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

Shun Yao, JunChen Liu, YiNan An, Minghui Yang

Volcano model

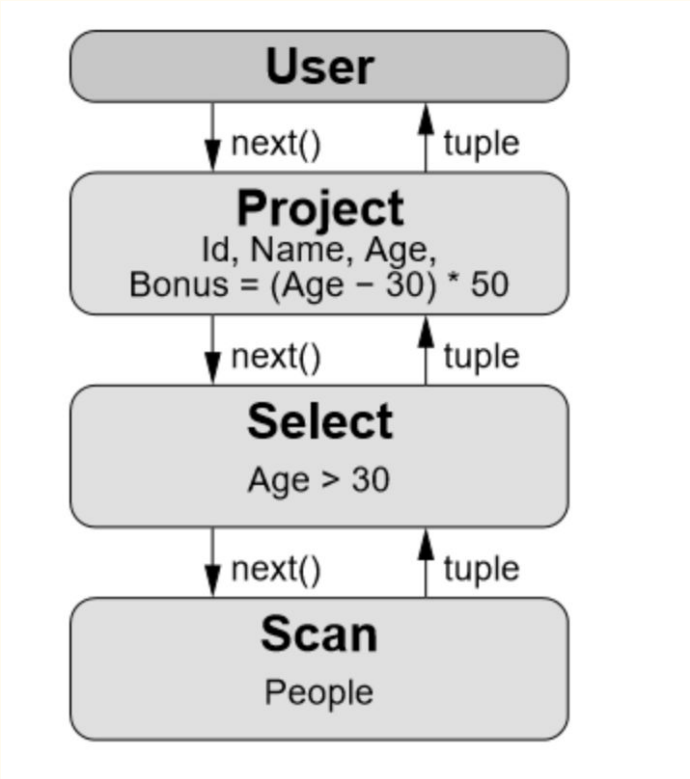
- Abstracts every operation in relational algebra as an Operator, and constructs the entire SQL as an Operator ***tree***, recursively calling the next() function from the root node to the leaf node from top to bottom.

- Example
  - **SELECT Id, Name, Age, (Age - 30) * 50 AS Bonus**

    FROM People

    WHERE Age > 30

Volcano model(cont'd)



Volcano model(cont'd)

- Advantage
  - It is simple. Each Operator can be abstracted individually without concern for the logic of other operators.
- Drawback
  - The next() function is called every time a Tuple is evaluated, resulting in a large number of virtual function calls, which can result in low CPU usage.

Deficiencies of past optimizers

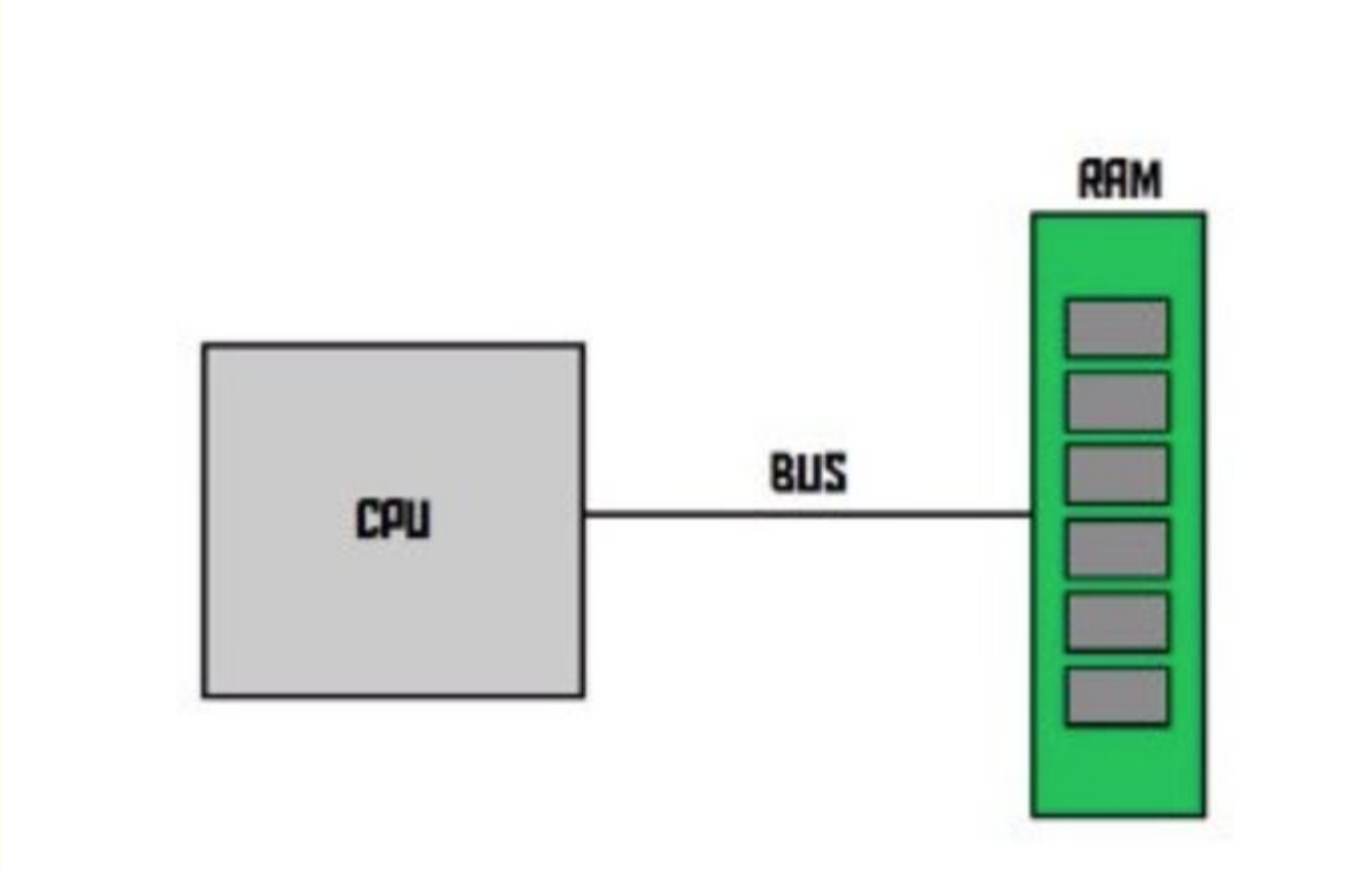
- Optimizer based on volcano model
- IO much slower than CPU
  - Hence, initially focus on reducing IO (rather than optimizing CPU utilization)
  - With larger memory capacity and multi-core processors, I/O is no longer a limitation
- Now people start to care about the rate of CPU usage

