom tailored environment for the application. Some researchers have pursued this direction with the goal of providing an OS that is more suitable for large database applications [39]–[41]. Custom operating systems aim to reduce the burden on developers, but their adoption has been limited. In the past, researchers in the systems community proposed a few new OSes for multicore architectures, including Corey [42], *Barrelfish* [43] and *fos* [44]. Although none were widely adopted by the industry, we believe these efforts underscore the need to investigate the impact of system and architectural aspects on query performance.

Some researchers have favored an application-oriented approach that fine-tunes query processing algorithms to the hardware. Wang et al. [8] proposed an aggregation algorithm for NUMA systems, based on radix partitioning. The authors also proposed a load balancing algorithm that focuses on inter-socket task stealing, and prohibits task stealing until a socket's local tasks have been completed. Leis et al. [9] presented a NUMA-aware parallel scheduling algorithm for hash joins, which uses dynamic task stealing in order to deal with dataset skew. Schuh et al. [7] conducted an in-depth comparison of thirteen main memory join algorithms on a NUMA system. Our work is orthogonal to these approaches and they can benefit from applying the application-agnostic strategies that we have suggested.

VI. Conclusion

In this work, we have outlined and investigated several application-agnostic strategies to speedup query processing on NUMA machines. Our experiments on five analytics workloads have shown that it is possible to obtain significant speedups by utilizing these strategies. We also demonstrated that current operating system default configurations are generally sub-optimal for in-memory data analytics. Our results, surprisingly, indicate that many elements of the default OS environment, such as *AutoNUMA*, *Transparent Hugepages*, default memory allocator (eg. *ptmalloc*), and the OS thread scheduler, should be disabled or customized for high performance analytical query processing, regardless of the hardware generation. We have also demonstrated that memory allocator performance on NUMA systems can be a major bottleneck and that this under-appreciated topic is ripe for investigation. We obtained large speedups for our query processing workloads by overriding the default dynamic memory allocator with alternatives such as *tbbmalloc*.

As our approach does not target a specific NUMA topology, we have shown that our findings can be applied to systems with different architectures. As hardware architectures continue to advance towards greater parallelism and greater levels of memory access partitioning, we hope our results and decision flowchart can help practitioners to accelerate data analytics.

References

[1] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011, pp. 195–206.

[2] S. Blagodurov, A. Fedorova, S. Zhuravlev, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *PACT*, 2010, pp. 557–558.

[3] M. Dashti *et al.*, "Traffic management: a holistic approach to memory placement on NUMA systems," *SIGPLAN Notices*, vol. 48, no. 4, 2013.

[4] B. Lepers, V. Quema, and A. Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," in *USENIX ATC*, 2015.

[5] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," *LWN. net*, 2012.

[6] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling up concurrent main-memory column-store scans: towards adaptive NUMA-aware data and task placement," *VLDBJ*, vol. 8, no. 12, 2015.

[7] S. Schuh, X. Chen, and J. Dittrich, "An experimental comparison of thirteen relational equi-joins in main memory," in *SIGMOD*, 2016.

[8] L. Wang, M. Zhou, Z. Zhang, M.-C. Shan, and A. Zhou, "NUMA-aware scalable and efficient in-memory aggregation on large domains," *TKDE*, vol. 27, no. 4, 2015.

[9] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age," in *SIGMOD*, 2014, pp. 743–754.

[10] T. Kissinger *et al.*, "ERIS: A NUMA-aware in-memory storage engine for analytical workloads," *VLDB Endow.*, vol. 7, no. 14, pp. 1–12, 2014.

[11] D. Porobic, E. Liarou, P. Tozun, and A. Ailamaki, "Atrapos: Adaptive transaction processing on hardware islands," in *ICDE*, 2014, pp. 688–699.

[12] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Adaptive NUMA-aware Data Placement and Task Scheduling for Analytical Workloads in Main-memory Column-stores," *VLDB*, pp. 37–48, 2016.

