# Experimental Evaluation of NUMA Effects on Database Management Systems

Tim Kiefer, Benjamin Schlegel, Wolfgang Lehner
Technische Universität Dresden
Database Technology Group
Dresden, Germany
{tim.kiefer, benjamin.schlegel, wolfgang.lehner}@tu-dresden.de

**Abstract:** NUMA systems with multiple CPUs and large main memories are common today. Consequently, database management systems (DBMSs) in data centers are deployed on NUMA systems. They serve a wide range of database use-cases, single large applications having high performance needs as well as many small applications that are consolidated on one machine to save resources and increase utilization.

Database servers often show a natural partitioning in the data that is accessed, e.g., caused by multiple applications accessing only their data. Knowledge about these partitions can be used to allocate a database's memory on the different nodes accordingly: a strategy that increases memory locality and reduces expensive communication between CPUs.

In this work, we show that partitioning a database's memory with respect to the data's access patterns can improve the query performance by as much as 75%. The allocation strategy is enabled by knowledge that is available only inside the DBMS. Additionally, we show that grouping database worker threads on CPUs, based on their data partitions, improves cache behavior, which in turn improves query performance. We use a self-developed synthetic, low-level benchmark as well as a real database benchmark executed on the MySQL DBMS to verify our hypotheses. We also give an outlook on how our findings can be used to improve future DBMS performance on NUMA systems.

## 1 Introduction

Servers with 2, 4, or 8 CPUs on a single board, many cores, and non-uniform memory access (NUMA systems) to 64, 128, or more gigabytes of RAM are common today. NUMA systems are scalable and offer the ease of programming with distributed memory hidden behind a global address space and a cache coherence protocol. Compared to other parallel or distributed systems, NUMA systems are tightly coupled with fast communication links. Both, bandwidth and latency, of the links that connect different CPUs have greatly improved in the last few years. However, applications with high performance needs may still benefit from careful memory and thread placement and optimized memory locality.

Database management systems in data centers are increasingly often deployed on NUMA systems where they serve different use-cases: single large applications as well as many small applications that are consolidated on one machine. The large amount of memory combined with the intensive use of compression techniques [WKHM00, ZHNB06] lead to structured data that often completely fit in memory. Therefore, main-memory database systems will gradually outnumber disk-based database systems. With more and more data in memory, there often is no need to perform expensive I/O operations to execute a query, which in turn reduces the relevance of traditional buffering mechanisms in the DBMS (buffer pools). Hence, the design and performance optimization focus shifts from disk-centric, with I/O having been the main bottleneck for decades, to memory-centric with new challenging research topics like optimal memory layout and access.

Database servers often show a natural partitioning in the data that is accessed, caused by, e.g., (a) multi-database operation, (b) private schema multi-tenancy, or (c) business or application requirements. A single database server can host different databases (a) for one or multiple applications. This can be used to allow an application access to all the databases it needs on a single machine or to introduce multi-tenancy and hence let applications with moderate performance requirements share resources [CJMB11, KL11]. Resource consolidation also motivates (b): private schema multi-tenancy. Here, different applications share a single database but operate on private tables. Some Database-as-a-Service providers use this scheme to provide scalable solutions, e.g., Microsoft SQL Azure [BCD$^+$11]. The third case of data partitioning (c) occurs when a single large database stores data of different parts of the same business. Although, e.g., marketing and sales tables are stored in the same database, it is likely that they are never accessed together in one transaction. All cases of database partitioning have in common that parts of the database system (databases, schemas, or sets of tables) are accessed independently of one another. Transactions that spread across multiple data partitions are either impossible (private databases) or at most rare.

Knowledge about data partitions can be used to allocate a database's memory on the different CPUs' memories accordingly. This strategy can increase memory locality and reduce expensive communication between CPUs. Our goal is to evaluate the potential of thread/memory placement strategies in a set of experiments. We quantify the performance improvement achieved by co-locating threads and the memory they access. Additionally, we show that grouping database worker threads that access the same data on one CPU improves cache behavior, which in turn improves query performance. We use a thorough synthetic benchmark as well as a real database benchmark based on the TPC-H schema, executed on the MySQL database management system to verify our hypotheses. We also give an outlook on how our findings can be used to improve future DBMS performance on NUMA systems.

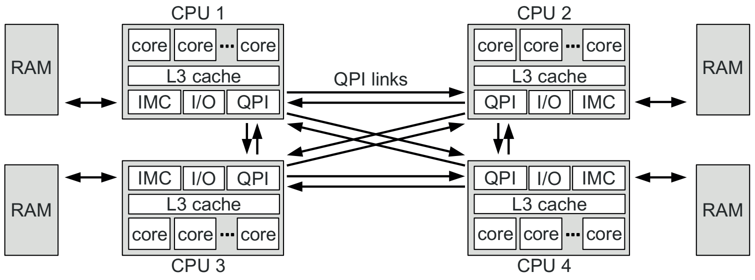


Figure 1: Overview of a NUMA system (e.g., Intel Westmere Architecture)

## 2 Preliminaries on NUMA Architecture

This section gives an overview on NUMA systems, main characteristics, and methods to influence application behavior on NUMA systems.

A NUMA system consists of multiple CPUs (also called sockets or nodes) that are connected via point-to-point connections. Each CPU has its own *local memory* that can be accessed with a lower latency and a higher bandwidth compared to the memories of the other CPUs (*remote memory*). Depending on the CPU vendor, the nodes are connected using different standards: Intel systems use QuickPath Interconnect[1] (QPI) whereas AMD systems use HyperTransport[2]. Figure 1 illustrates an Intel NUMA system with four sockets. Each of the CPUs has its own RAM that is connected via an *integrated memory controller* (IMC). The cores of a CPU each have a dedicated L1- and L2-cache (not shown); the L3-cache (also *last level cache* or *LLC*) is usually shared among all cores of a CPU. Each CPU has four outgoing and incoming QPI links, of which three are connected to other CPUs. Hence, four CPUs can be fully connected, i.e., each CPU can access any remote memory with only a single hop. Larger systems are usually not fully connected.

