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

April 21, 2025

1 Introduction

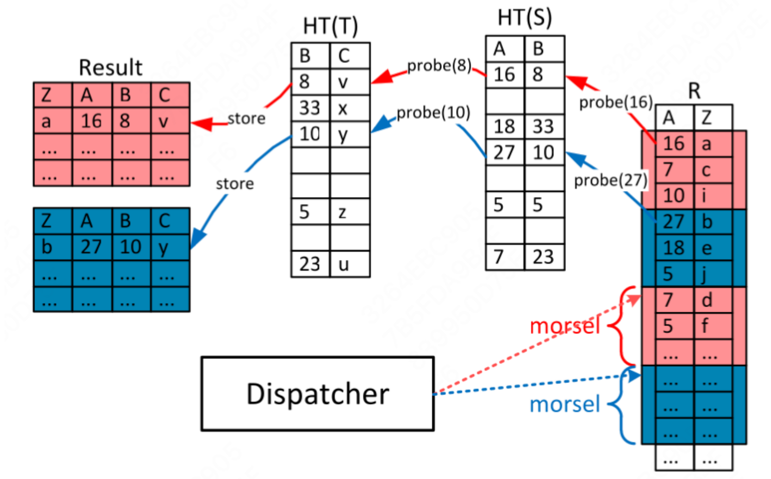


Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

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

Parallism 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.

2 Morsel-Driven Execution

Consider

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

Assuming that R is the largest table (after filtering) the optimizer would choose R as probe input and build hash tables of the other two S and T.

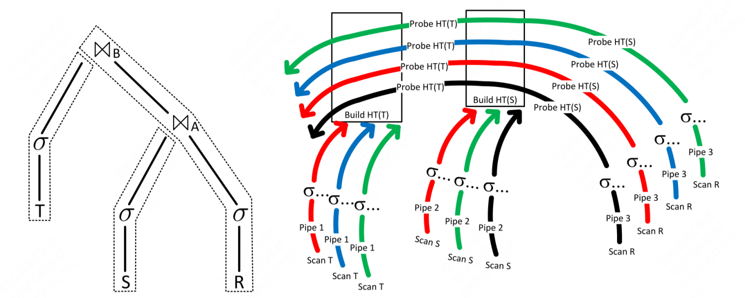


Figure 2: Parallellizing the three pipelines of the sample query plan: (left) algebraic evaluation plan; (right) three- respectively four-way parallel processing of each pipeline

The plan consists of the three pipelines:

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.

The morsel-driven execution of the algebraic plan is controlled by a so called `QEPobject` which transfers executable pipelines to a dispatcher. 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 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.

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.