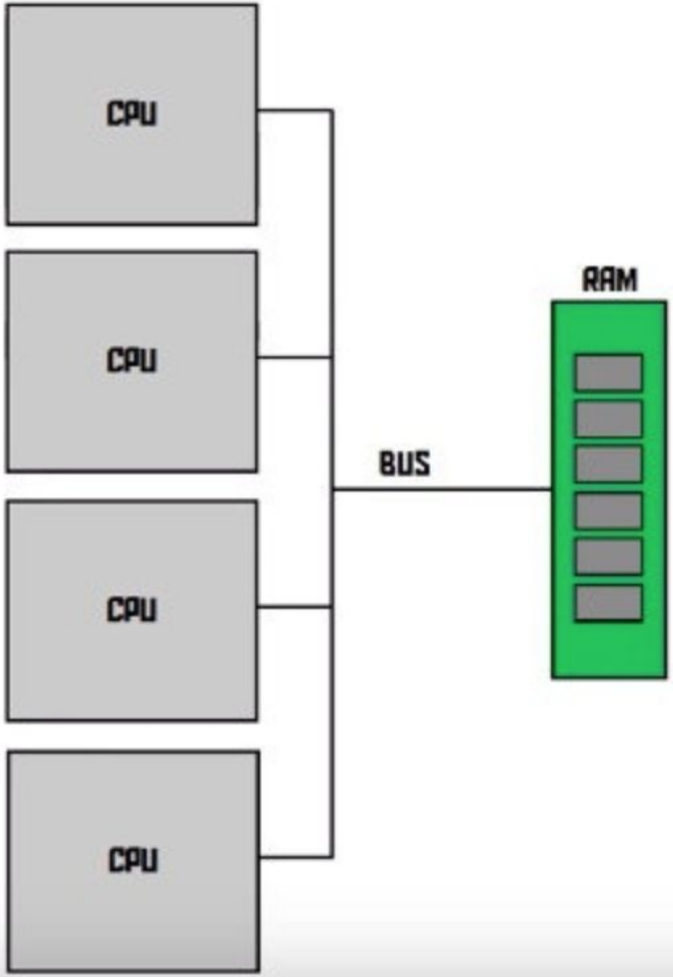
NUMA

- Non-uniform memory access (NUMA) is a computer memory design for multiple processors. The memory access time depends on the memory location of the processor. Under NUMA, a processor can access its own local memory faster than non-local memory

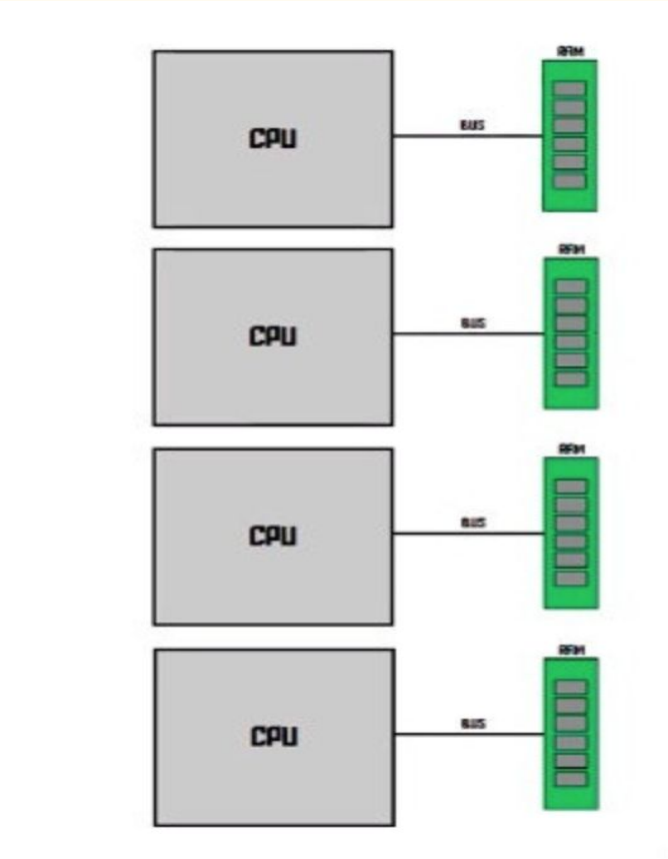
One cpu connects to

a RAM by a BUS





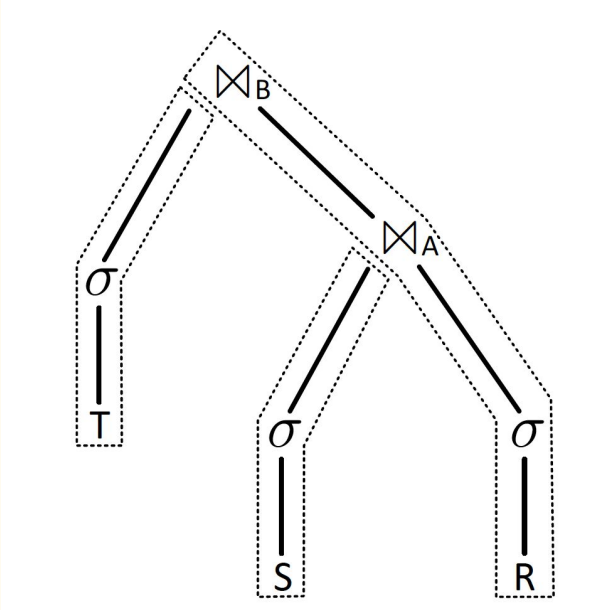
A number of cpu are connected to a RAM by a BUS



By treating the CPU and neighboring RAM as one node, the CPU preferentially accesses the nearest RAM. At the same time, the CPU has a fast channel connection directly, so each CPU still has access to all RAM locations (only at different speeds).

Processing of parallel threads(macro)

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.

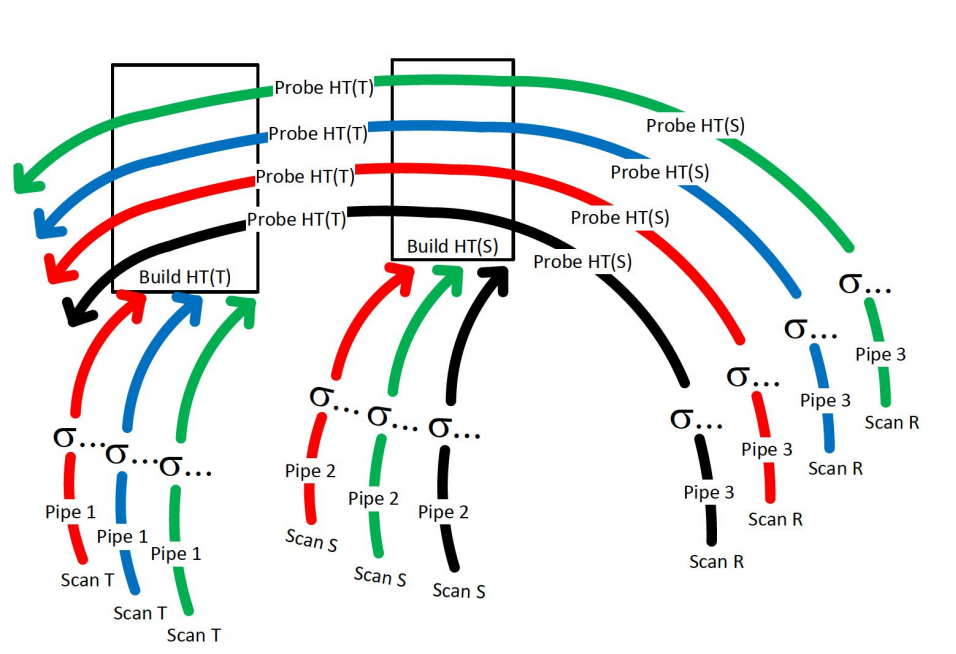


Processing of parallel threads(micro)

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.