[13] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, University of California, Berkeley, Tech. Rep., 2006.

[14] P. Memarzia, S. Ray, and V. C. Bhavsar, "A Six-dimensional Analysis of In-memory Aggregation," in *EDBT*, 2019, pp. 289–300.

[15] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *SIGMOD*, 2011.

[16] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," in *DaMoN*. ACM, 2016, pp. 1–8.

[17] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*. ACM, 2012, pp. 183–196.

[18] T. Bingmann, "STX B+ Tree," panthema.net/2007/stx-btree, 2019.

[19] S. Vokes, "skiplist," github.com/silentbicycle/skiplist, 2016.

[20] "TPC-H benchmark specification 2.18.0_rc2," 2019.

[21] MonetDB B.V., "MonetDB," monetdb.org, 2018.

[22] "PostgreSQL," postgresql.org, 2019.

[23] Oracle Corporation, "MySQL," mysql.com, 2019.

[24] J. M. Patel *et al.*, "Quickstep: A data platform based on the scaling-up approach," *VLDBJ*, vol. 11, no. 6, pp. 663–676, 2018.

[25] J. Evans, "A scalable concurrent malloc (3) implementation for FreeBSD," in *BSDCan*, 2006.

[26] S. Ghemawat and P. Menage, "Tcmalloc: Thread-caching malloc," github.com/gperftools/, 2015.

[27] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *SIGARCH*, vol. 28, no. 5, pp. 117–128, 2000.

[28] W. Kim and M. Voss, "Multicore desktop programming with intel threading building blocks," *IEEE software*, vol. 28, no. 1, pp. 23–31, Jan 2011.

[29] B. C. Kuszmaul, "SuperMalloc: a super fast multithreaded malloc for 64-bit machines," in *SIGPLAN Notices*, vol. 50. ACM, 2015, pp. 41–55.

[30] A. Umayabara and H. Yamana, "MCMalloc: A scalable memory allocator for multithreaded applications on a many-core shared-memory machine," in *IEEE Big Data*, 2017, pp. 4846–4848.

[31] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *ICDE*, 2013, pp. 362–373.

[32] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper, "Massively Parallel NUMA-Aware Hash Joins," in *IMDM*, 2013.

[33] Red Hat Inc, "Red Hat Enterprise Linux Product Documentation," 2018.

[34] G. Hager, G. Wellein, and J. Treibig, "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments," in *ICPP*. IEEE, 2010, pp. 207–216.

[35] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *EuroSys*, 2014, pp. 1–14.

[36] T. Kiefer, B. Schlegel, and W. Lehner, "Experimental evaluation of numa effects on database management systems," *BTW*, 2013.

[37] M. Popov, A. Jimborean, and D. Black-Schaffer, "Efficient thread/page/-parallelism autotuning for numa systems," in *ICS*. ACM, 2019, pp. 342–353.

[38] D. Durner, V. Leis, and T. Neumann, "On the impact of memory allocation on high-performance query processing," in *DaMoN*, ser. DaMoN'19. ACM, 2019, pp. 21:1–21:3.

[39] J. Giceva, "Operating Systems Support for Data Management on Modern Hardware," sites.computer.org/debull/A19mar/p36.pdf, 2019.

[40] J. Giceva, A. Schüpbach, G. Alonso, and T. Roscoe, "Towards database/-operating system co-design," in *SFMA*, vol. 12, 2012.

[41] J. Giceva, G. Zellweger, G. Alonso, and T. Rosco, "Customized OS support for data-processing," in *DaMoN*, 2016, pp. 1–6.

[42] S. Boyd-Wickizer *et al.*, "Corey: An operating system for many cores," in *USENIX OSDI*, ser. OSDI'08, 2008, pp. 43–57.

[43] A. Baumann *et al.*, "The multikernel: A new os architecture for scalable multicore systems," in *SIGOPS SOSP*, 2009, pp. 29–44.

[44] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, 2009.