Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis* Peter Boncz† Alfons Kemper* Thomas Neumann*

* Technische Universität München † CWI

* {leis,kemper,neumann}@in.tum.de † p.boncz@cwi.nl

ABSTRACT

With modern computer architecture evolving, two problems conspire against the state-of-the-art approaches in parallel query execution: (i) to take advantage of many-cores, all query work must be distributed evenly among (soon) hundreds of threads in order to achieve good speedup, yet (ii) dividing the work evenly is difficult even with accurate data statistics due to the complexity of modern out-of-order cores. As a result, the existing approaches for "plan-driven" parallelism run into load balancing and context-switching bottlenecks, and therefore no longer scale. A third problem faced by many-core architectures is the decentralization of memory controllers, which leads to Non-Uniform Memory Access (NUMA).

In response, we present the "morsel-driven" query execution framework, where scheduling becomes a fine-grained run-time task that is NUMA-aware. Morsel-driven query processing takes small fragments of input data ("morsels") and schedules these to worker threads that run entire operator pipelines until the next pipeline breaker. The degree of parallelism is not baked into the plan but can *elastically* change during query execution, so the dispatcher can react to execution speed of different morsels but also adjust resources dynamically in response to newly arriving queries in the workload. Further, the dispatcher is aware of data locality of the NUMA-local morsels and operator state, such that the great majority of executions takes place on NUMA-local memory. Our evaluation on the TPC-H and SSB benchmarks shows extremely high absolute performance and an average speedup of over 30 with 32 cores.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

Keywords

Morsel-driven parallelism; NUMA-awareness

1. INTRODUCTION

The main impetus of hardware performance improvement nowadays comes from increasing multi-core parallelism rather than from speeding up single-threaded performance [2]. By SIGMOD 2014 Intel's forthcoming mainstream server Ivy Bridge EX, which can run 120 concurrent threads, will be available. We use the term *many-core* for such architectures with tens or hundreds of cores.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
ACM 978-1-4503-2376-5/14/06.
http://dx.doi.org/10.1145/2588555.2610507 .

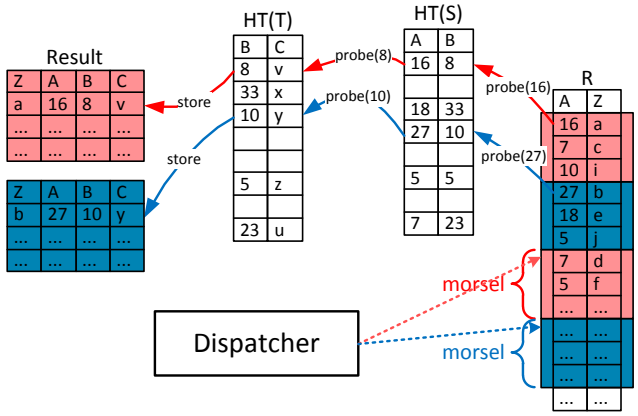


Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

At the same time, increasing main memory capacities of up to several TB per server have led to the development of main-memory database systems. In these systems query processing is no longer I/O bound, and the huge parallel compute resources of many-cores can be truly exploited. Unfortunately, the trend to move memory controllers into the chip and hence the decentralization of memory access, which was needed to scale throughput to huge memories, leads to non-uniform memory access (NUMA). In essence, the computer has become a network in itself as the access costs of data items varies depending on which chip the data and the accessing thread are located. Therefore, many-core parallelization needs to take RAM and cache hierarchies into account. In particular, the NUMA division of the RAM has to be considered carefully to ensure that threads work (mostly) on NUMA-local data.

Abundant research in the 1990s into parallel processing led the majority of database systems to adopt a form of parallelism inspired by the Volcano [12] model, where operators are kept largely unaware of parallelism. Parallelism is encapsulated by so-called "exchange" operators that route tuple streams between multiple threads each executing identical pipelined segments of the query plan. Such implementations of the Volcano model can be called *plan-driven*: the optimizer statically determines at query compile-time how many threads should run, instantiates one query operator plan for each thread, and connects these with exchange operators.

In this paper we present the adaptive *morsel-driven* query execution framework, which we designed for our main-memory database system HyPer [16]. Our approach is sketched in Figure 1 for the three-way-join query $R \bowtie_A S \bowtie_B T$. Parallelism is achieved

by processing each pipeline on different cores in parallel, as indicated by the two (upper/red and lower/blue) pipelines in the figure. The core idea is a *scheduling* mechanism (the "dispatcher") that allows flexible parallel execution of an operator pipeline, that can change the parallelism degree even during query execution. A query is divided into segments, and each executing segment takes a morsel (e.g, 100,000) of input tuples and executes these, materializing results in the next pipeline breaker. The morsel framework enables NUMA local processing as indicated by the color coding in the figure: A thread operates on NUMA-local input and writes its result into a NUMA-local storage area. Our dispatcher runs a fixed, machine-dependent number of threads, such that even if new queries arrive there is no resource over-subscription, and these threads are pinned to the cores, such that no unexpected loss of NUMA locality can occur due to the OS moving a thread to a different core.

The crucial feature of morsel-driven scheduling is that task distribution is done at run-time and is thus *fully elastic*. This allows to achieve perfect load balancing, even in the face of uncertain size distributions of intermediate results, as well as the hard-to-predict performance of modern CPU cores that varies even if the amount of work they get is the same. It is elastic in the sense that it can handle workloads that change at run-time (by reducing or increasing the parallelism of already executing queries in-flight) and can easily integrate a mechanism to run queries at different priorities.

The morsel-driven idea extends from just scheduling into a complete query execution framework in that all physical query operators must be able to execute morsel-wise in parallel in all their execution stages (e.g., both hash-build and probe), a crucial need for achieving many-core scalability in the light of Amdahl's law. An important part of the morsel-wise framework is awareness of data locality. This starts from the locality of the input morsels and materialized output buffers, but extends to the state (data structures, such as hash tables) possibly created and accessed by the operators. This state is shared data that can potentially be accessed by any core, but does have a high degree of NUMA locality. Thus morsel-wise scheduling is flexible, but strongly favors scheduling choices that maximize NUMA-local execution. This means that remote NUMA access only happens when processing a few morsels per query, in order to achieve load balance. By accessing local RAM mainly, memory latency is optimized and cross-socket memory traffic, which can slow other threads down, is minimized.

In a pure Volcano-based parallel framework, parallelism is hidden from operators and shared state is avoided, which leads to plans doing on-the-fly data partitioning in the exchange operators. We argue that this does not always lead to the optimal plan (as partitioning effort does not always pay off), while the locality achieved by on-the-fly partitioning can be achieved by our locality-aware dispatcher. Other systems have advocated per-operator parallelization [21] to achieve flexibility in execution, but this leads to needless synchronization between operators in one pipeline segment. Nevertheless, we are convinced that the morsel-wise framework can be integrated in many existing systems, e.g., by changing the implementation of exchange operators to encapsulate morsel-wise scheduling, and introduce e.g., hash-table sharing. Our framework also fits systems using Just-In-Time (JIT) code compilation [19, 25] as the generated code for each pipeline occurring in the plan, can subsequently be scheduled morsel-wise. In fact, our HyPer system uses this JIT approach [25].

In this paper we present a number of related ideas that enable efficient, scalable, and elastic parallel processing. The main contribution is an architectural blueprint for a query engine incorporating the following:

- **Morsel-driven query execution** is a new parallel query evaluation framework that fundamentally differs from the traditional Volcano model in that it distributes work between threads dynamically using work-stealing. This prevents unused CPU resources due to load imbalances, and allows for *elasticity*, i.e., CPU resources can be reassigned between different queries at any time.
- A set of fast **parallel algorithms** for the most important relational operators.
- A systematic approach to integrating **NUMA-awareness** into database systems.

The remainder of this paper is organized as follows. Section 2 is devoted to a detailed discussion of pipeline parallelization and the fragmentation of the data into morsels. In Section 3 we discuss the dispatcher, which assigns tasks (pipeline jobs) and morsels (data fragments) to the worker threads. The dispatcher enables the full elasticity which allows to vary the number of parallel threads working on a particular query at any time. Section 4 discusses algorithmic and synchronization details of the parallel join, aggregation, and sort operators. The virtues of the query engine are evaluated in Section 5 by way of the entire TPC-H query suite. After discussing related work in order to point out the novelty of our parallel query engine architecture in Section 6, we conclude the paper.

2. MORSEL-DRIVEN EXECUTION

Adapted from the motivating query of the introduction, we will demonstrate our parallel pipeline query execution on the following example query plan:

$$\sigma_{...}(R) \bowtie_A \sigma_{...}(S) \bowtie_B \sigma_{...}(T)$$

Assuming that R is the largest table (after filtering) the optimizer would choose R as probe input and build (team) hash tables of the other two, S and T. The resulting algebraic query plan (as obtained by a cost-based optimizer) consists of the three pipelines illustrated on the left-hand side of Figure 2:

1. Scanning, filtering and building the hash table $HT(T)$ of base relation T,
2. Scanning, filtering and building the hash table $HT(S)$ of argument S,
3. Scanning, filtering R and probing the hash table $HT(S)$ of S and probing the hash table $HT(T)$ of T and storing the result tuples.