All currently available NUMA systems ensure cache coherency to keep the caches of the participating CPUs consistent (sometimes denoted as ccNUMA systems). There are several cache-coherence protocols like MESI, MOESI, and MESIF. QPI relies on the MESIF protocol, which adds a fifth state (*Forward*) to the MESI protocol, while HyperTransport uses MOESI with (*Owner*) being the fifth state.

By default, the operating system scheduler places threads based on the utilization of the CPUs. More precisely, a thread is created on the CPU that has the lowest CPU usage. For this reason, it is common that the threads of a single process are spread over all CPUs of a NUMA system. The scheduler can also place threads (possibly on a different CPU) after they were interrupted or sleeping. Moving a thread to another CPU can be quite expensive because the thread's cache state has

[1]`http://www.intel.de/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html`

[2]`http://www.hypertransport.org/docs/uploads/HT_General_Overview.pdf`

to be also moved and the thread's memory may not be local anymore. Therefore, the linux scheduler has a concept of scheduling domains that model the memory hierarchy and that reduce the likelihood of threads to migrate between nodes. To further control the risk of thread migrations, programming languages provide functionality to forbid the scheduler to move threads, i.e., threads can be bound to a specific CPU or even a specific core. Linux `sched_setaffinity` or functions of linux `libnuma` allow to do that. The linux tool `numactl` can also be used to force application-to-node bindings for all threads of an application. However, binding threads to specific CPUs or cores limits the scheduler's opportunities for balancing the load. This can lead to worse performance when some CPUs are overloaded while others are underutilized. For this reason, explicitly binding threads to single CPUs is a method that should be used carefully and only with the necessary application and context knowledge.

Besides thread placement, data placement is the second important aspect to consider on NUMA architectures. Naturally, data should be located close to the CPU that accesses it frequently. The default data placement policy of linux is called *first touch*. Newly allocated memory is placed local to the thread that actually uses (touches) it for the first time. This policy is especially advantageous when a single thread allocates large amounts of memory for multiple worker threads that are eventually executed on different nodes. The `libnuma` library provides functionality to directly specify a set of CPUs on which a thread allocates memory. Similar to the placement of threads, the `numactl` tool can be used for the placement of memory for an entire application. It provides standard placement policies like *preferred* allocation on a single CPU or *interleaved* and *local* allocation on a defined set of CPUs. Interleaved allocation allocates memory in a round-robin manner on the CPUs while local allocation is similar to the first touch strategy but with a restricted set of CPUs.

## 3 Synthetic Memory-Access Benchmark

To better understand the effects of the NUMA architecture on a DBMSs' query performance, we have first designed and implemented a synthetic benchmark to measure memory accesses in different situations. The benchmark's intention is to mimic a database system's memory access behavior. Therefore, our benchmark is positioned between low-level benchmarks like in [MHSM09] and full-scale application benchmarks. The results of our benchmark tool for simple setups are consistent with established memory access benchmarks. By measuring our benchmark's execution times and monitoring the operating system as well as hardware performance counters, we are able to identify, quantify, and explain effects that memory and thread placement have on the performance. As our experiments show, especially cache sharing (sharing of LLCs among readers) and cache migrations (pulling the content of the LLC to another socket after a thread has migrated) have major impacts on performance.

## 3.1 Benchmark Setup and Execution

The benchmark is based on a tool, developed by us and written in C++. We have made our code available to other researchers.[3] In the tool, threads access arrays of configurable size by repeatedly reading or writing random entries. A benchmark run consists of either reads or writes, mixed workloads are not supported. Multiple threads can share access to an array to simulate multiple clients that read or write the same data. The `libnuma` library is used to control where threads are executed and on which socket they allocate memory. It is also used to force threads to migrate to another socket during the test run.

The metric of the benchmark is latency, i.e., how long it takes to execute a random read/write operation. During the benchmark, we use a modified version of the Intel Performance Counter Monitor[4] (PCM) to count and log certain events like LLC hits and misses or the number of packets sent over any QPI link. All presented results are measured on a 4-Socket Intel Westmere EX machine (see Table 2 in Section 4 on page 10 for details on the *Intel machine*). For all tests, we report the average access time of four billion accesses.

## 3.2 Benchmark Configuration: Custom Experiments

We use our benchmark tool with different configurations to measure remote memory access costs, thread migration costs, and to isolate cache effects. All results are summarized in Table 1.

**Remote Memory Access Costs (Experiment 1)** In a first experiment, we compare random access latency to local and remote memory. We therefore configure our tool to spawn a single thread that allocates an array of 128MBs. The size of the array is chosen to be larger than the LLC of our machine so that we observe and measure memory access and QPI link utilization. The working thread repeatedly reads (respectively writes) random bytes in the array that is either allocated in local memory or on a remote socket. Results are shown in Table 1. It can be seen that executing the workload with remote memory access takes about 1.18 times longer compared to local access (26 nanoseconds for local access versus 30.8 nanoseconds for remote access).