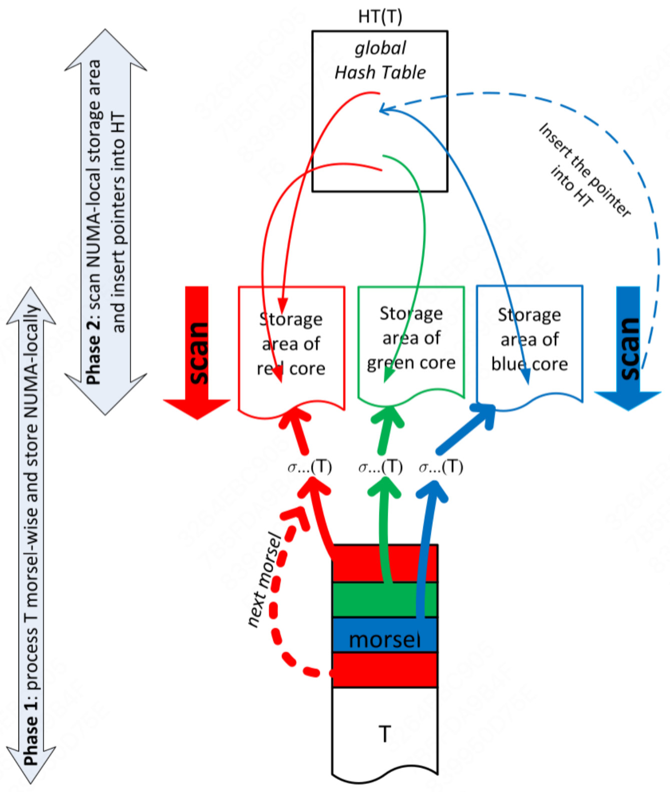

HT(T)
global Hash Table
Insert the pointer into HT
Phase 2: scan NUMA-local storage area and insert pointers into HT
Phase 1: process T morsel-wise and store NUMA-locally
scan
Storage area of red core
Storage area of green core
Storage area of blue core
scan
σ...(T)
σ...(T)
σ...(T)
next morsel
morsel
T

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

The **parallel** processing of the pipeline for filtering T and building the hash table $HT(T)$ is shown in Figure 2. 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

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. What if the data is skewed?

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.

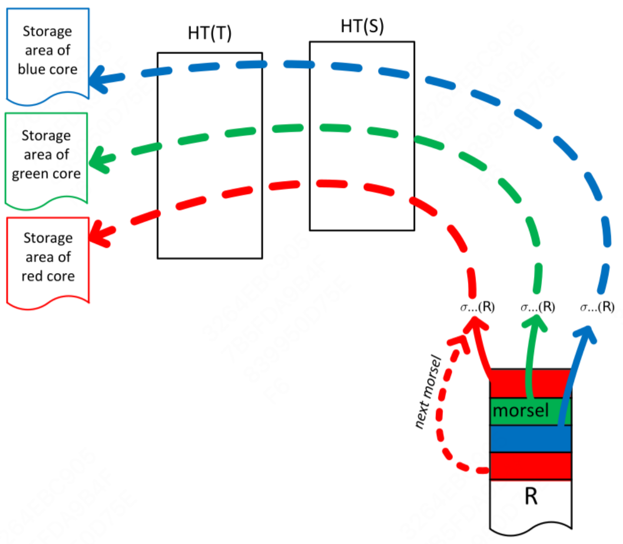


Figure 4: Morsel-wise processing of the probe phase

After both hash tables have been constructed, the probing pipeline can be scheduled as in Figure 2. Again, a thread requests work from the dispatcher which assigns a morsel in the corresponding NUMA partition.

3 Dispatcher: Scheduling Parallel Pipeline Tasks

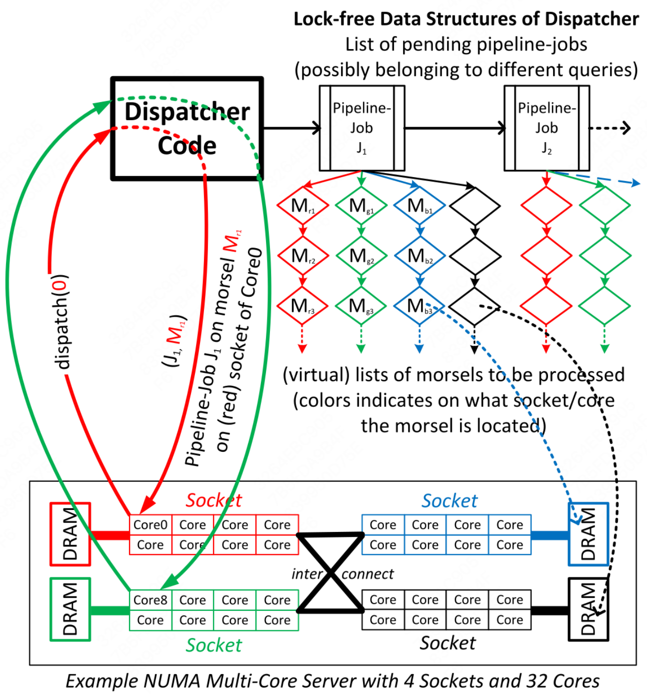

Lock-free Data Structures of Dispatcher
List of pending pipeline-jobs
(possibly belonging to different queries)
Dispatcher Code
Pipeline-Job J1
Pipeline-Job J2
dispatch(0)
(J1, Mr1)
Pipeline-Job J1 on morsel Mr1 on (red) socket of Core0
(virtual) lists of morsels to be processed
(colors indicates on what socket/core the morsel is located)
Socket
Core0
Core8
DRAM
inter connect
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

The **dispatcher** is controlling and assigning the compute resources to the parallel pipelines. We (pre-)create one worker thread for each hardware thread that the machine provides and permanently bind each worker to it. Preemption of a task occurs at morsel boundaries.