Date storage during the processing

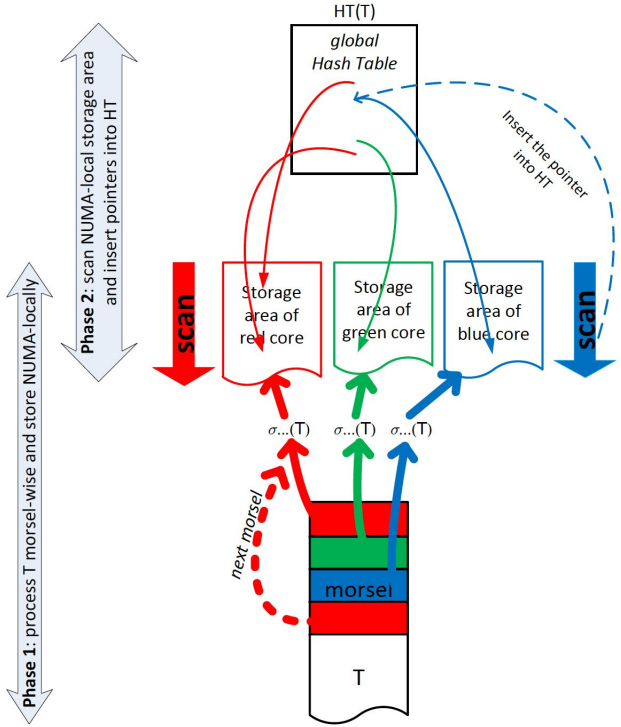


Figure 3: NUMA-aware processing of the build-phase

The following diagram shows how to speed up building hashTable (T)

We can see that there are three threads to do one thing at the same time, and they have their own local storage

When the first red thread completes the task, it will work on a new Morsel

3. Dispatcher: Scheduling Parallel Pipeline Tasks

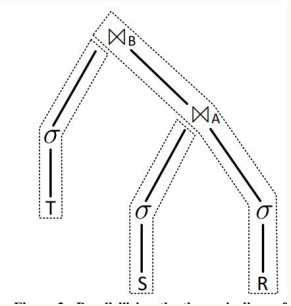
- Dispatcher - Assign tasks to work threads
- Task - A morsel + A pipeline job
- Tradeoff - Instant Elasticity Adjustment, LB & Low Maintenance Overhead
- Goals - Locality, Elasticity & Load Balancing

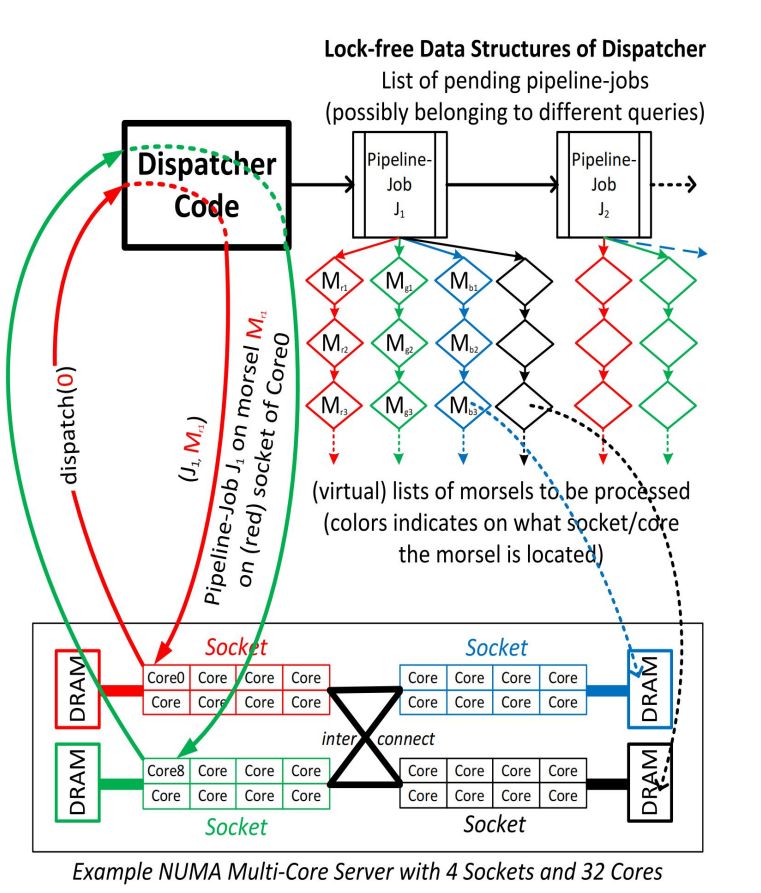
Goals for assigning tasks to threads

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

Dispatcher Architecture

- Maintain a list of pending pipeline jobs.
- Maintains lists of pending morsels
- A core requests a task from the dispatcher
- Dispatcher dispatches a task to core





Elasticity

- Elasticity: The ability to assign a core to a different query at anytime
- Why it is good? Example: Q-l(long query) vs Q+(important query)

  Decrease the degree of parallelism of Q-l at any stage of processing in order to prioritize a more important query Q+.

  Q+ is finished —> back to Q-l (by dispatching most cores to Q-l)

- Require: Dispatching jobs `a morsel at a time` + A priority-based scheduling component

Implementation of Dispatcher

- The next morsel is cut down from storage area
- Dispatcher is implemented as a lock-free data structure
- QEPobject - passive state machine

Pros of morsel-wise processing

- No performance penalty if a morsel does not fit into cache. (In contrast to systems like Vectorwise and IBM's BLU, which use vectors/strides to pass data between operators,)
- Morsel size is not very critical for performance

Morsel Size

- Morsel size is not very critical for performance
- It needs to be large enough to amortize scheduling overhead

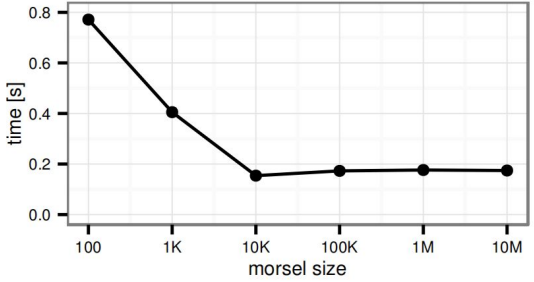


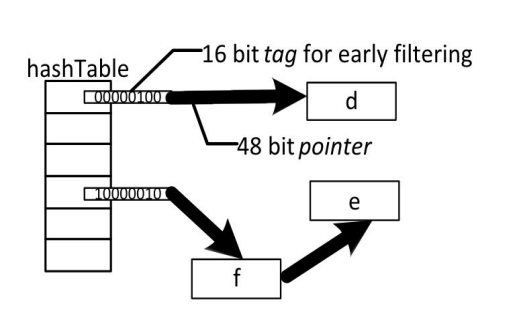
Figure 6: Effect of morsel size on query execution