**Thread Migration Costs (Experiment 2)** The second experiment evaluates the costs for migrating a thread from one socket to another one. The benchmark tool spawns a single thread that is either executed on one socket for the whole benchmark or migrated to another socket frequently. We investigate two different cases: 16MBs and 128MBs of memory that are accessed. In the first case, the whole

[3]`http://wwwdb.inf.tu-dresden.de/misc/user/kiefer/numa_benchmark.zip`

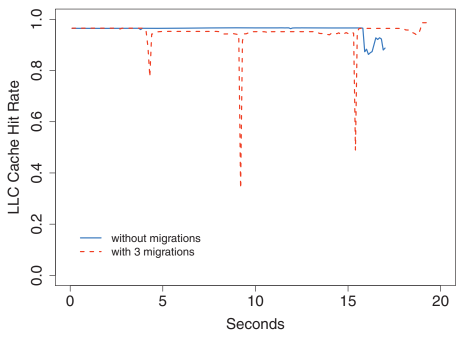
[4]`http://software.intel.com/en-us/articles/intel-performance-counter-monitor/`

array fits in the LLC of our machine. Hence, after the thread has migrated to each socket once, all sockets contain the data in their LLCs. Any further migration only costs a context switch and no consecutive remote memory access (we verify that with performance counters for LLC hits, which show a hit rate of almost 100% after 3 migrations). In the second case (128MBs), the memory does not fit in cache and hence each migration to a socket other than the first socket leads to consecutive remote memory accesses. The results in Table 1 show the isolated context migration costs for a small dataset (11.2 seconds versus 14.6 seconds for reads). For the large dataset, costs for remote memory accesses and costs for context switches overlay. The results show execution times comparable to Experiment 1, which suggests that for larger datasets, remote access costs dominate costs for context switches.

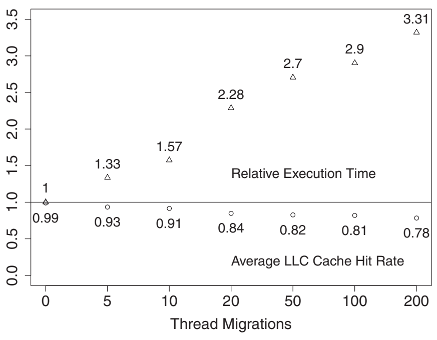
**Cache Migration Costs (Experiment 3)** The third experiment is intended to measure the costs for repeated migration of a thread followed by loading the data into the local last level cache. In contrast to the second experiment, a thread that migrates to a socket does not find any data in the last level cache (from a possible previous execution on that socket) because the data was evicted from the cache by

| Experiment | Setup | Access Times in Nanoseconds | |
|---|---|---|---|
| | | Local Access | Remote Access |
| Experiment 1 | read 128MB | 26.0 | 30.8 |
| | write 128MB | 26.0 | 31.0 |
| | | w/o Migrations | 100 Migrations |
| Experiment 2 | read 16MB | 11.2 | 14.6 |
| | write 16MB | 12.5 | 15.2 |
| | read 128MB | 25.7 | 31.6 |
| | write 128MB | 26.5 | 32.3 |
| | | w/o Migrations | 100 Migrations |
| Experiment 3 | 4 threads read 16MB | 11.1 | 31.8 |
| | 4 threads write 16MB | 11.4 | 27.0 |
| | 4 threads read 128MB | 26.4 | 40.9 |
| | 4 threads write 128MB | 27.5 | 50.3 |
| | | Co-located | Round-robin |
| Experiment 4 | 4 groups read 16MB | 10.8 | 26.3 |
| | 4 groups write 16MB | 11.0 | 21.3 |
| | 4 groups read 128MB | 30.0 | 36.2 |
| | 4 groups write 128MB | 43.5 | 47.5 |

Table 1: Synthetic benchmark results



(a) LLC hit rate with and without thread migrations



(b) Average LLC hit rate and relative execution times (normalized to execution without migrations)

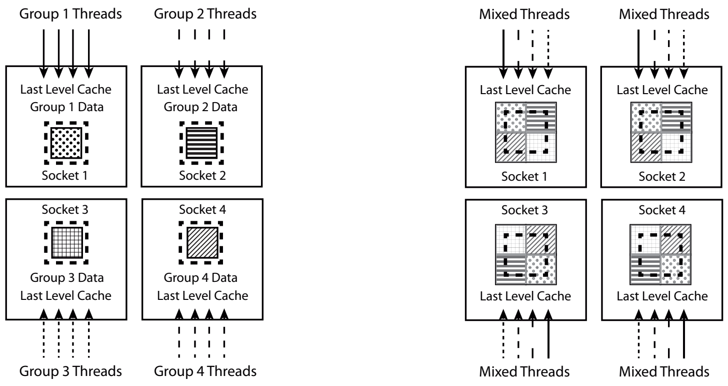
Figure 2: LLC rates for experiments with context switches

another thread in the meanwhile. The third experiment simulates the worst case costs for a thread migration.

To measure this effect, our benchmark tool spawns four threads, each reading randomly from an array of 16MBs. Without migration, each thread can fit the whole array in the local LLC and therefore read quickly from it. With thread migrations, a thread that migrates has to pull the data it reads to the socket where it migrated to. Doing so, this thread also evicts all the data that the previous thread on this socket held in cache. Hence, each migration comprises costs for the context switch and the following re-load of the last level cache. The results in Table 1 show that frequent migrations lead to about 2.8 times higher execution times compared to the experiment without migrations. The results in the table also show that for larger datasets (128MBs), the penalty for frequent thread migrations and the following re-load of the LLC is lower because most of the data does not fit in the LLC and needs to be read from memory.

We confirmed the described LLC behavior with measurements from the Intel PCM. They show that with migrations, the LLC hit rate drops for an instant after each migration because the thread has to pull the data from memory to the new LLC (shown in Figure 2a for only 3 migrations). With 200 migrations, the average hit ratio over the entire benchmark run drops to about 78% compared to 99% for the experiment without migrations. Figure 2b shows how the LLC hit rates decline while the execution times increase for a growing number of thread migrations.

**Cache Concurrency Costs (Experiment 4)** The fourth experiment shows and measures the effect of cache concurrency, which occurs when multiple threads access the same data but from different sockets. For this experiment, four instances of our benchmark tool are started in parallel. Each instance spawns four threads,



(a) Data layout with co-located threads (each LLC holds all data of one group)

(b) Data layout with distributed threads (each LLC holds one fourth of the data of all groups)

Figure 3: Cache concurrency effect for four groups of threads

which have access to the same array (comparable to multiple users accessing the same data in a database). We call the threads that share data a *thread group*.