HyPer uses Just-In-Time (JIT) compilation to generate highly efficient machine code. Each pipeline segment, including all operators, is compiled into one code fragment. This achieves very high raw performance, since interpretation overhead as experienced by traditional query evaluators, is eliminated. Further, the operators in the pipelines do not even materialize their intermediate results, which is still done by the already much more efficient vector-at-a-time evaluation engine of Vectorwise [34].

The morsel-driven execution of the algebraic plan is controlled by a so called *QEPobject* which transfers executable pipelines to a dispatcher – cf. Section 3. It is the *QEPobject*'s responsibility to observe data dependencies. In our example query, the third (probe) pipeline can only be executed after the two hash tables have been built, i.e., after the first two pipelines have been fully executed. For each pipeline the *QEPobject* allocates the temporary storage areas into which the parallel threads executing the pipeline write their results. After completion of the entire pipeline the temporary storage areas are logically re-fragmented into equally sized

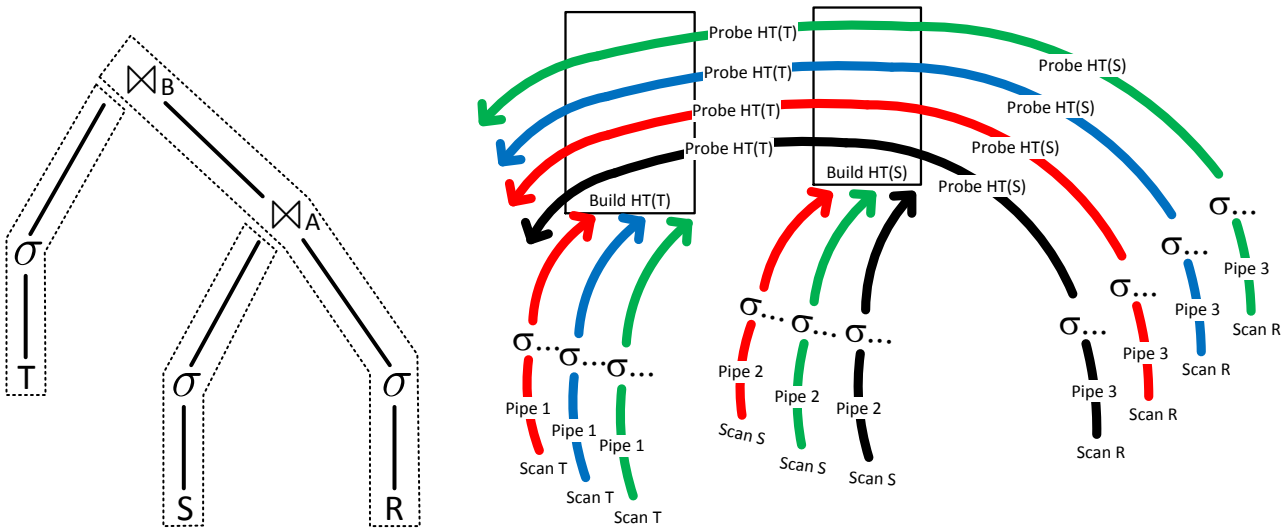


Figure 2: Parallellizing the three pipelines of the sample query plan: (left) algebraic evaluation plan; (right) three- respectively four-way parallel processing of each pipeline

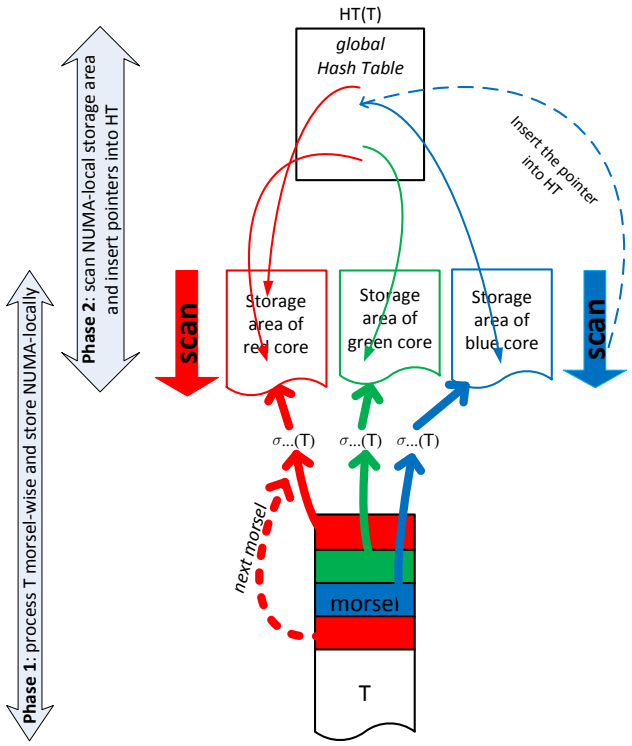


Figure 3: NUMA-aware processing of the build-phase

morsels; this way the succeeding pipelines start with new homogeneously sized morsels instead of retaining morsel boundaries across pipelines which could easily result in skewed morsel sizes. The number of parallel threads working on any pipeline at any time is bounded by the number of hardware threads of the processor. In order to write NUMA-locally and to avoid synchronization while writing intermediate results the *QEPobject* allocates a storage area for each such thread/core for each executable pipeline.

The **parallel** processing of the pipeline for filtering T and building the hash table $HT(T)$ is shown in Figure 3. Let us concentrate on the processing of the first phase of the pipeline that filters input T and stores the "surviving" tuples in temporary storage areas. In our figure three parallel threads are shown, each of which operates on one *morsel* at a time. As our base relation T is stored "morsel-wise" across a NUMA-organized memory, the scheduler assigns, whenever possible, a morsel located on the same socket where the thread is executed. This is indicated by the coloring in the figure: The red thread that runs on a core of the red socket is assigned the task to process a red-colored morsel, i.e., a small fragment of the base relation T that is located on the red socket. Once, the thread has finished processing the assigned morsel it can either be delegated (dispatched) to a different task or it obtains another morsel (of the same color) as its next task. As the threads process one morsel at a time the system is fully elastic. The degree of parallelism (MPL) can be reduced or increased at any point (more precisely, at morsel boundaries) while processing a query.

The logical algebraic pipeline of (1) scanning/filtering the input T and (2) building the hash table is actually broken up into two physical processing pipelines marked as phases on the left-hand side of the figure. In the first phase the filtered tuples are inserted into NUMA-local storage areas, i.e., for each core there is a separate storage area in order to avoid synchronization. To preserve NUMA-locality in further processing stages, the storage area of a particular core is locally allocated on the same socket.

After all base table morsels have been scanned and filtered, in the second phase these storage areas are scanned – again by threads located on the corresponding cores – and pointers are inserted into the hash table. Segmenting the logical hash table building pipeline into two phases enables perfect sizing of the global hash table because after the first phase is complete, the exact number of "surviving" objects is known. This (perfectly sized) global hash table will be probed by threads located on various sockets of a NUMA system; thus, to avoid contention, it should not reside in a particular NUMA-area and is therefore is interleaved (spread) across all sockets. As many parallel threads compete to insert data into this hash table, a lock-free implementation is essential. The implementation details of the hash table are described in Section 4.2.

After both hash tables have been constructed, the probing pipeline can be scheduled. The detailed processing of the probe pipeline is shown in Figure 4. Again, a thread requests work from the dispatcher which assigns a morsel in the corresponding NUMA partition. That is, a thread located on a core in the red NUMA partition is assigned a morsel of the base relation R that is located on the cor-

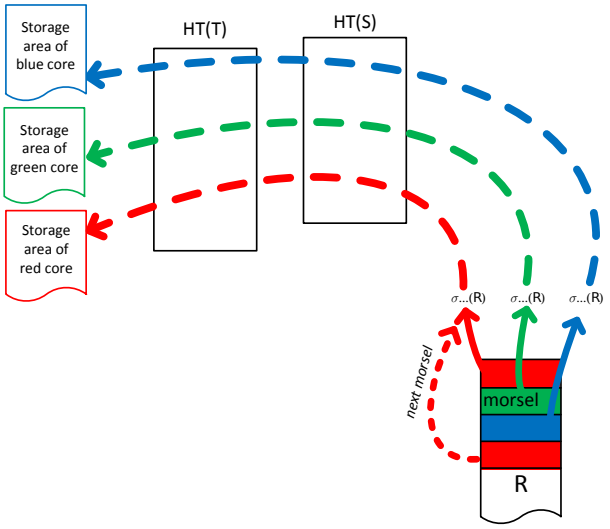


Figure 4: Morsel-wise processing of the probe phase

responding "red" NUMA socket. The result of the probe pipeline is again stored in NUMA local storage areas in order to preserve NUMA locality for further processing (not present in our sample query plan).