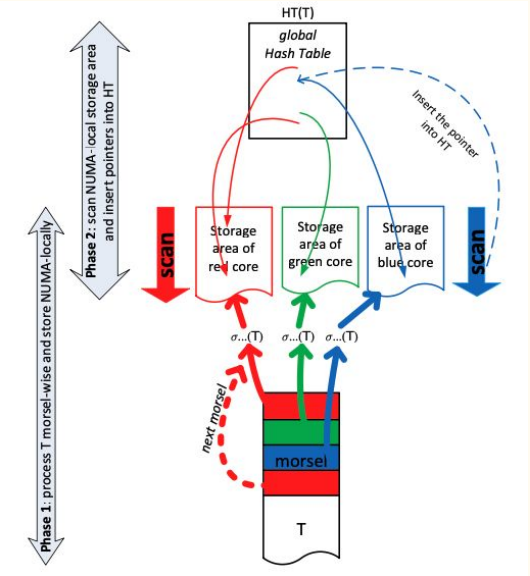
Parallel Operator

- Hash Join
- Lock-Free Tagged Hash Table
- Table Partitioning
- Aggregation
- Sorting

Hash Join

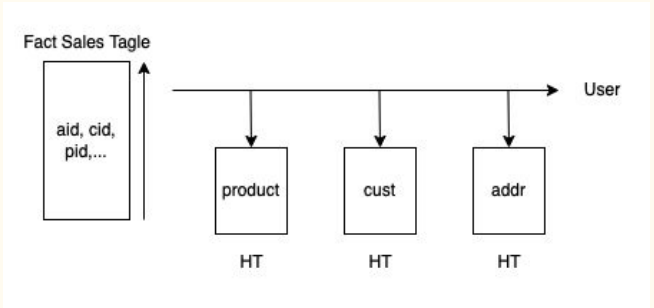
- Hash table construction
  - First phase(no synchronization): input tuples store in the local storage area(reserve space for a pointer within each tuple), and create an empty hash table. The size of table is known precisely.
  - Second phase: each thread scan its local storage and insert pointers



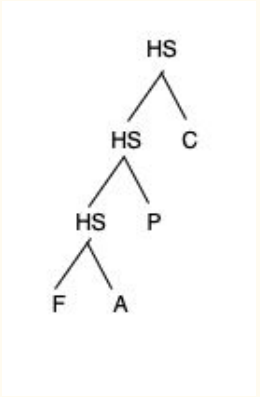


Hash Join

- Advantages:
  - less space (fully pipelined, in-place)
  - "good team player"
  - efficient (cardinalities differ strongly)
  - benefit from skew key distribution
  - insensitive to tuple size
  - no hardware-specific parameters



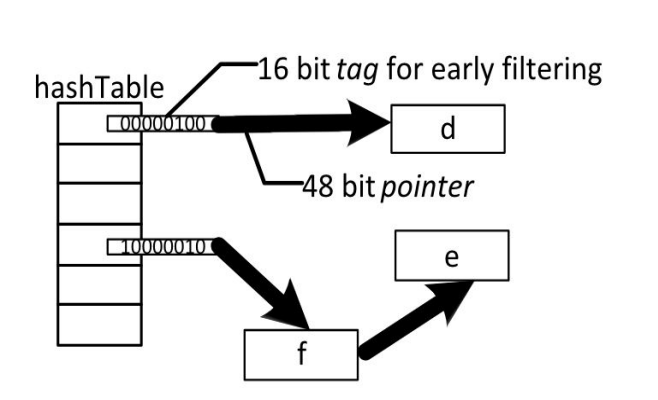
select from
F, A, P, C
where
F.date in [...]
F.aid = A.aid
...

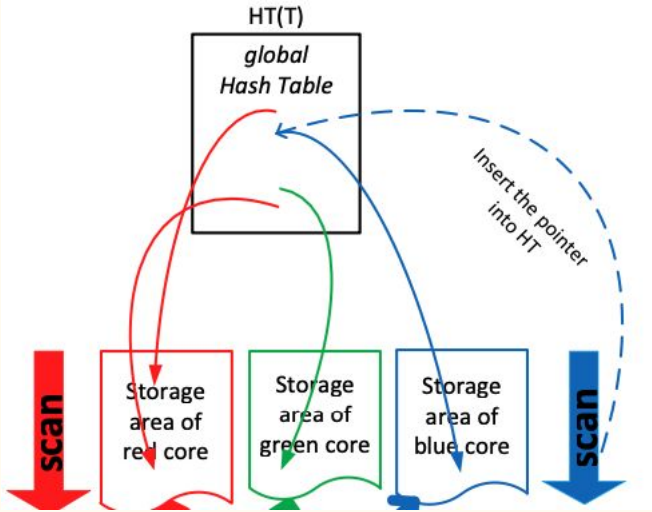


Lock-Free Tagged Hash Table

Second phase: scan storage and insert pointers using the atomic compare-and-swap instruction.

- Early-filtering optimization(16 bit tag and 48 bit pointer)
  - save space
  - single atomic compare-and-swap operation