Each thread in a group tries to load the data to the (thread-local) LLC. When all threads of a group are executed on the same socket, they beneficially share the LLC (see Figure 3a). When all threads are distributed across the sockets (see Figure 3b), they all try to load their respective data to the local LLC. This leads to constant cache line eviction of other groups' data and considerably lower cache hit rates. In the worst case scenario, where each group has a hot set of data that is about as large as a socket's last level cache, the effective LLC size is reduced by a factor as high as the number of sockets. The results in Table 1 show that for a 16MBs dataset, reading with distributed threads (round-robin) takes about 2.4 times longer than with co-located threads. The table also shows that for larger datasets, the effect is less important (reading only takes 1.2 times longer for 128MBs).

Our measurements with the Intel PCM verify the explained behavior. They show that the average LLC hit rate drops from about 98% when threads are co-located to as little as 33% with distributed threads (for the 16MBs experiment). They furthermore show that with distributed threads the data are replicated to all LLCs. With co-located threads, almost all cache hits (99%) are *unshared hits*, i.e., on cache lines in `modified` or `exclusive` state. With distributed threads on the other hand, cache line hits are mostly (94%) on cache lines that are either in `forward` or in `share` state and hence present in multiple LLCs.

### 3.3 Synthetic Benchmark Conclusion

Our results show that for certain memory access patterns such as multiple threads accessing the same data, which are common in modern database management systems, thread and memory placement can have a significant impact on performance. Our results also confirm the "common wisdom" that sharing of caches and memory with non-uniform access, although transparent to the software, can lead to performance benefits when instrumented properly or penalties when it is ignored.

Our expectation is that the demonstrated performance impact of thread and memory placement is also visible in a database management system that aims and is implemented for the highest-possible performance.

## 4 Impact of NUMA Architecture on DBMS Performance

In this section, we describe our experiments with a database management system on two different NUMA systems. Our goal is to transfer the results of our synthetic benchmark to the significantly more complex software of a DBMS. In our experiments, we concentrate on fixed placements of threads and memory and hence on remote access costs and cache concurrency effects. Incorporating thread migrations in a controllable fashion to also measure related effects was out of the scope of this paper and is considered future work. After detailing the experiment setup, we will show and interpret results from various runs of the chosen workload on both machines.

### 4.1 Experiment Setup

**Metrics and Methodology** We always report the average query throughput of multiple runs as the metric of our benchmarks. The MulTe database benchmark framework [KSL12] is used to execute multiple queries without think time in parallel and to measure the throughput of the database server. Additionally, we measure and log selected system parameters, e.g., hardware performance counters (Intel machine only, see next paragraph), to confirm a certain configuration or to help explaining certain effects. We are especially interested in core events like cache hits and misses and uncore events like QPI link utilization or events related to the cache coherence protocol.

**Machines** We use two different machines for our experiments, a 2-socket AMD architecture (*AMD machine*) and a 4-socket Intel machine (*Intel machine*). The machines were selected because they allow us to compare results for two different architectures and because they were readily available to us. An extended exper-

Table 2: Machines Used for Experiments

| **AMD machine** | **Intel machine** |
|---|---|
| 2x AMD Opteron (Istanbul) | 4x Intel Xeon (Westmere EX) |
| 2.6 GHz | 2.13 GHz |
| 6 cores per CPU | 8 cores per CPU (2 HW threads) |
| 16GB main memory per socket | 32 GB main memory per socket |
| 64KB L1, 512KB L2-cache per core | 32KB L1, 256KB L2-cache per core |
| 6MB LLC per socket | 24MB LLC per socket |
| HyperTransport @ 9.6 GB/s per link | QPI @ 12.8 GB/s per link |
| SLES 11 (2.6.32.12-0) | Ubuntu 11.10 server (3.0.0-12) |

iment with other machines, especially with more sockets, is planned as soon as respective machines are available. Table 2 summarizes both machines used for our experiments.

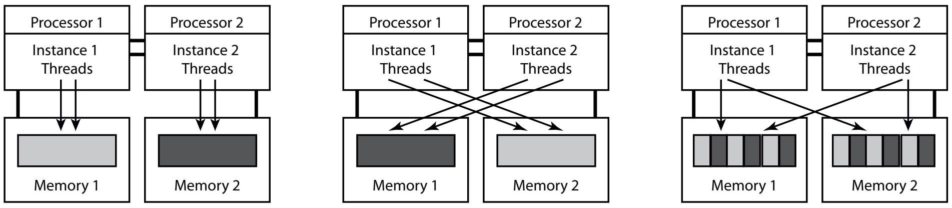
**Database Management System** All experiments are executed on the open-source DBMS MySQL Community Server (v5.5.17). The availability of the source code will allow us to implement future placement strategies in the DBMS (see Section 5). MySQL allows the use of different storage engines, including the MEMORY (HEAP) storage engine that holds all tables in memory. We use this in-memory engine (although it does not offer any persistence layer) because we think it is the best choice for our benchmark. It is a natural fit for data that resides in memory all the time and it scales better with the number of concurrent users compared to, e.g., the InnoDB engine.

**Workload** We base our experiments on the TPC-H database schema and workload [TPC12]. The scaling factor for the database is 1, i.e., 1GB of raw data. We use a subset of the official benchmark queries together with some synthetic queries that execute common data access patterns. All experiments are restricted to read-only queries.

### 4.2 Data Partitioning

Our experiments are motivated by the observation that in DBMS data partitions are accessed independently from one another. To mimic such partitions, we install multiple MySQL Server instances, e.g., one for each user group that accesses the data. Each instance contains a database with the TPC-H schema. To simplify our experiments, we use one instance per socket on each machine.