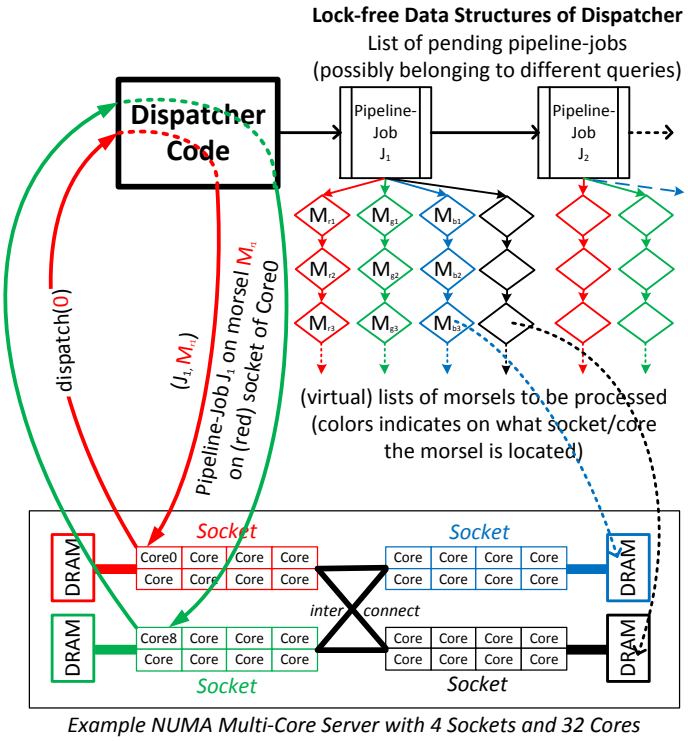
In all, morsel-driven parallelism executes multiple pipelines in parallel, which is similar to typical implementations of the Volcano model. Different from Volcano, however, is the fact that the pipelines are not independent. That is, they share data structures and the operators are aware of parallel execution and must perform synchronization (through efficient lock-free mechanisms – see later). A further difference is that the number of threads executing the plan is fully elastic. That is, the number may differ not only between different pipeline segments, as shown in Figure 2, but also inside the same pipeline segment *during* query execution – as described in the following.

3. DISPATCHER: SCHEDULING PARALLEL PIPELINE TASKS

The *dispatcher* is controlling and assigning the compute resources to the parallel pipelines. This is done by assigning tasks to worker threads. We (pre-)create one worker thread for each hardware thread that the machine provides and permanently bind each worker to it. Thus, the level of parallelism of a particular query is not controlled by creating or terminating threads, but rather by assigning them particular tasks of possibly different queries. A task that is assigned to such a worker thread consists of a pipeline job and a particular morsel on which the pipeline has to be executed. Preemption of a task occurs at morsel boundaries – thereby eliminating potentially costly interrupt mechanisms. We experimentally determined that a morsel size of about 100,000 tuples yields good tradeoff between instant elasticity adjustment, load balancing and low maintenance overhead.

There are three main goals for assigning tasks to threads that run on particular cores:

1. Preserving (NUMA-)locality by assigning data morsels to cores on which the morsels are allocated
2. Full elasticity concerning the level of parallelism of a particular query



Example NUMA Multi-Core Server with 4 Sockets and 32 Cores

Figure 5: Dispatcher assigns pipeline-jobs on morsels to threads depending on the core

3. Load balancing requires that all cores participating in a query pipeline finish their work at the same time in order to prevent (fast) cores from waiting for other (slow) cores[1].

In Figure 5 the architecture of the *dispatcher* is sketched. It maintains a list of pending pipeline jobs. This list only contains pipeline jobs whose prerequisites have already been processed. E.g., for our running example query the build input pipelines are first inserted into the list of pending jobs. The probe pipeline is only inserted after these two build pipelines have been finished. As described before, each of the active queries is controlled by a *QEPobject* which is responsible for transferring executable pipelines to the dispatcher. Thus, the dispatcher maintains only lists of pipeline jobs for which all dependent pipelines were already processed. In general, the dispatcher queue will contain pending pipeline jobs of different queries that are executed in parallel to accommodate inter-query parallelism.

3.1 Elasticity

The fully elastic parallelism, which is achieved by dispatching jobs "a morsel at a time", allows for intelligent scheduling of these inter-query parallel pipeline jobs depending on a quality of service model. It enables to gracefully decrease the degree of parallelism of, say a long-running query Q_l at any stage of processing in order to prioritize a possibly more important interactive query Q_+. Once the higher prioritized query Q_+ is finished, the pendulum swings back to the long running query by dispatching all or most cores to tasks of the long running query Q_l. In Section 5.4 we demonstrate this dynamic elasticity experimentally. In our current implementation all queries have the same priority, so threads are distributed

[1]This assumes that the goal is to minimize the response time of a particular query. Of course, an idle thread could start working on another query otherwise.

equally over all active queries. A priority-based scheduling component is under development but beyond the scope of this paper.

For each pipeline job the dispatcher maintains lists of pending morsels on which the pipeline job has still to be executed. For each core a separate list exists to ensure that a work request of, say, Core 0 returns a morsel that is allocated on the same socket as Core 0. This is indicated by different colors in our architectural sketch. As soon as Core 0 finishes processing the assigned morsel, it requests a new task, which may or may not stem from the same pipeline job. This depends on the prioritization of the different pipeline jobs that originate from different queries being executed. If a high-priority query enters the system it may lead to a decreased parallelism degree for the current query. Morsel-wise processing allows to reassign cores to different pipeline jobs without any drastic interrupt mechanism.

3.2 Implementation Overview

For illustration purposes we showed a (long) linked list of morsels for each core in Figure 5. In reality (i.e., in our implementation) we maintain storage area boundaries for each core/socket and segment these large storage areas into morsels on demand; that is, when a core requests a task from the dispatcher the next morsel of the pipeline argument's storage area on the particular socket is "cut out". Furthermore, in Figure 5 the *Dispatcher* appears like a separate thread. This, however, would incur two disadvantages: (1) the dispatcher itself would need a core to run on or might preempt query evaluation threads and (2) it could become a source of contention, in particular if the morsel size was configured quite small. Therefore, the dispatcher is implemented as a lock-free data structure only. The dispatcher's code is then executed by the work-requesting query evaluation thread itself. Thus, the dispatcher is automatically executed on the (otherwise unused) core of this worker thread. Relying on lock-free data structures (i.e., the pipeline job queue as well as the associated morsel queues) reduces contention even if multiple query evaluation threads request new tasks at the same time. Analogously, the *QEPobject* that triggers the progress of a particular query by observing data dependencies (e.g., building hash tables before executing the probe pipeline) is implemented as a passive state machine. The code is invoked by the dispatcher whenever a pipeline job is fully executed as observed by not being able to find a new morsel upon a work request. Again, this state machine is executed on the otherwise unused core of the worker thread that originally requested a new task from the dispatcher.

Besides the ability to assign a core to a different query at any time – called elasticity – the morsel-wise processing also guarantees load balancing and skew resistance. All threads working on the same pipeline job run to completion in a "photo finish": they are guaranteed to reach the finish line within the time period it takes to process a single morsel. If, for some reason, a core finishes processing all morsels on its particular socket, the dispatcher will "steal work" from another core, i.e., it will assign morsels on a different socket. On some NUMA systems, not all sockets are directly connected with each other; here it pays off to steal from closer sockets first. Under normal circumstances, work-stealing from remote sockets happens very infrequently; nevertheless it is necessary to avoid idle threads. And the writing into temporary storage will be done into NUMA local storage areas anyway (that is, a red morsel turns blue if it was processed by a blue core in the process of stealing work from the core(s) on the red socket).

So far, we have discussed intra-pipeline parallelism. Our parallelization scheme can also support *bushy parallelism*, e.g., the pipelines "filtering and building the hash table of T" and "filtering and building the hash table of S" of our example are independent and could therefore be executed in parallel. However, the usefulness of this form of parallelism is limited. The number of independent pipelines is usually much smaller than the number of cores, and the amount of work in each pipeline generally differs. Furthermore, bushy parallelism can decrease performance by reducing cache locality. Therefore, we currently avoid to execute multiple pipelines from one query in parallel; in our example, we first execute pipeline T, and only after T is finished, the job for pipeline S is added to the list of pipeline jobs.