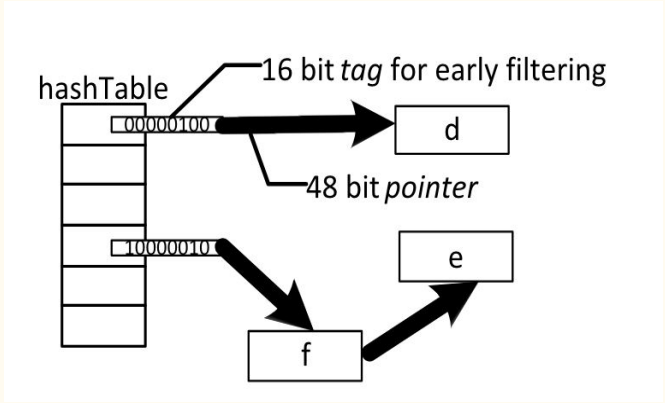




Insert entry to tagged hash table

```
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



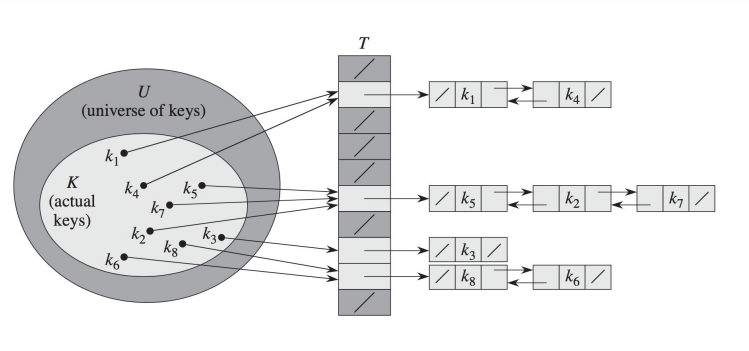
CAS is used to set pointer to the new element

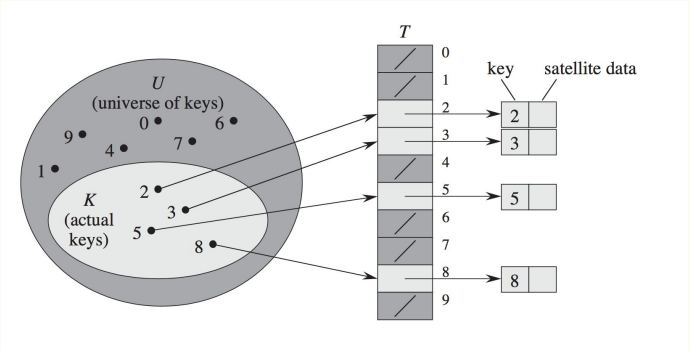
CAS fails when another insert occurred simultaneously.

Hash table stores pointers instead of tuples

chaining vs. open addressing

- save space(tuples are much larger than pointers)
- chaining allows for tuples of variable size, which is not possible for open addressing
- probe misses are faster





Tag VS. Bloom filter

- Bloom filter:
  - It is additional data structure and incurs multiple memory read.
  - Size must be proportional to hash table size to be effective. For large tables, bloom filter may not fit into cache.
- Tag:
  - Only 16 bits, less space. No unnecessary access to memory.
  - Only a small number of cheap bitwise operation.

Table Partitioning

In order to scan tables, relations need to distribute

- Round-Robin assignment
- partition relations by important attribute(primary key, foreign key).
  - relations are co-located for frequent join so there's less cross-socket communication
  - matching tuples are usually on the same socket