Each instance of MySQL runs as a separate process. Therefore, we are able to explicitly control thread placement and memory allocation on instance level, i.e.,



(a) Aligned setup (b) 1-ahead setup (c) Interleaved setup

Figure 4: Thread and memory assignment strategies

process level, with the `numactl` tool. To show the potential of the NUMA effects on database server performance, we compare three different setups with the *default* case: *aligned*, *1-ahead*, and *interleave*. The default case does not take the NUMA architecture into consideration (beyond what the operating system does). In the aligned setup—the expected best case—we bind each instance's threads explicitly to one socket and allocate all the memory on the same socket (shown in Figure 4a). In the 1-ahead setup—the expected worst case—we bind threads to one socket and allocate memory on the other one (AMD machine) or the next one[5] (Intel machine) (shown in Figure 4b). The third setup (*interleave*) binds threads to a fixed socket but interleaves memory on all sockets (shown in Figure 4c).

For each instance of MySQL, we use as many parallel connections to query the database as there are cores (AMD machine) or hardware threads (Intel machine). We compute the sum of the throughputs of all instances to get the overall system throughput. We measure a single query at a time (multiple executions of the same query with different parameter values) to minimize the side effects that may occur in a mixed workload.

## 4.3 Results and Interpretation

The results of our experiments are shown in Figure 5 for the AMD machine and in Figure 6 for the Intel machine. Both charts show the query throughputs of the aligned, 1-ahead, and interleave setups, normalized to the default case. Dotted lines separate TPC-H queries from the synthetic ones.

It can be seen in Figure 5 that on the AMD machine the aligned setup either is as fast as the default setup or outperforms it by as much as 18%. On average, aligning threads and memory leads to improvements between 10% and 15% which is notable, given the simple modification that is necessary to achieve the improvement. Also, one expects that the effects will be amplified in systems with more sockets. The

[5]Since all sockets are fully connected in our machine, the *next* socket, can be an arbitrary one. We use the current socket ID, incremented by one and modulo the number of sockets as the *next* socket.

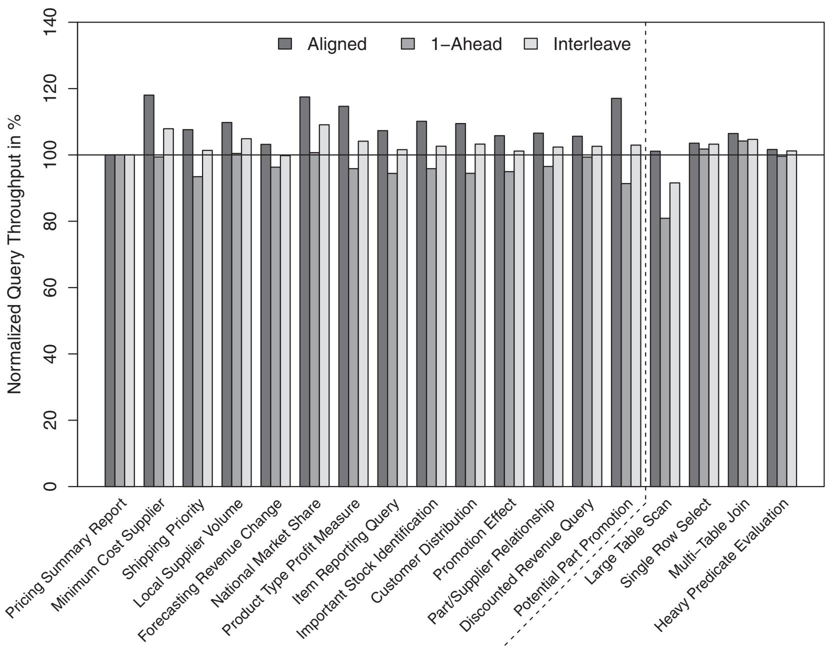


Figure 5: Normalized query throughput on the AMD machine

chart furthermore shows that placing threads and memory on different sockets (1-ahead) often leads to a degradation of the query throughput. This is to be expected, given the higher amount of communication. It is interesting to see that even the naive approach of interleaving the memory on all sockets leads to a slight improvement in some cases and no degradation for any of the queries. We account this to the threads of each instance being assigned to a fixed socket and the resulting improved LLC usage. We will revisit this effect further down when we analyze the results on the Intel machine.

Figure 6 shows the results for the Intel machine. The throughput improvement when aligning threads and memory can be as high as 75%. Interestingly, even the 1-ahead setup does not degrade query throughput, but slight improves it in most cases. The interleaved setup lies between aligned and 1-ahead. The result for 1-ahead is counter-intuitive, given the communication that is needed to access the data on the remote socket. We believe that improvements in LLC usage—caused by assigning each instances's threads to a fixed socket—lead to the better query throughput. The last result we take from both experiments is that only the first synthetic query stands out as it shows the worst performance degradation among all queries with the 1-ahead and interleaved setups. The other synthetic queries show average improvements. Finding better synthetic queries that show the connection between access pattern and performance improvements is subject to future work.