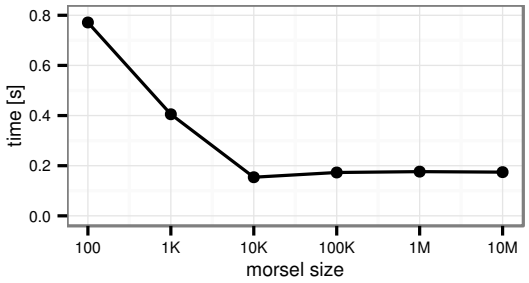


Figure 6: Effect of morsel size on query execution

Besides elasticity, morsel-driven processing also allows for a simple and elegant implementation of query canceling. A user may have aborted her query request, an exception happened in a query (e.g., a numeric overflow), or the system is running out of RAM. If any of these events happen, the involved query is marked in the dispatcher. The marker is checked whenever a morsel of that query is finished, therefore, very soon all worker threads will stop working on this query. In contrast to forcing the operating system to kill threads, this approach allows each thread to clean up (e.g., free allocated memory).

3.3 Morsel Size

In contrast to systems like Vectorwise [9] and IBM's BLU [31], which use vectors/strides to pass data between operators, there is no performance penalty if a morsel does not fit into cache. Morsels are used to break a large task into small, constant-sized work units to facilitate work-stealing and preemption. Consequently, the morsel size is not very critical for performance, it only needs to be large enough to amortize scheduling overhead while providing good response times. To show the effect of morsel size on query performance we measured the performance for the query `select min(a) from R` using 64 threads on a Nehalem EX system, which is described in Section 5. This query is very simple, so it stresses the work-stealing data structure as much as possible. Figure 6 shows that the morsel size should be set to the smallest possible value where the overhead is negligible, in this case to a value above 10,000. The optimal setting depends on the hardware, but can easily be determined experimentally.

On many-core systems, any shared data structure, even if lock-free, can eventually become a bottleneck. In the case of our work-stealing data structure, however, there are a number of aspects that prevent it from becoming a scalability problem. First, in our implementation the total work is initially split between all threads, such that each thread temporarily owns a local range. Because we cache line align each range, conflicts at the cache line level are unlikely. Only when this local range is exhausted, a thread tries to steal work from another range. Second, if more than one query is executed concurrently, the pressure on the data structure is further reduced. Finally, it is always possible to increase the morsel size. This results in fewer accesses to the work-stealing data structure. In the

worst case a too large morsel size results in underutilized threads but does not affect throughput of the system if enough concurrent queries are being executed.

4. PARALLEL OPERATOR DETAILS

In order to be able to completely parallelize each pipeline, each operator must be capable to accept tuples in parallel (e.g., by synchronizing shared data structures) and, for operators that start a new pipeline, to produce tuples in parallel. In this section we discuss the implementation of the most important parallel operators.

4.1 Hash Join

As discussed in Section 2 and shown in Figure 3, the hash table construction of our hash join consists of two phases. In the first phase, the build input tuples are materialized into a thread-local storage area[2]; this requires no synchronization. Once all input tuples have been consumed that way, an empty hash table is created with the perfect size, because the input size is now known precisely. This is much more efficient than dynamically growing hash tables, which incur a high overhead in a parallel setting. In the second phase of the parallel build phase each thread scans its storage area and inserts pointers to its tuples using the atomic compare-and-swap instruction. The details are explained in Section 4.2.

Outer join is a minor variation of the described algorithm. In each tuple a marker is additionally allocated that indicates if this tuple had a match. In the probe phase the marker is set indicating that a match occurred. Before setting the marker it is advantageous to first check that the marker is not yet set, to avoid unnecessary contention. Semi and anti joins are implemented similarly.

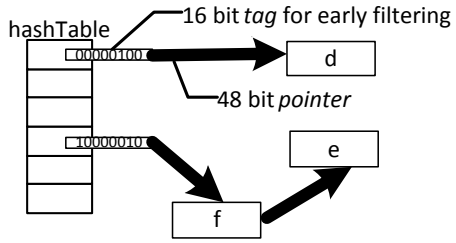
Using a number of single-operation benchmarks, Balkesen et al. showed that a highly-optimized radix join can achieve higher performance than a single-table join [5]. However, in comparison with radix join our single-table hash join

- is fully pipelined for the larger input relation, thus uses less space as the probe input can be processed *in place*,
- is a "good team player" meaning that multiple small (dimension) tables can be joined as a team by a probe pipeline of the large (fact) table through all these dimension hash tables,
- is very efficient if the two input cardinalities differ strongly, as is very often the case in practice,
- can benefit from skewed key distributions[3] [7],
- is insensitive to tuple size, and
- has no hardware-specific parameters.

Because of these practical advantages, a single-table hash join is often preferable to radix join in complex query processing. For example, in the TPC-H benchmark, 97.4% of all joined tuples arrive at the probe side, and therefore the hash table often fits into cache. This effect is even more pronounced with the Star Schema Benchmark where 99.5% of the joined tuples arrive at the probe side. Therefore, we concentrated on a single-table hash join which has the advantage of having no hardware-specific parameters and not relying on query optimizer estimates while providing very good (if the table fits into cache) or at least decent (if the table is larger than cache) performance. We left the radix join implementation, which is beneficial in some scenarios due to higher locality, for future enhancement of our query engine.

[2]We also reserve space for a next pointer within each tuple for handling hash collisions.

[3]One example that occurs in TPC-H is positional skew, i.e., in a 1:n join all join partners occur in close proximity which improves cache locality.




```
1  insert(entry) {
2    // determine slot in hash table
3    slot = entry->hash >> hashTableShift
4    do {
5      old = hashTable[slot]
6      // set next to old entry without tag
7      entry->next = removeTag(old)
8      // add old and new tag
9      new = entry | (old&tagMask) | tag(entry->hash)
10     // try to set new value, repeat on failure
11   } while (!CAS(hashTable[slot], old, new))
12 }
```

Figure 7: Lock-free insertion into tagged hash table

4.2 Lock-Free Tagged Hash Table

The hash table that we use for the hash join operator has an early-filtering optimization, which improves performance of selective joins, which are quite common. The key idea is to tag a hash bucket list with a small filter into which all elements of that particular list are "hashed" to set their 1-bit. For selective probes, i.e., probes that would not find a match by traversing the list, the filter usually reduces the number of cache misses to 1 by skipping the list traversal after checking the tag. As shown in Figure 7 (top), we encode a tag directly into 16 bits of each pointer in the hash table. This saves space and, more importantly, allows to update both the pointer and the tag using a single atomic compare-and-swap operation.

For low-cost synchronization we exploit the fact that in a join the hash table is insert-only and lookups occur only after all inserts are completed. Figure 7 (bottom) shows the pseudo code for inserting a new entry into the hash table. In line 11, the pointer to the new element (e.g, "f" in the picture) is set using compare-and-swap (CAS). This pointer is augmented by the new tag, which is computed from the old and the new tag (line 9). If the CAS failed (because another insert occurred simultaneously), the process is repeated.

Our tagging technique has a number of advantages in comparison to Bloom filters, which can be used similarly and are, for example, used in Vectorwise [8], SQL Server [21], and BLU [31]. First, a Bloom filter is an additional data structure that incurs multiple reads. And for large tables, the Bloom filter may not fit into cache (or only relatively slow last-level cache), as the Bloom filter size must be proportional to the hash table size to be effective. Therefore, the overhead can be quite high, although Bloom filters can certainly be a very good optimization due to their small size. In our approach no unnecessary memory accesses are performed, only a small number of cheap bitwise operations. Therefore, hash tagging has very low overhead and can always be used, without relying on the query optimizer to estimate selectivities. Besides join, tagging is also very beneficial during aggregation when most keys are unique.

The hash table array only stores pointers, and not the tuples themselves, i.e., we do not use open addressing. There are a number of reasons for this: Since the tuples are usually much larger than pointers, the hash table can be sized quite generously to at least twice the size of the input. This reduces the number of collisions

without wasting too much space. Furthermore, chaining allows for tuples of variable size, which is not possible with open addressing. Finally, due to our filter, probe misses are in fact faster with chaining than with open addressing.

We use large virtual memory pages (2MB) both for the hash table and the tuple storage areas. This has several positive effects: The number of TLB misses is reduced, the page table is guaranteed to fit into L1 cache, and scalability problems from too many kernel page faults during the build phase are avoided. We allocate the hash table using the Unix *mmap* system call, if available. Modern operating systems do not eagerly allocate the memory immediately, but only when a particular page is first written to. This has two positive effects. First, there is no need to manually initialize the hash table to zero in an additional phase. Second, the table is adaptively distributed over the NUMA nodes, because the pages will be located on the same NUMA node as the thread that has first written to that page. If many threads build the hash table, it will be pseudo-randomly interleaved over all nodes. In case only threads from a single NUMA node construct the hash table, it will be located on that node – which is exactly as desired. Thus, relying on the operating system automatically takes into consideration that often multiple independent queries are being executed concurrently.