Aggregation

Many groups, many cache misses => contention from parallel access

- local pre-aggregation
  - aggregate NUMA-local morsel data to fixed-size thread-local hashtable ht
  - flush to thread-local partitions
- partition-wise aggregation
  - every thread scan the partitions and aggregate into a final thread-local hash table HT

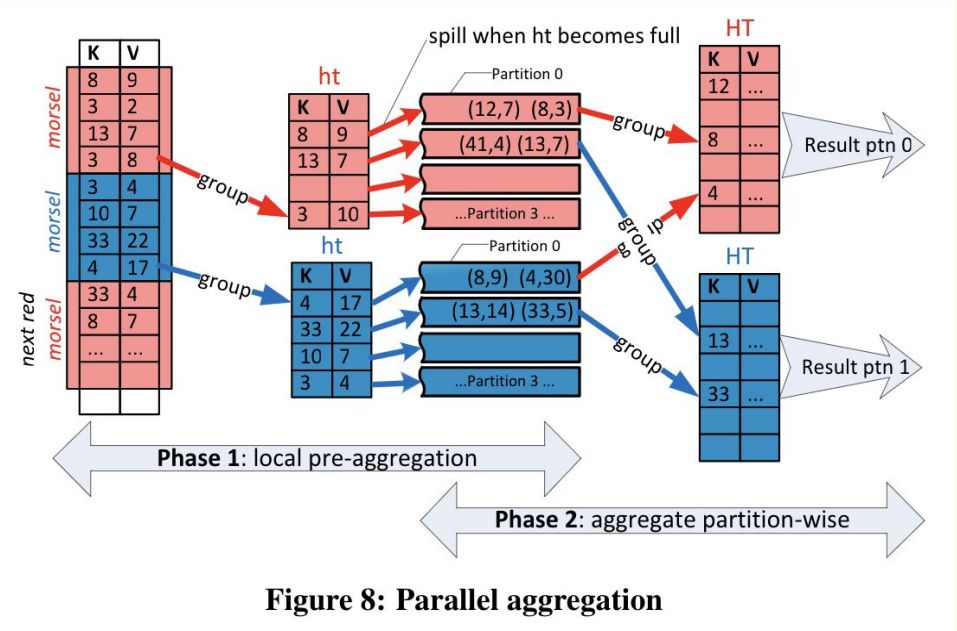
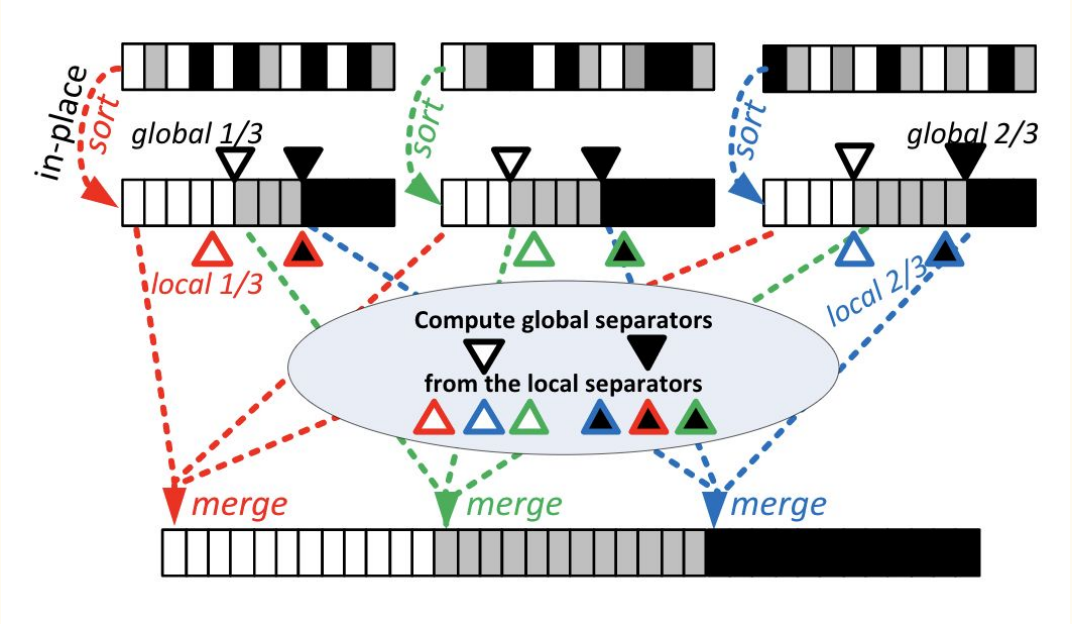


Figure 8: Parallel aggregation

Sorting

- local sort in each thread
- get the local separator keys and find the global separator keys (median-of-medians algorithm)
- parallel merge the runs into a final output array

top-k queries(heap)



Evaluation

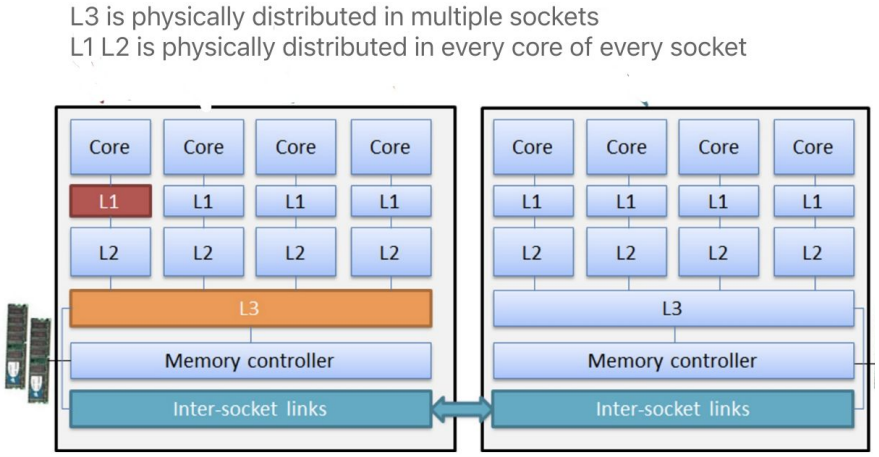
- TPC-H Experiment
- Additional Exps For Importance Of NUMA-Awareness
- Elasticity Experiment

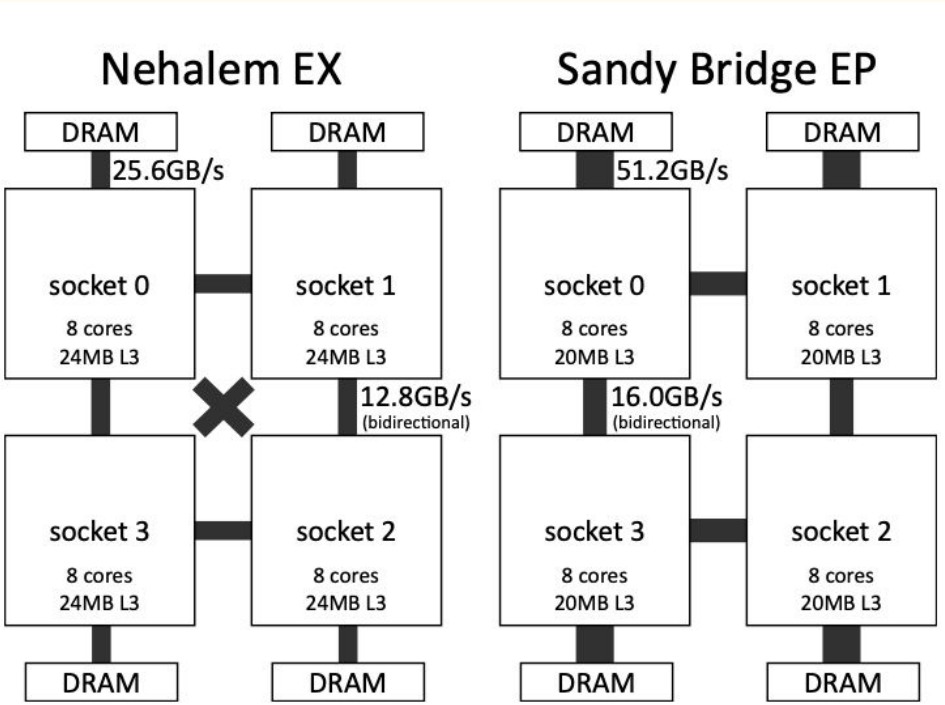
Experiment - Setup

- **Dataset**: TPC-H 100G
- **Queries: Only use primary key** (no index), results mainly measure the performance and scalability of the table **scan, aggregation, and join** (including outer, semi, anti join) operators
- **Competitors:** HyPer (use column-base storage) vs. Vectorwise (resembles a classical Volcano-style engine plus vector processing)