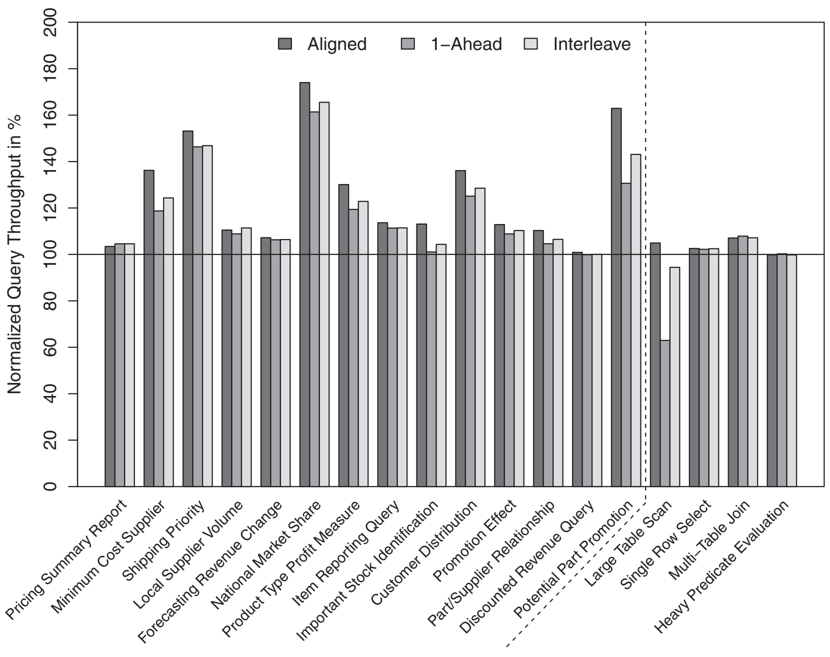
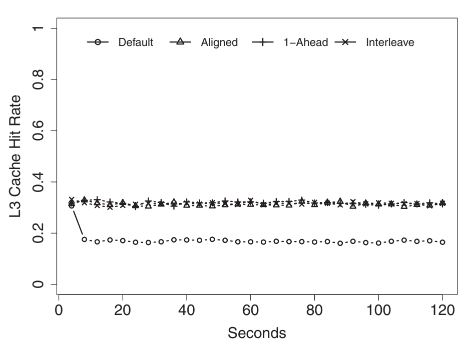
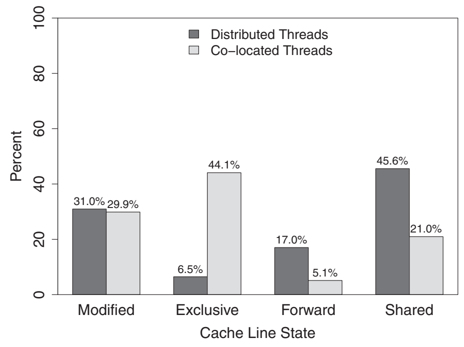


Figure 6: Normalized query throughput on the Intel machine

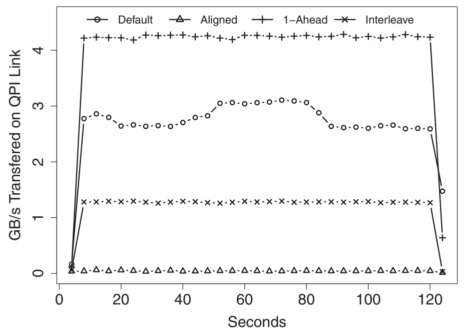
**Cache Behavior:** All three setups (aligned, 1-ahead, and interleave) have in common that all threads that access the same data are executed on the same socket. Hence, data that one thread reads is more likely to stay in the LLC for another thread that reads the same data. In the default case (without `numactl`), threads of all instances are executed on all sockets and even migrate between sockets from time to time (we have verified that with data monitored in the `/proc` pseudo-filesystem). This is comparable to the *Cache Concurrency Costs* experiment (Experiment 4) of our synthetic benchmark: multiple threads from one instance run on different sockets and therefore load the same data to their local LLCs. The effective size of the cache is considerably reduced due to data that is stored redundantly in several caches. To support this claim, we again monitored the LLC hit rate and cache line states for one query execution. Figures 7a and 7b show the respective results. The LLC hit rate for the default case is lower (about 20%) than for all other setups (about 35%). The second chart, shown in Figure 7b, confirms the cache concurrency. In the default case (distributed threads), memory accesses often hit shared cache lines, i.e., cache lines in *shared* or *forward* state (together 62.6%). There are fewer hits on unshared cache lines, i.e., in *exclusive* or *modified* state (together 37.4%). The setups with co-located threads (aligned, 1-ahead, and interleave all use `numactl` with `cpubind`) show significantly higher hit rates on unshared cache lines (together 74%) while *shared* and *forward* lines are hit only with a rate of 26%.



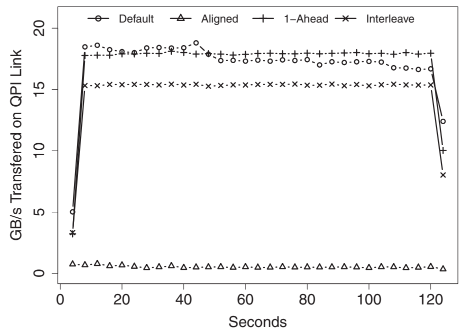
(a) LLC hit rate



(b) LLC-line states (% of all hits)



(c) Single QPI link utilization



(d) Global QPI link utilization

Figure 7: L3-cache usage and QPI link utilization

**QPI Link Usage:** All three setups (aligned, 1-ahead, and interleave) show the same LLC but still differ in the query throughput results, likely caused by differences in communication between sockets. We have analyzed the QPI link utilization and show the results for a single query and for a single QPI link (Figure 7c); and aggregated for all QPI links (Figure 7d). One can see that the throughput on a single QPI link follows the intuition from the given setups. The aligned setup does not need any inter-socket communication while the 1-ahead setup puts the most pressure on a single QPI link. The interleave setup lies somewhere in between. Interestingly, the default case that uses the first-touch policy to allocate memory needs more communication than the interleave setup. The aggregated QPI link utilization over all sockets and all links looks slightly different (Figure 7d). Interesting here, the default case causes about as much communication as the expected worst case (1-ahead). This and the improved LLC behavior can explain why on the Intel machine, even the 1-ahead setup is often slightly faster than the default setup.

## 5 Towards Database-Level Scheduling

Our experiments were able to show the potential of NUMA awareness in DBMSs which leads to many opportunities for future work. Control mechanisms for thread placement and memory allocation need to be implemented in the DBMS (instead of using `numactl`). Only there, the necessary knowledge about data partitions is available and only there it is possible to allocate threads and memory in a fine-grained fashion. Since DBMSs are usually deployed on dedicated machines, it is safe to assume that all important threads and memory consumers of the whole system are known to the DBMS. We have started to modify the MySQL DBMS to support thread placement based on the data that an application will access but detailed reports are subject to future work.