4.3 NUMA-Aware Table Partitioning

In order to implement NUMA-local table scans, relations have to be distributed over the memory nodes. The most obvious way to do this is round-robin assignment. A better alternative is to partition relations using the hash value of some "important" attribute. The benefit is that in a join between two tables that are both partitioned on the join key (e.g., by the primary key of one and by the foreign key of the other relation), matching tuples usually reside on the same socket. A typical example (from TPC-H) would be to partition *orders* and *lineitem* on the *orderkey* attribute. Note that this is more a performance hint than a hard partitioning: Work stealing or data imbalance can still lead to joins between tuples from different sockets, but most join pairs will come from the same socket. The result is that there is less cross-socket communication, because the relations are co-located for this frequent join. This also affects the hash table array, because the same hash function used for determining the hash partition is also used for the highest bits of the hash buckets in a hash join. Except for the choice of the partitioning key, this scheme is completely transparent, and each partition contains approximately the same number of tuples due to the use of hash-based fragmentation. It should be stressed that this co-location scheme is beneficial but not decisive for the high performance of morsel-driven execution, as NUMA-locality is, in either case, guaranteed for table scans, and after the first pipeline that materializes results NUMA locally.

4.4 Grouping/Aggregation

The performance characteristics of the aggregation operator differs very much depending on the number of groups (distinct keys). If there are few groups, aggregation is very fast because all groups fit into cache. If, however, there are many groups, many cache misses happen. Contention from parallel accesses can be a problem in both cases (if the key distribution is skewed). To achieve good performance and scalability in all these cases, without relying on query optimizer estimates, we use an approach similar to IBM BLU's aggregation [31].

As indicated by Figure 8, our algorithm has two phases. In the first phase, thread-local pre-aggregation efficiently aggregates heavy hitters using a thread-local, fixed-sized hash table. When this small pre-aggregation table becomes full, it is flushed to overflow

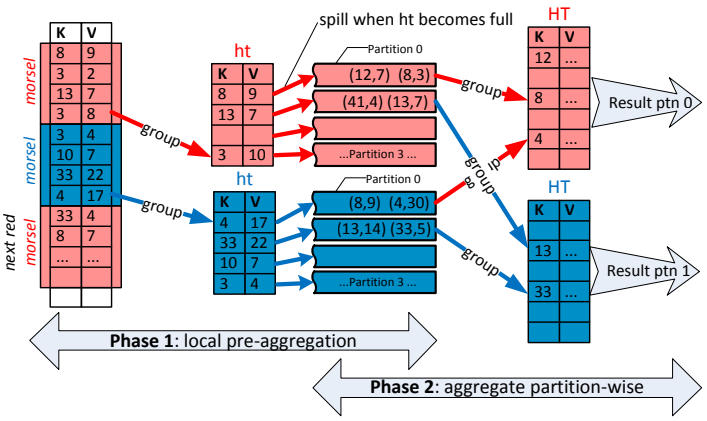


Figure 8: Parallel aggregation

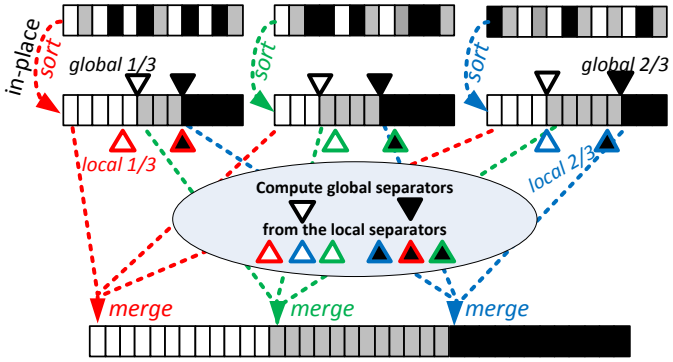


Figure 9: Parallel merge sort

partitions. After all input data has been partitioned, the partitions are exchanged between the threads.

The second phase consists of each thread scanning a partition and aggregating it into a thread-local hash table. As there are more partitions than worker threads, this process is repeated until all partitions are finished. Whenever a partition has been fully aggregated, its tuples are immediately pushed into the following operator before processing any other partitions. As a result, the aggregated tuples are likely still in cache and can be processed more efficiently.

Note that the aggregation operator is fundamentally different from join in that the results are only produced after all the input has been read. Since pipelining is not possible anyway, we use partitioning – not a single hash table as in our join operator.

4.5 Sorting

In main memory, hash-based algorithms are usually faster than sorting [4]. Therefore, we currently do not use sort-based join or aggregation, and only sort to implement the *order by* or *top-k* clause. In our parallel sort operator each thread first materializes and sorts its input locally and in place. In the case of top-*k* queries, each thread directly maintains a heap of *k* tuples.

After local sort, the parallel merge phase begins, as shown in Figure 9. The difficulty lies in computing separators, so that merges are independent and can be executed in parallel without synchronization. To do this, each thread first computes local separators by picking equidistant keys from its sorted run. Then, to handle skewed distribution and similar to the median-of-medians algorithm, the local separators of all threads are combined, sorted, and the eventual, global separator keys are computed. After determining the global

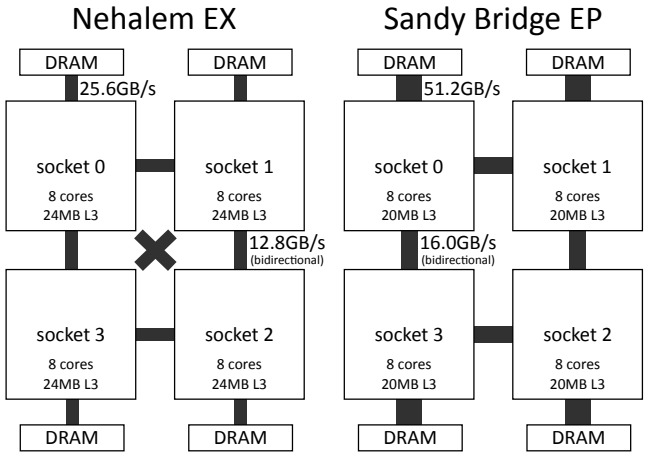


Figure 10: NUMA topologies, theoretical bandwidth

separator keys, binary (or interpolation) search finds the indexes of them in the data arrays. Using these indexes, the exact layout of the output array can be computed. Finally, the runs can be merged into the output array without any synchronization.

5. EVALUATION

We integrated our parallel query evaluation framework into HyPer, a main-memory column database system that supports SQL-92 and has very good single-threaded performance, but, so far, did not use intra-query parallelism. In this evaluation we focus on ad hoc decision support queries, and, except for declaring primary keys, do not enable any additional index structures. Therefore, our results mainly measure the performance and scalability of the table scan, aggregation, and join (including outer, semi, anti join) operators. HyPer supports both row and column-wise storage; we used the column format in all experiments.

5.1 Experimental Setup

We used two different hardware platforms – both running Linux. Unless indicated otherwise we use a 4-socket Nehalem EX (Intel Xeon X7560 at 2.3GHz). Additionally, some experiments are performed on a 4-socket Sandy Bridge EP (Intel Xeon E5-4650L at 2.6GHz-3.1GHz). Such systems are particularly suitable for main-memory database systems, as they support terabytes of RAM at reasonable cost. Although both systems have 32 cores, 64 hardware threads, and almost the same amount of cache, their NUMA topology is quite different. As Figure 10 shows, each of the Sandy Bridge CPUs has twice the theoretical per-node memory bandwidth but is only connected to two other sockets. Consequently, some memory accesses (e.g., from socket 0 to socket 2) require two hops instead of one; this increases latency and reduces memory bandwidth because of cross traffic [23]. Note that the upcoming 4-socket Ivy Bridge platform will come in two versions, Ivy Bridge EX which is fully connected like Nehalem EX, and Ivy Bridge EP with only a single interconnect per node like Sandy Bridge EP.

As our main competitor we chose the official single-server TPC-H leader Vectorwise. We also measured the performance of the open source row store PostgreSQL and a column store that is integrated into one of the major commercial database systems. On TPC-H, in comparison with HyPer, PostgreSQL was slower by a factor of 30 on average, the commercial column store by a factor of 10. We therefore concentrate on Vectorwise (version 2.5) in further experiments, as it was much faster than the other systems.

In this evaluation we used a classical ad-hoc TPC-H situation. This means that no hand-tuning of physical storage was used, as this way the plans used are similar (hash joins everywhere). The Vectorwise results from the TPC web site include this additional tuning, mainly clustered indexes, which allows to execute some of the larger joins with merge-join algorithms. Additionally, these indexes allow the query optimizer to propagate range restrictions from one join side to the other [8], which greatly improves performance for a small number of queries, but does not affect the query processing itself very much. This tuning also does not improve the scalability of query execution; on average the speedup is below 10× both with and without tuning. For completeness, we also provide results for Vectorwise on Nehalem EX with the settings from the TPC-H full disclosure report:

system	geo. mean	sum	scal.
HyPer	0.45s	15.3s	28.1×
Vectorwise	2.84s	93.4s	9.3×
Vectorwise, full-disclosure settings	1.19s	41.2s	8.4×

In HyPer the data can be updated cheaply in-place. The two TPC-H refresh streams on scale factor 100 execute in less than 1 second. This is in contrast to heavily read-optimized systems (e.g., [10]), where updates are expensive due to heavy indexing and reordering. Our system transparently distributes the input relations over all available NUMA sockets by partitioning each relation using the first attribute of the primary key into 64 partitions. The execution times include memory allocation and deallocation (from the operating system) for intermediate results, hash tables, etc.