- **Hardware platform:**
  - Nehalem EX and some with Sandy Bridge EP : different NUMA topology leads to memory accesses (e.g., from socket 0 to socket 2 in Sandy Bridge EP require two hops instead of one)
  - 32 cores, 64 hardware threads (core 1-32 real, 33-64 virtual)

Inter-connect

- Remote: shows the percentage of data being accessed through the interconnects
- QPI: show the utilization of the most-utilized interconnect link





Experiment - Analysis

- **Scal:** “Speedup”, speedup in latency, help to evaluate the scalability on multi-threads system
- **rd. (read), wr. (write)**: Throughput
- **Remote**: measures the locality of each query. Exp: 1% in Query 1, indicates most memory accesses are local
- **QPI**(Interconnect) : can become a bottleneck as remote access increases

Example: For query 1, with 32 threads, HyPer system reads 82.6GB/s, 99% of it locally, and uses 40% of the QPI link bandwidth.

	HyPer				[%]		Vectorwise				[%]	
TPC-H #	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

Experiment-Analysis

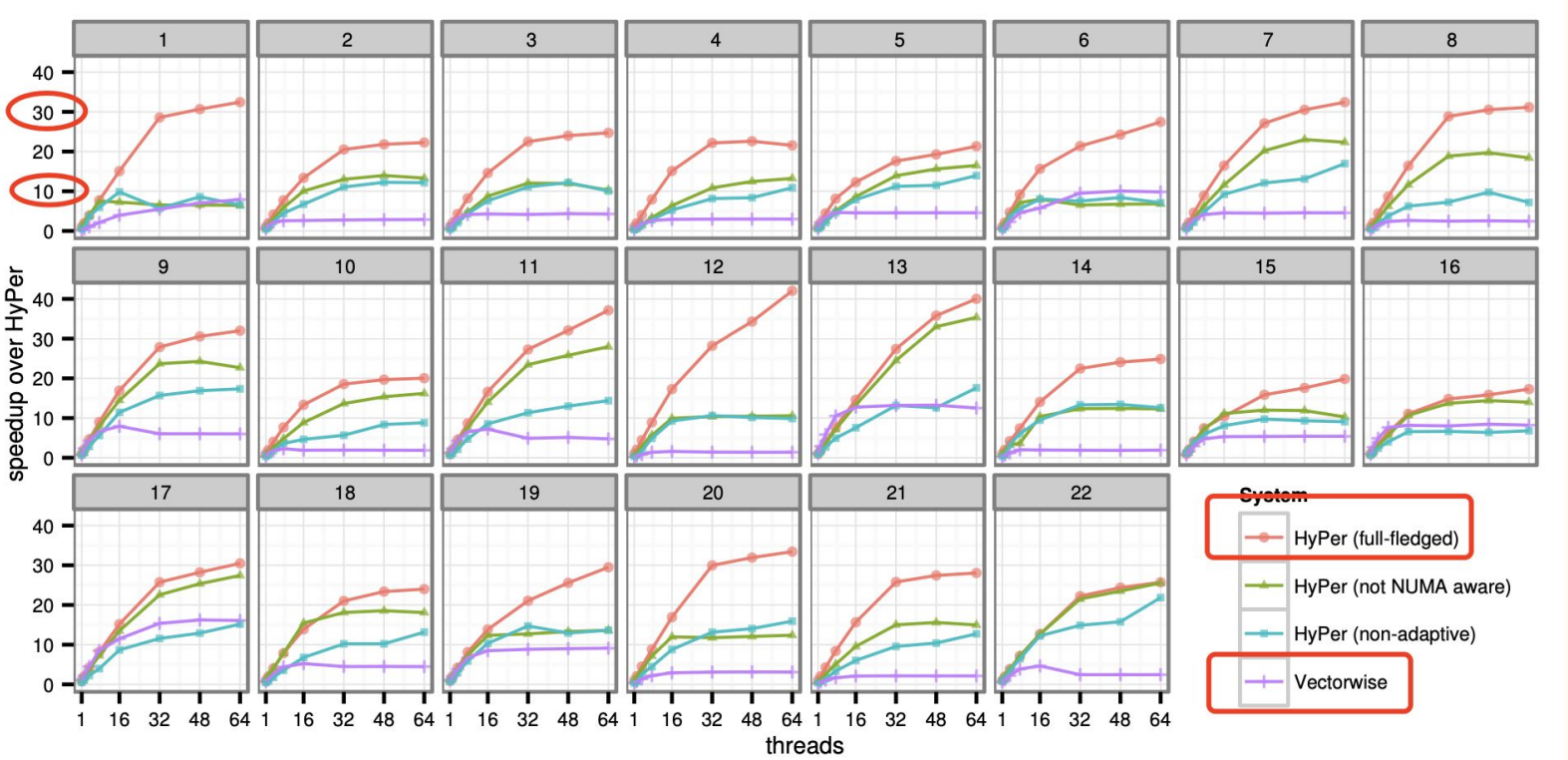


Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are "real", cores 33-64 are "virtual")