All data partitions (i.e., instances) were evenly queried in our experiments. A NUMA-aware DBMS will have to deal with skewed loads and dynamic changes in the query frequency of certain partitions. The workload used in the experiments contained only single queries (no mixed workload) of a certain type. Database workloads are traditionally classified as either being OLTP (online transaction processing) or OLAP (online analytical processing). OLTP workload on the one hand presents a high frequency of short queries that often access only a small amount of data. The performance of OLTP queries is dominated by the latency of the data access. OLAP queries on the other hand are usually less frequent. They access and aggregate large amounts of data and not seldom read whole tables in the process. Consequently, read operations are rather sequential and the performance is dominated by the available bandwidth. Recent trends in database management systems have weakened the separation of OLTP and OLAP systems and many systems are nowadays used to answer both types of queries. Hence, a NUMA-aware DBMS that takes care of thread and memory placement needs to take both query types and therefore very different access patterns into account.

Once a DBMS is equipped with tools to allocate memory and assign threads based on data partitioning and knowledge about the architecture, the next step will be to develop a cost model to evaluate thread and memory placement. Based on the cost model, the DBMS needs to decide where to execute threads and where to allocate memory. Using more resources on multiple sockets has to be weighted against having resources local to the execution.

Figure 8 shows a possible architecture for a NUMA-aware DBMS. The DBMS has the necessary partitioning information and sends thread scheduling and memory placement policies to the operating system. The operating system on the other hand provides information about the NUMA architecture (like the memory topology) and an interface to scheduling and memory placement decisions.

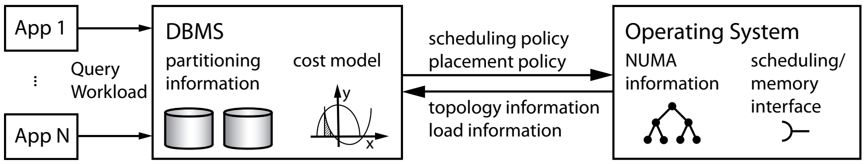


Figure 8: Architecture of a NUMA-aware DBMS

## 6 Related Work

Related work from different communities is relevant for the given topic. We distinguish work that concentrates on NUMA in general from papers that investigate DBMSs on modern hardware, possibly but not necessarily NUMA architectures.

### 6.1 Work related to NUMA architectures

The consequences of non-uniform memory access as well as thread-placement in multicore systems have been studied in the operating systems and high-performance computing communities for many years. Among the more recent results are Molka and Hackenberg et al. who have investigated the low-level memory performance and cache coherence effects at the granularity of single cache lines [HMN09, MHSM09]. McCurdy and Vetter give a general overview on finding and fixing NUMA-related performance problems [MV10] and Blagodurov et al. propose a NUMA-aware scheduler and user-level scheduling on NUMA systems, though not specific to a certain application [BZFD10, BF11]. The problem of mapping threads to cores (and possibly co-locating them) to reduce communication and increase data sharing has been investigated by, e.g., Tam et al. [TAS07] and Tang et al. [TMV+11].

### 6.2 Work related to DBMSs on modern hardware

There is little information available on NUMA awareness of commercial DBMSs. From blogs[6] and personal communication, we know that Microsoft SQL Server and the Oracle DBMS are NUMA-aware. In SQL Server, it is possible to direct user connections to certain nodes and the system automatically partitions buffer pools and prefers local memory for temporal and session data.

The research community has analyzed the impact of modern (multicore) processors on existing DBMSs [ADHW99, HPJ+07]. To overcome the general shortcomings of these systems, i.e., their poor scalability with the number of CPUs and cores,

[6]E.g., `http://blogs.msdn.com/b/slavao/archive/2005/08/02/446648.aspx` or `http://kevinclosson.wordpress.com/2009/05/14/you-buy-a-numa-system-oracle-says-disable-numa-what-gives-part-ii/`

different solutions have been proposed. All solutions can roughly be categorized as *distributed systems*, which treat multiple CPUs/cores as if they were a distributed system, and *shared systems*, which try to overcome scalability issues by reducing communication and contention.

**Distributed system approaches** Distributed database systems have been investigated first in the late 80s and early 90s [DGS$^+$90, AvdBF$^+$92]. Many aspects of these early distributed DBMSs can be used to improve scalability on modern systems. The Multimed system at ETH-Zürich [SSGA11] deploys multiple instances of an existing database engine (e.g., PostgreSQL or MySLQ) on non-overlapping subsets of all cores. A master database on one partition receives all updates and propagates them asynchronously to satellite databases, which in turn receive read-only load. H-Store [SMA$^+$07] partitions the data horizontally over nodes and cores and executes transactions sequentially (single-threaded) on each core. Hence, there is no synchronization of access to each partition needed. Depending on what data a transaction accesses, multiple partitions need to communicate. The HyPer system [KN11] is an in-memory system that uses processor-inherent lazy copy-on-write and virtual memory management to support OLTP and parallel OLAP transactions. Multiple read-only queries can be executed at a time, but writing requests are executed sequentially to ensure consistency without synchronization. Also in the context of distributed (or shared-nothing) database systems are works like Schism by Curino et al. [CJZM10] that aim at reducing the need for inter-partition communication by means of workload-aware partitioning of the data.

**Shared system approaches** An example for a shared-everything system is the Shore-MT system by Johnson et al. [JPH$^+$09]. After identifying bottlenecks in existing database storage managers, the authors develop a multithreaded, scalable storage manager by optimizing locks, latches, and synchronization and thereby reducing contention. Physiological partitioning, a compromise between the distributed and the shared approach, was proposed by Pandis and Tözün et al. [PTJA11, TPJA12]. While the data is still shared, a multi-rooted B+ tree is used to partition the data which avoids costly page-latches. The DORA-system, also by Pandis et al. [PJHA10], binds threads to disjoint sets of the data and decomposes transactions to smaller actions according to the data they access. Each thread has private locking mechanism to control access to the data it owns. A recent work by Albutiu et al. [AKN12] analyzes sort-merge joins in multi-core database systems. The authors recognize the NUMA characteristics of such systems and motivate their algorithm design with micro-benchmarks related to our synthetic benchmark.