5.2 TPC-H

Figure 11 compares the scalability of HyPer with Vectorwise on the Nehalem system; both DBMSs are normalized by the single-threaded execution time of HyPer. Note that up to 32 threads, "real" cores are used, the rest are "HyperThreads" (simultaneous multithreading). For most queries, HyPer reaches a speedup close to 30. In some cases a speedup close to or above 40 is reached due to simultaneous multithreading. Although Vectorwise has similar single-threaded performance as HyPer, its overall performance is severely limited by its low speedup, which is often less than 10. One problem is load balancing: in the – trivially parallelizable – scan-only query 6 the slowest thread often finishes work 50% before the last. While in real-world scenarios it is usually data skew that challenges load balancing, this is not the case in the fully uniform TPC-H. These issues are related to the use of the Volcano model for parallelizing queries in Vectorwise [3]. This approach, which is commonly followed (e.g., in Oracle and SQL Server), as it allows to implement parallelism without affecting existing query operators, bakes the parallelism into the plan at planning time by instantiating a set of query plans on separate plans and connecting then using "exchange" operators [12]. We point out that fixed work division combined with lack of NUMA-awareness can lead to significant performance differences between threads (Vectorwise up to version 3 is not NUMA-aware, as confirmed by our experiments in Section 5.3).

Figure 11 also shows scalability results where we disabled some important features of our query engine. Performance is significantly lower when we disable explicit NUMA-awareness and rely on the operating system instead (cf. "HyPer (not NUMA aware)"). A further performance penalty can be observed, if we additionally disable adaptive morsel-wise processing and the performance enhancements introduced in this paper like hash tagging. This gives an impression of the effects of the individual techniques. But note

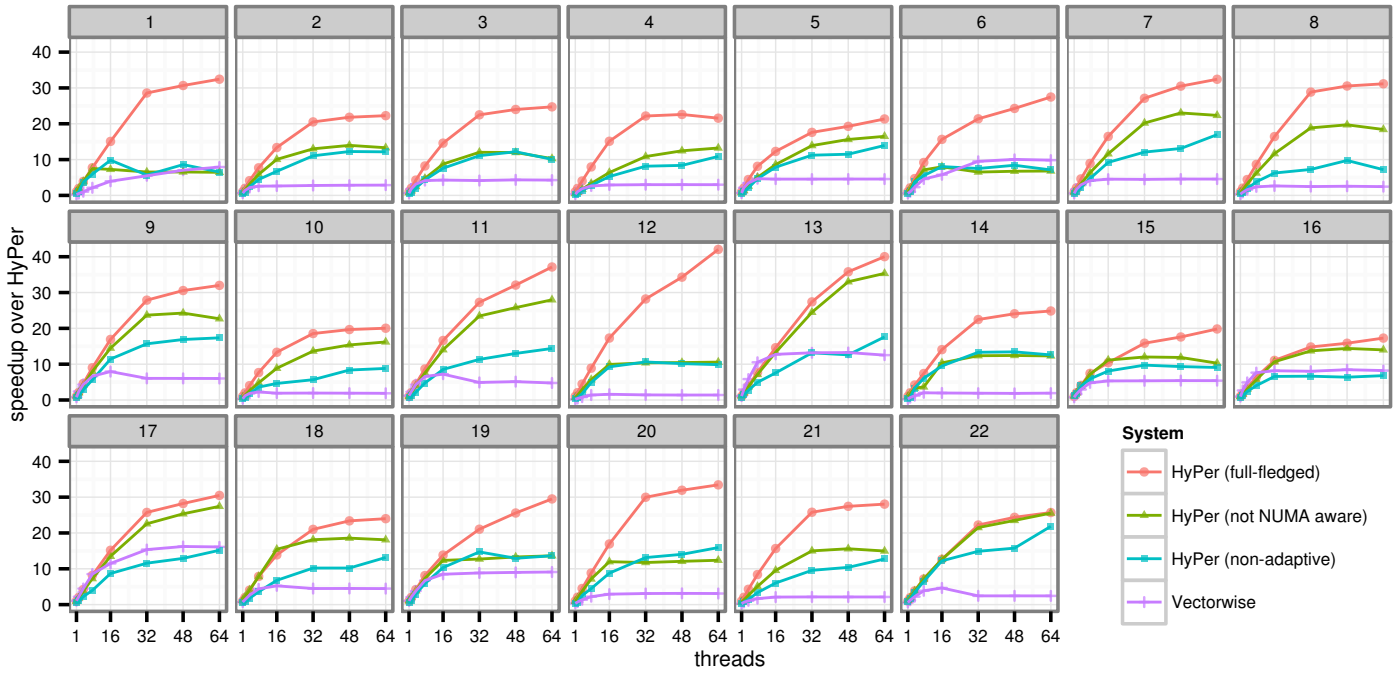


Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are "real", cores 33-64 are "virtual")

TPC-H	HyPer				[%]		Vectorwise				[%]	
#	time [s]	scal. [×]	rd. [GB/s]	wr.	remote	QPI	time [s]	scal. [×]	rd. [GB/s]	wr.	remote	QPI
1	0.28	32.4	82.6	0.2	1	40	1.13	30.2	12.5	0.5	74	7
2	0.08	22.3	25.1	0.5	15	17	0.63	4.6	8.7	3.6	55	6
3	0.66	24.7	48.1	4.4	25	34	3.83	7.3	13.5	4.6	76	9
4	0.38	21.6	45.8	2.5	15	32	2.73	9.1	17.5	6.5	68	11
5	0.97	21.3	36.8	5.0	29	30	4.52	7.0	27.8	13.1	80	24
6	0.17	27.5	80.0	0.1	4	43	0.48	17.8	21.5	0.5	75	10
7	0.53	32.4	43.2	4.2	39	38	3.75	8.1	19.5	7.9	70	14
8	0.35	31.2	34.9	2.4	15	24	4.46	7.7	10.9	6.7	39	7
9	2.14	32.0	34.3	5.5	48	32	11.42	7.9	18.4	7.7	63	10
10	0.60	20.0	26.7	5.2	37	24	6.46	5.7	12.1	5.7	55	10
11	0.09	37.1	21.8	2.5	25	16	0.67	3.9	6.0	2.1	57	3
12	0.22	42.0	64.5	1.7	5	34	6.65	6.9	12.3	4.7	61	9
13	1.95	40.0	21.8	10.3	54	25	6.23	11.4	46.6	13.3	74	37
14	0.19	24.8	43.0	6.6	29	34	2.42	7.3	13.7	4.7	60	8
15	0.44	19.8	23.5	3.5	34	21	1.63	7.2	16.8	6.0	62	10
16	0.78	17.3	14.3	2.7	62	16	1.64	8.8	24.9	8.4	53	12
17	0.44	30.5	19.1	0.5	13	13	0.84	15.0	16.2	2.9	69	7
18	2.78	24.0	24.5	12.5	40	25	14.94	6.5	26.3	8.7	66	13
19	0.88	29.5	42.5	3.9	17	27	2.87	8.8	7.4	1.4	79	5
20	0.18	33.4	45.1	0.9	5	23	1.94	9.2	12.6	1.2	74	6
21	0.91	28.0	40.7	4.1	16	29	12.00	9.1	18.2	6.1	67	9
22	0.30	25.7	35.5	1.3	75	38	3.14	4.3	7.0	2.4	66	4

Table 1: TPC-H (scale factor 100) statistics on Nehalem EX

that we still use highly tuned operator implementations that try to maximize locality.

Table 1 and Table 2 allow to compare the TPC-H performance of the Nehalem and Sandy Bridge systems. The overall performance is similar on both systems, because the missing interconnect links on Sandy Bridge EP, which result in slightly lower scalability, are compensated by its higher clock rate. Notice that all queries complete within 3 seconds – on a 100GB data set using ad hoc hash joins and without using any index structures.

5.3 NUMA Awareness

Table 1 shows memory bandwidth and QPI statistics[4] for each of the 22 TPC-H queries. Query 1, which aggregates the largest relation, for example, reads 82.6GB/s getting close to the theoretical bandwidth maximum of 100GB/s. The "remote" column in the table shows the percentage of data being accessed though the interconnects (remotely), and therefore measures the locality of each query. Because of NUMA-aware processing, most data is accessed locally, which results in lower latency and higher bandwidth. From the "QPI" column[5], which shows the saturation of the most heavily used QPI link, one can conclude that the bandwidth of the QPI links is sufficient on this system. The table also shows that Vectorwise is not NUMA optimized: most queries have high percentages of remotely accessed memory. For instance, the 75% remote accesses in query 1 shows that its buffer manager is not NUMA-aware. However, the QPI links are utilized fairly evenly, as the database relations seem to be spread over all 4 NUMA nodes. This prevents a single memory controller and the QPI links to it from becoming the bottleneck.