3.1 Elasticity

3.2 Implementation Overview

In Figure 3 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
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.

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.

Therefore, we currently avoid to execute multiple pipelines from one query in parallel; in our example,

3.3 Morsel Size

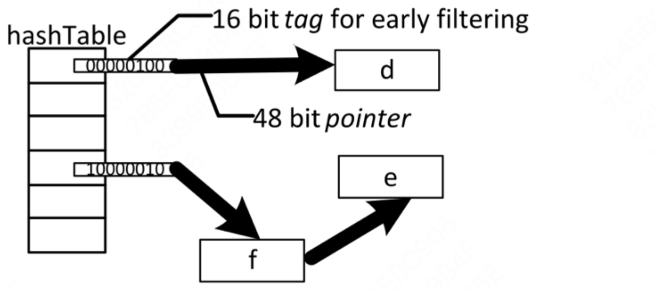
it only needs to be large enough to amortize scheduling overhead while providing good response times.

4 Parallel Operator Details

4.1 Hash Join

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.



```
insert(entry) {
  // determine slot in hash table
  slot = entry->hash >> hashTableShift
  do {
    old = hashTable[slot]
    // set next to old entry without tag
    entry->next = removeTag(old)
    // add old and new tag
    new = entry | (old&tagMask) | tag(entry->hash)
    // try to set new value, repeat on failure
  } while (!CAS(hashTable[slot], old, new))
}
```

Figure 7: Lock-free insertion into tagged hash table

4.3 NUMA-Aware Table Partitioning

Goal: NUMA-local tables scan

Prerequisite: relations have to be distributed over the memory nodes.

How: partition relations using the hash value of some "important" attribute.

4.4 Grouping/Aggregation

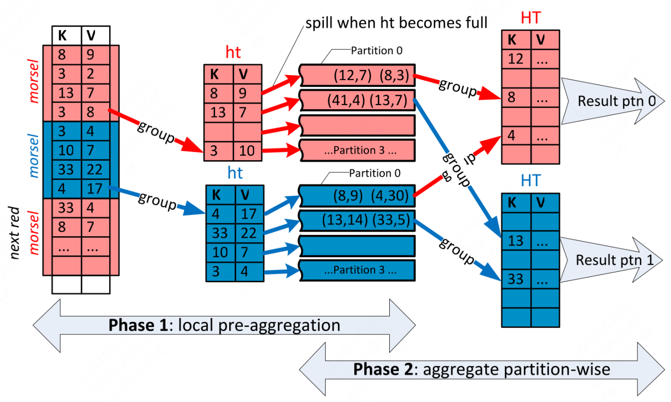


Figure 8: Parallel aggregation

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 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.

4.5 Sorting

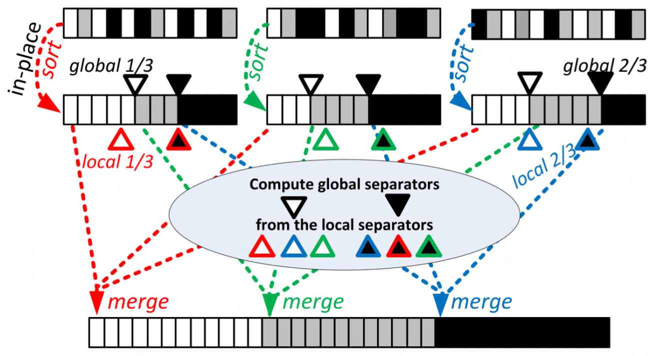


Figure 9: Parallel merge sort

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 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 Problems

6 References

References