A recent work that falls between the strictly shared or distributed approach and that also aims at demonstrating and utilizing the NUMA effects in a DMBS was published by Porobic et al. [PPB$^+$12]. The authors perform a detailed analysis of different shared and distributed deployments in NUMA systems. To show the performance impact of the NUMA architecture on a DBMS, they use ShoreMT as a scalable storage manager and TPC-C as an OLTP workload.

## 7 Conclusion

We showed with our synthetic benchmark that remote memory access and cache usage related to thread placement and thread movement heavily influence performance on NUMA systems. We furthermore showed that database management systems—when executed on NUMA systems—can benefit from careful thread and memory placement. Executing an idealized workload showed the potential for throughput improvements of up to 75% compared to the naive execution that ignores the characteristics of the NUMA system. We had to experience that reliable experiments with a DBMS are extremely hard to conduct. MySQL, used as we did for our studies, suffers from different scalability and reliability issues so that we had to revise our experiments many times to isolate and quantify the NUMA effects. Nevertheless, our experiments have confirmed that not only the non-uniform main memory access is responsible for performance differences but also the fact that each node has its own cache hierarchy. This leads to disadvanteous cache evictions when threads migrate between nodes or when threads access the same data from different nodes.

## References

[ADHW99] Anastasia Ailamaki, David J Dewitt, Mark D Hill, and David A Wood. DBMSs On A Modern Processor: Where Does Time Go? In *VLDB '99*, Edinburgh, Scottland, 1999. VLDB Endowment.

[AKN12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. In *VLDB '12*, volume 5, Istanbul, Turkey, 2012. VLDB Endowment.

[AvdBF$^+$92] Peter MG Apers, Carel A van den Berg, Jan Flokstra, Paul WPJ Grefen, Martin L Kersten, and Annita N Wilschut. PRISMA/DB: A Parallel, Main Memory Relational DBMS. *Knowledge and Data Engineering*, 4(6):541–554, 1992.

[BCD$^+$11] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *ICDE '11*, pages 1255–1263, Hannover, Germany, 2011. IEEE.

[BF11] Sergey Blagodurov and Alexandra Fedorova. User-level scheduling on NUMA multicore systems under Linux. In *Linux Symposium*, pages 81–91, Ottawa, Canada, 2011.

[BZFD10] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Mohammad Dashti. A Case for NUMA-aware Contention Management on Multicore Systems. In *PACT'10*, Vienna, Austria, 2010.

[CJMB11] Carlo Curino, Evan P.C. Jones, Sam Madden, and Hari Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD '11*, pages 313–324, Athens, Greece, 2011. ACM.

[CJZM10] Carlo Curino, Evan Jones, Y. Zhang, and Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In *VLDB '10*, volume 3, pages 48–57, Singapore, China, 2010. VLDB Endowment.

[DGS+90] David J Dewitt, Shahram Ghandeharizadeh, Donovan Schneider, Allen Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[HMN09] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO '09*, pages 413–422, New York, New York, USA, 2009.

[HPJ+07] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju G Mancheril, Anastasia Ailamaki, and Babak Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR '07*, pages 79–87, Asilomar, California, USA, 2007.

[JPH+09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT '09*, pages 24–35, Saint Petersburg, Russia, 2009. ACM.

[KL11] Tim Kiefer and Wolfgang Lehner. Private Table Database Virtualization for DBaaS. In *UCC '11*, volume 1, pages 328–329, Melbourne, Australia, December 2011. IEEE.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE '11*, pages 195–206, Hannover, Germany, 2011. IEEE.

[KSL12] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. MulTe: A Multi-Tenancy Database Benchmark Framework. In *TPCTC '12*, Istanbul, Turkey, 2012.

[MHSM09] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *PACT'09*, pages 261–270, Raleigh, North Carolina, USA, September 2009. IEEE.

[MV10] Collin McCurdy and Jeffrey Vetter. Memphis: Finding and Fixing NUMA-related Performance Problems on Multi-core Platforms. In *ISPASS'10*, pages 87–96, White Plains, NY, USA, 2010. IEEE.

[PJHA10] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-Oriented Transaction Execution. In *VLDB '10*, volume 3, Singapore, China, 2010. VLDB Endowment.

[PPB+12] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. OLTP on Hardware Islands. In *VLDB '12*, pages 1447–1458, Istanbul, Turkey, 2012. VLDB Endowment.

[PTJA11] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: Page Latch-free Shared-everything OLTP. In *VLDB '11*, Seattle, Washington, USA, 2011. VLDB Endowment.

[SMA+07] Michael Stonebraker, Sam Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB '07*, Vienna, Austria, 2007. VLDB Endowment.

[SSGA11] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *EuroSys '11*, page 14, Salzburg, Austria, 2011. ACM Press.

[TAS07] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering : Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys '07*, Lisboa, Portugal, 2007.

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *ISCA '11*, San Jose, California, USA, 2011.

[TPC12] TPC. Transaction Processing Performance Council, TPC-H, 2012.

[TPJA12] Pinar Tözün, Ippokratis Pandis, Ryan Johnson, and Anastasia Ailamaki. Scalable and dynamically balanced shared-everything OLTP with physiological partitioning. *The VLDB Journal*, June 2012.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The Implementation and Performance of Compressed Databases. In *SIGMOD '00*, pages 55–67, New York, NY, USA, 2000. ACM.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE '06*, page 59, Washington, DC, USA, 2006. IEEE.