Most experiments so far used our NUMA-aware storage layout, NUMA-local scans, the NUMA-aware partitioning, which reduces remote accesses in joins, and the fact that all operators try to keep data NUMA-local whenever possible. To show the overall

[4]These statistics were obtained using the Open Source tool "Intel Performance Counter Monitor" (`www.intel.com/software/pcm`). The "rd." (read), "wr." (write), and "remote" values are aggregated over all sockets. The "QPI" column shows the utilization of the most-utilized QPI link (though with NUMA-awareness the utilization of the links is very similar). Unfortunately, these statistics are not exposed on Sandy Bridge EP.

[5]The QPI links are used both for sending the actual data, as well as for broadcasting cache coherency requests, which is unavoidable and happens even for local accesses. Query 1, for example, reads 82.6GB/s, 99% of it locally, but still uses 40% of the QPI link bandwidth.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
time [s]	0.21	0.10	0.63	0.30	0.84	0.14	0.56	0.29	2.44	0.61	0.10	0.33	2.32	0.33	0.33	0.81	0.40	1.66	0.68	0.18	0.74	0.47
scal. [×]	39.4	17.8	18.6	26.9	28.0	42.8	25.3	33.3	21.5	21.0	27.4	41.8	16.5	15.6	20.5	11.0	34.0	29.1	29.6	33.7	26.4	8.4

Table 2: TPC-H (scale factor 100) performance on Sandy Bridge EP

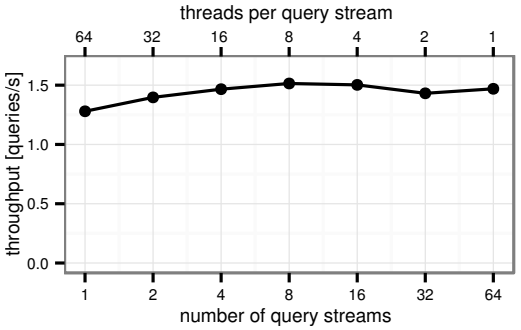


Figure 12: Intra- vs. inter-query parallelism with 64 threads

performance benefit of NUMA-awareness we also experimented with plausible alternatives: "OS default", where the placement is performed by the operating system[6], and "interleaved", where all memory is allocated round robin over all nodes. We report the geometric mean and maximum speedup of our NUMA-aware approach on TPC-H:

	Nehalem EX		Sandy Bridge EP	
	geo. mean	max	geo. mean	max
OS default	1.57×	4.95×	2.40×	5.81×
interleaved	1.07×	1.24×	1.58×	5.01×

Clearly, the default placement of the operating system is sub-optimal, as the memory controller of one NUMA node and the QPI links to it become the bottleneck. These results also show that on Nehalem EX, simply interleaving the memory is a reasonable, though not optimal strategy, whereas on Sandy Bridge EP NUMA-awareness is much more important for good performance. The reason is that these two systems are quite different in their NUMA behavior, as can be seen from a micro benchmark that compares NUMA-local accesses with a random mix of 25% local and 75% remote (including 25% two-hop accesses on Sandy Bridge EP) accesses:

	bandwidth [GB/s]		latency [ns]	
	local	mix	local	mix
Nehalem EX	93	60	161	186
Sandy Bridge EP	121	41	101	257

On Sandy Bridge EP only a small fraction of the theoretical memory bandwidth can be reached unless most accesses are local, and the latency it 2.5× higher than for local accesses. On Nehalem EX, in contrast, these effects are much smaller, which explains why the positive effect of NUMA-awareness is smaller on this system. The importance of NUMA-awareness clearly depends on the speed and number of the cross-socket interconnects.

5.4 Elasticity

To demonstrate the elasticity of our approach, we performed an experiment where we varied the number parallel query streams. The 64 available hardware threads are distributed uniformly over the streams, and each stream executes random permutations of the TPC-H queries. Figure 12 shows that the throughput stays high even if few streams (but many cores per stream) are used. This allows to minimize response time for high priority queries without sacrificing too much throughput.

[6]In practice, the database itself is located on a single NUMA node, because the data is read from disk by a single thread. Other allocations are local to the thread that first wrote to that memory. Thus, hash tables are distributed randomly over the nodes.

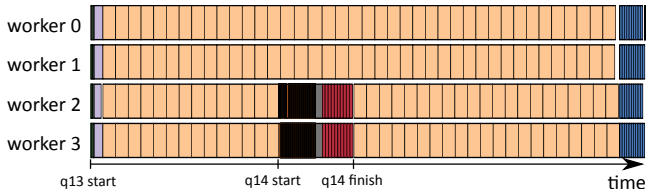


Figure 13: Illustration of morsel-wise processing and elasticity

SSB #	time [s]	scal. [×]	read [GB/s]	write [GB/s]	remote [%]	QPI [%]
1.1	0.10	33.0	35.8	0.4	18	29
1.2	0.04	41.7	85.6	0.1	1	44
1.3	0.04	42.6	85.6	0.1	1	44
2.1	0.11	44.2	25.6	0.7	13	17
2.2	0.15	45.1	37.2	0.1	2	19
2.3	0.06	36.3	43.8	0.1	3	25
3.1	0.29	30.7	24.8	1.0	37	21
3.2	0.09	38.3	37.3	0.4	7	22
3.3	0.06	40.7	51.0	0.1	2	27
3.4	0.06	40.5	51.9	0.1	2	28
4.1	0.26	36.5	43.4	0.3	34	34
4.2	0.23	35.1	43.3	0.3	28	33
4.3	0.12	44.2	39.1	0.3	5	22

Table 3: Star Schema Benchmark (scale 50) on Nehalem EX

Figure 13 illustrates morsel-wise processing by showing an annotated execution trace from our parallel profiler. Each color represents one pipeline stage and each block is one morsel. For graphical reasons we used only 4 threads in this experiment. We started by executing TPC-H query 13, which received 4 threads; after some time, TPC-H query 14 was started. As the trace shows, once the current morsels of worker thread 2 and 3 are finished, these threads switch to query 14 until it is finished, and finally continue working on query 13. This experiment shows that it is possible to dynamically reassign worker threads to other queries, i.e., that our parallelization scheme is *fully elastic*.

As mentioned in the introduction, the Volcano approach typically assigns work to threads statically. To compare with this approach, we emulated it in our morsel-driven scheme by splitting the work into as many chunks as there are threads, i.e., we set the morsel size to n/t, where n is the input size and t is the number of threads. As long as we only execute a single TPC-H query at a time, this change alone does not significantly decrease performance, because the input data is uniformly distributed on this workload. However, if we add some interference from other processes, this picture changes. For example, when we ran the TPC-H queries while another, unrelated single-threaded process occupied one core, query performance dropped by 36.8% with static approach, but only 4.7% with dynamic morsel assignment.

5.5 Star Schema Benchmark

Besides TPC-H, we also measured the performance and scalability of our system on the Star Schema Benchmark (SSB) [26], which mimics data warehousing scenarios. Table 3 shows that our parallelization framework works very well on this workload, achieving

a speedup of over 40 for most queries. The scalability is higher than on TPC-H, because TPC-H is a much more complex and challenging workload. TPC-H contains a very diverse set of queries: queries that only scan a single table, queries with complex joins, queries with simple and with complex aggregations, etc. It is quite challenging to obtain good performance *and* scalability on such a workload, as all operators must be scalable and capable of efficiently handling very diverse input distributions. All SSB queries, in contrast, join a large fact table with multiple smaller dimension tables where the pipelining capabilities of our hash join algorithm are very beneficial. Most of the data comes from the large fact table, which can be read NUMA-locally (cf. column "remote" in Figure 3), the hash tables of the dimensions are much smaller than the fact table, and the aggregation is quite cheap in comparison with the rest of the query.

6. RELATED WORK

This paper is related to three distinct lines of work: papers that focus on multi-core join or aggregation processing in isolation, full systems descriptions, and parallel execution frameworks, most notably Volcano.

The radix hash join was originally designed to increase locality [24]. Kim et al. postulated it for parallel processing based on repeatedly partitioning the input relations [18]. Blanas et al. [7] were the first to compare the radix join with a simple, single global hash table join. Balkesen et al. [5, 4] comprehensively investigated hash- and sort-based join algorithms. Ye et al. evaluated parallel aggregation algorithms on multi-core CPUs [33]. Polychroniou and Ross designed an aggregation algorithm to efficiently aggregate heavy hitters (frequent items) [27].

A number of papers specifically focus on NUMA. In one of the first paper that pinpoints the relevance of NUMA-locality, Teubner and Müller [32] presented a NUMA-aware window-based stream join. In another early NUMA paper, Albutiu et al. designed a NUMA-aware parallel sort merge join [1]. Li et al. refined this algorithm by explicitly scheduling the shuffling of the sorted runs in order to avoid cross traffic in the NUMA interconnection network [23]. However, despite its locality-preserving nature this algorithm turned out to be less efficient than hash joins due to the high cost of sorting [4, 20]. Lang et al. [20] devised a low synchronization overhead NUMA-aware hash join, which is similar to our algorithm. It relies on a single latch-free hash table interleaved across all NUMA nodes into which all threads insert the build input.

Unfortunately, the conclusiveness of these single-operator studies for full-fledged query engines is limited because the micro-benchmarks used for testing usually have single simple keys (sometimes even containing hash values), and typically use very small payloads (one column only). Furthermore, each operator was analyzed in isolation, which ignores how data is passed between operators and therefore, for example, ignores the different pipelining capabilities of the algorithms. In our morsel-driven database system, we have concentrated on (non-materializing) pipelined hash joins, since in practice, often one of the join sides is much larger than the others. Therefore, teams of pipelined joins are often possible and effective. Further, for certain often-traversed large joins (such as orders-lineitem in TPC-H), pre-partitioned data storage can achieve NUMA locality on large joins without need for materialization.

The new IBM BLU query engine [31] and Microsoft's Apollo project [22] are two prominent commercial projects to exploit modern multi-core servers for parallel query processing. IBM BLU processes data in "Vectorwise" fashion, a so-called stride at a time. In this respect there is some resemblance to our morsel-wise processing technique. However, there was no indication that the strides are maintained NUMA-locally across processing steps/pipelines. In addition, the full elasticity w.r.t. the degree of parallelism that we propose was not covered. Very similar to Volcano-style parallelization, in Oracle the individual operators are largely unaware of parallelism. [6] addresses some problems of this model, in particular reliance on query optimizer estimates, by adaptively changing data distribution decisions during query execution. In an experimental study Kiefer et al. [17] showed that NUMA-awareness can improve database performance considerably. Porobic et al. investigated [29] and improved NUMA-placement in OLTP systems by partitioning the data and internal data structures in a NUMA-aware way [28]. Heimel et al. presented a hardware-oblivious approach to parallelization that allows operators to be compiled to different hardware platforms like CPUs or GPUs [15]. In this paper we focus on classical, query-centric parallelization, i.e., parallelizing individual queries in isolation. Another fruitful approach is to exploit common work from multiple queries. This operator-centric approach is used by QPipe [14] and SharedDB [11].

The seminal Volcano model [12] forms the basis of most current query evaluation engines enabling multi-core as well as distributed [13] parallelism. Note that Volcano in a non-parallel context is also associated with an interpreted iterator execution paradigm where results are pulled upwards through an operator tree, by calling the *next()* method on each operator, which delivers the next tuple. Such a tuple-at-a-time execution model, while elegant in its implementation, has been shown to introduce significant interpretation overhead [25]. With the advent of high-performance analytical query engines, systems have been moving from this model towards vector or batch-oriented execution, where each *next()* method works on hundreds or thousands of tuples. This vector-wise execution model appears in Vectorwise [3], but also in the batch-mode execution offered by ColumnStore Index tables in SQL Server [22] (the Apollo project), as well as in stride-at-a-time execution in IBM's BLU engine for DB2 [31]. In HyPer we rely on a compiled query evaluation approach as first postulated by Krikellas et al. [19] and later refined by Neumann [25] to obtain the same, or even higher raw execution performance.

As far as parallelism is concerned, Volcano differentiates between vertical parallelism, where essentially the pipeline between two operators is transformed into an asynchronous producer/consumer model, and horizontal parallelism, where one operator is parallelized by partitioning the input data and have each parallel thread work on one of the partitions. Most systems have implemented horizontal parallelism, since vertical and bushy parallelism are less useful due to their unbalanced nature, as we observed earlier. Examples of such horizontal Volcano parallelism are found in e.g., Microsoft SQL Server and Vectorwise [3].

While there may be (undisclosed) implementation differences between these systems, morsel-driven execution differentiates itself by making parallel query scheduling fine-grained, adaptive at run-time and NUMA-aware. The parallel query engine described here relies on chunking of the input data into fine-grained morsels. A morsel resides completely in a single NUMA partition. The dispatcher assigns the processing of a morsel to a thread running on a core of the same socket in order to preserve NUMA locality. The morsel-wise processing also facilitates the full elasticity, meaning that the degree of parallelism can be adjusted at any time, e.g., at mid-query processing. As soon as a morsel is finished, the thread can be assigned a morsel belonging to the same query pipeline or be assigned a different task of, e.g., another more important query. This way the dispatcher controls parallelism explicitly as opposed to the recently proposed approach by Psaroudakis et al. [30] where the number of threads is changed based on the core utilization.

7. CONCLUSIONS AND FUTURE WORK

We presented the morsel-driven query evaluation framework for parallel query processing. It is targeted at solving the major bottlenecks for analytical query performance in the many-core age, which are load-balancing, thread synchronization, memory access locality, and resource elasticity. We demonstrated the good scalability of this framework in HyPer on the full TPC-H and SSB query workloads. It is important to highlight, that at the time of this writing, the presented results are by far the fastest achieved (barring the hand-written queries on a fully indexed and customized storage scheme [10][7]) on a single-server architecture. This is not being noted to claim a performance record – these are academic and non-audited results – but rather to underline the effectiveness of the morsel-driven framework in achieving scalability. In particular, one needs to keep in mind that it is much easier to provide linear scalability on computationally slow systems than it is on fast systems such as HyPer. The comparison with the state-of-the-art Vectorwise system, which uses a classical implementation of Volcano-style parallelism [3], shows that beyond 8 cores, in many-core territory, the morsel-driven framework speeds ahead; and we believe that its principles in fine-grained scheduling, full operator parallelization, low-overhead synchronization and NUMA-aware scheduling can be used to improve the many-core scaling in other systems as well.

Besides scalability, the fully elastic morsel-driven parallelism allows for intelligent priority-based scheduling of dynamic query workloads. The design and evaluation of such a scheduler, which takes quality-of-service constraints into account, was beyond the scope of this paper and will be addressed in forthcoming work.

Our system performs well on a number of very different hardware platforms despite having no hardware-specific parameters (we tested with single-socket systems and NUMA systems with different topologies). Nevertheless, it would be interesting to investigate algorithms that take knowledge of the underlying hardware into account. There is certainly room for further optimizations, specifically those that further reduce remote NUMA access, as shown by the slower results on the Sandy Bridge EP platform with its partially connected NUMA topology when compared with the fully-connected Nehalem EX.

8. REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10), 2012.

[2] G. Alonso. Hardware killed the software star. In *ICDE*, 2013.

[3] K. Anikiej. Multi-core parallelization of vectorized query execution. Master's thesis, University of Warsaw and VU University Amsterdam, 2010. `http://homepages.cwi.nl/~boncz/msc/2010-KamilAnikijej.pdf`.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1), 2013.

[5] 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.

[6] S. Bellamkonda, H.-G. Li, U. Jagtap, Y. Zhu, V. Liang, and T. Cruanes. Adaptive and big data scale parallel execution in oracle. *PVLDB*, 6(11), 2013.

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

[7] The paper by Dees and Sanders [10], while interesting as an extreme take on TPC-H, visibly violates many of its implementation rules, including the use of precomputed joins, precomputed aggregates, and full-text indexing. It generally presents a storage structure that is very expensive to update.

[8] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, 2013.

[9] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[10] J. Dees and P. Sanders. Efficient many-core query execution in main memory column-stores. In *ICDE*, 2013.

[11] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6), 2012.

[12] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, 1990.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), 1993.

[14] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[15] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9), 2013.

[16] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[17] T. Kiefer, B. Schlegel, and W. Lehner. Experimental evaluation of NUMA effects on database management systems. In *BTW*, 2013.

[18] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2), 2009.

[19] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[20] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel NUMA-aware hash joins. In *IMDM Workshop*, 2013.

[21] P.-Å. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL Server column stores. In *SIGMOD*, 2013.

[22] P.-Å. Larson, E. N. Hanson, and S. L. Price. Columnar storage in SQL Server 2012. *IEEE Data Eng. Bull.*, 35(1), 2012.

[23] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[24] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4), 2002.

[25] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4, 2011.

[26] P. O'Neil, B. O'Neil, and X. Chen. The star schema benchmark (SSB), 2007. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[27] O. Polychroniou and K. A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, 2013.

[28] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive transaction processing on hardware islands. In *ICDE*, 2014.

[29] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on hardware islands. *PVLDB*, 5(11), 2012.

[30] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *ADMS Workshop*, 2013.

[31] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. In *VLDB*, 2013.

[32] J. Teubner and R. Müller. How soccer players would do stream joins. In *SIGMOD*, 2011.

[33] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, 2011.

[34] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1), 2012.