- In all Queries, **Hyper scales better than Vectorwise**, with most queries up to 30 times faster
- Full-fledged HyPer takes advantage of multiple threads better than other systems

Experiment-Analysis

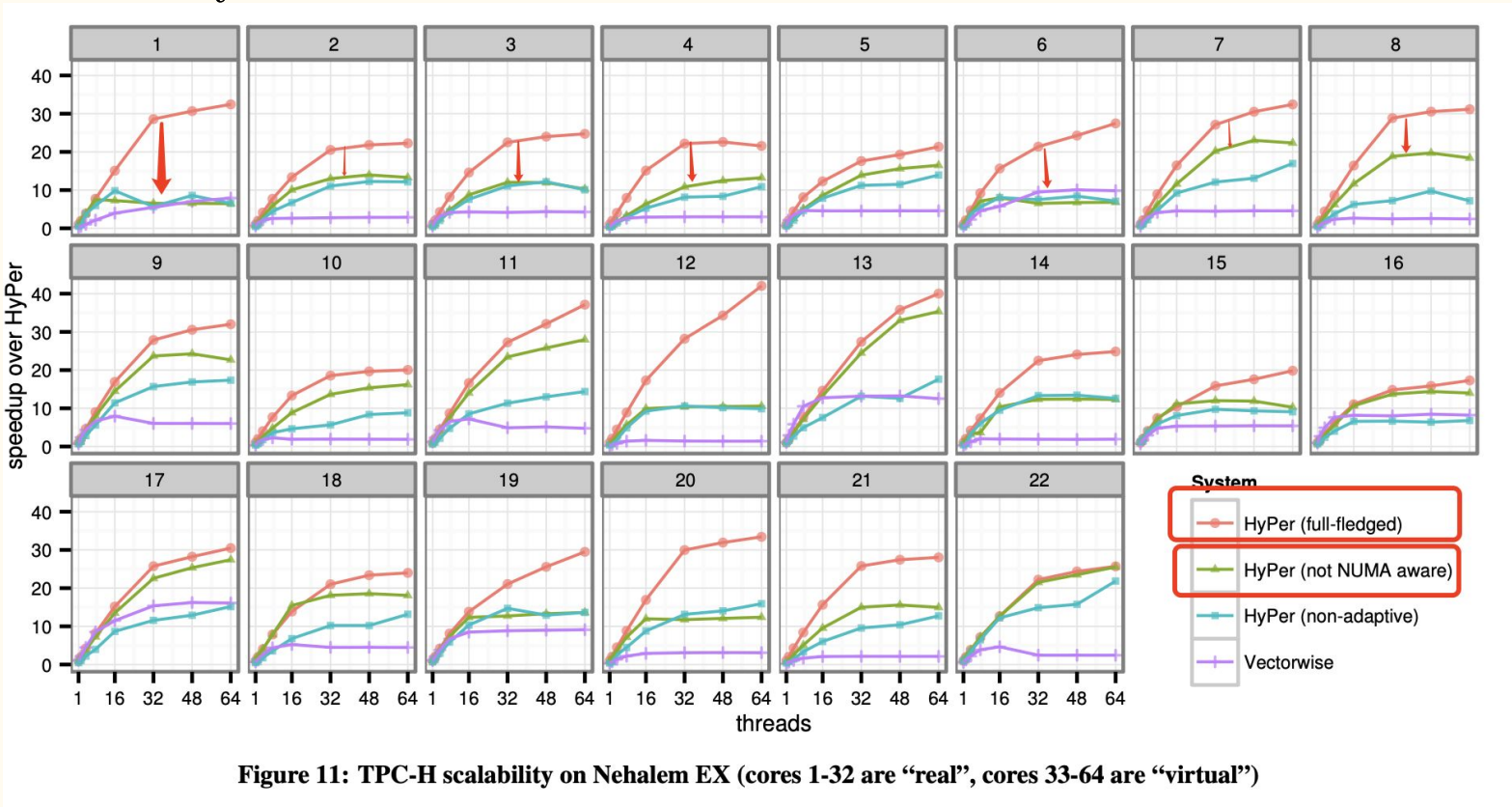


Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are "real", cores 33-64 are "virtual")

- Performance is significantly lower when we **disable NUMA-awareness** and rely on the operating system
- **Reduction of Locality results in higher latency and lower throughput**

Experiment-Analysis

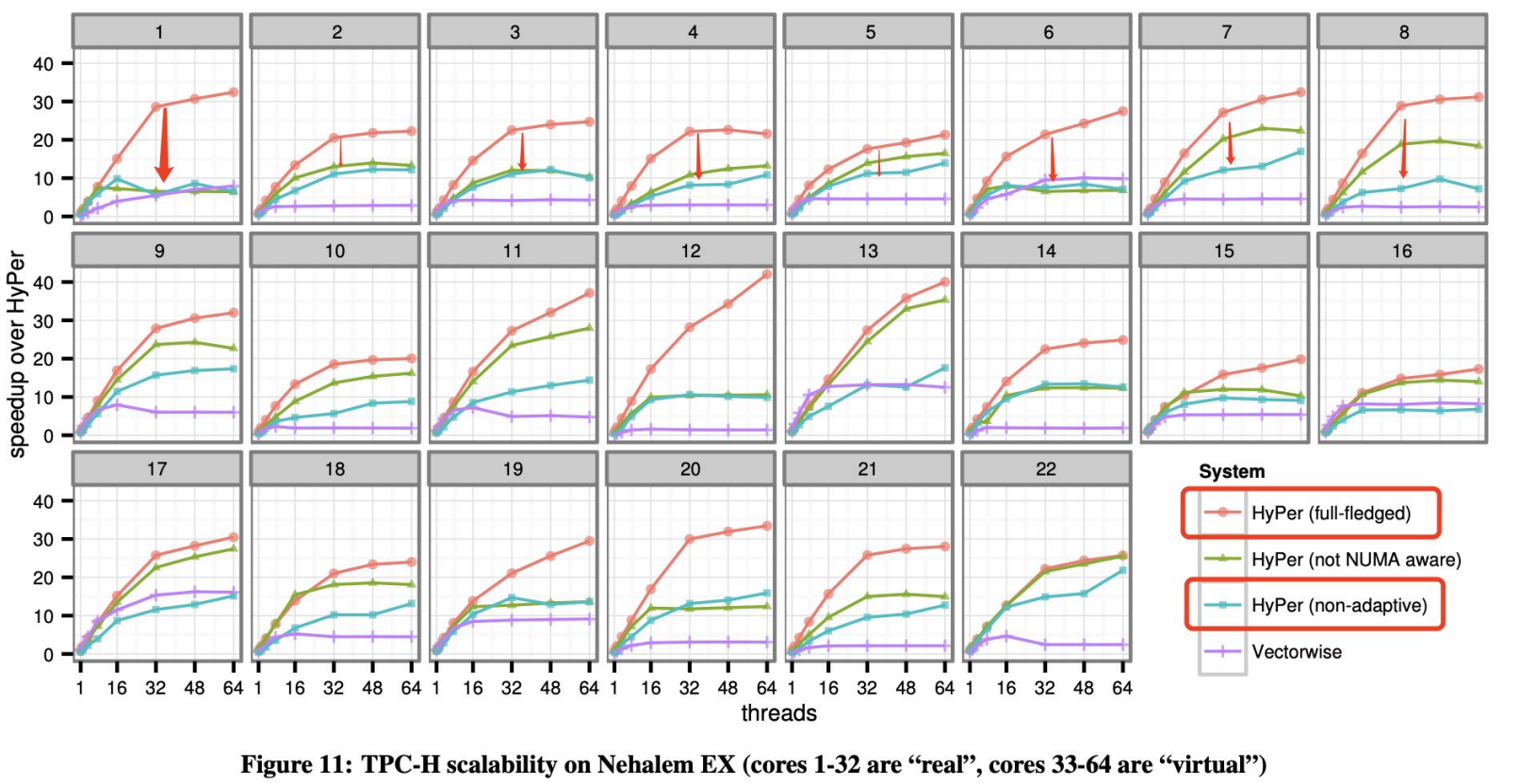
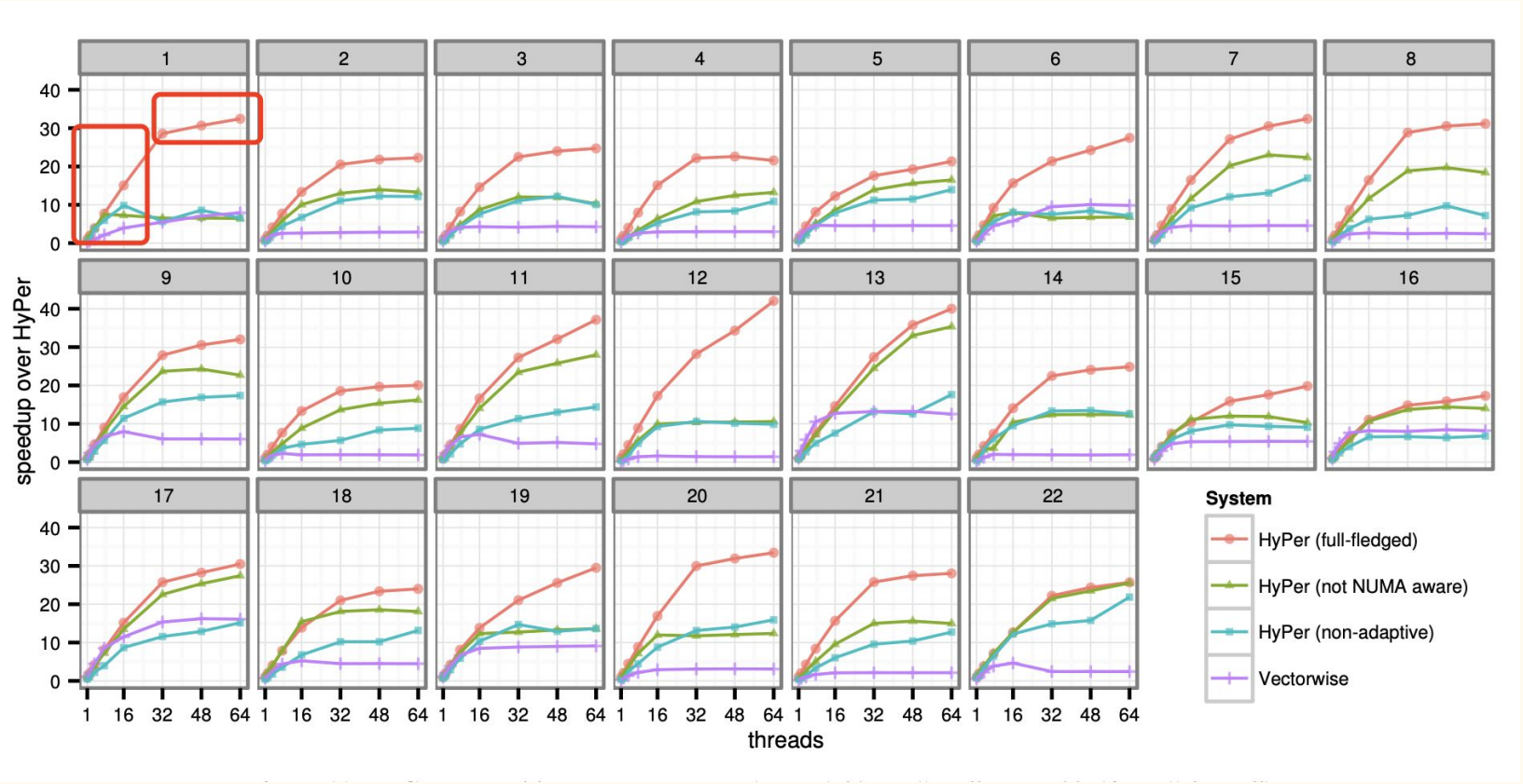


Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are "real", cores 33-64 are "virtual")

- Performance is significantly lower when we **disable adaptive morsel-wise processing**
- **Load imbalance in threads**



Why doesn't the "scal" on the 33-64 threads increases in equal proportion ?

Virtual Core

The basic steps for a CPU pipeline are:

Stage 1 (Instruction Fetch) (IF)

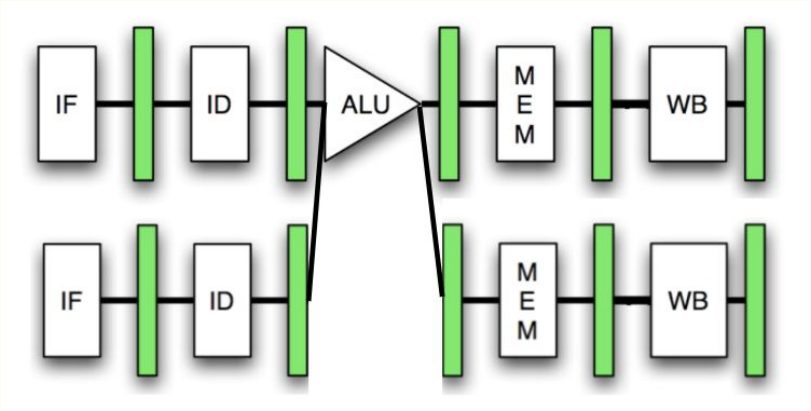
Stage 2 (Instruction Decode) (ID)

Stage 3 (Instruction Execute) (ALU):ALU stands for Arithmetic and Logical Unit and occupies the majority of the silicon on the chip.

Stage 4 (Memory Access) (MA)

Stage 5 (Write Back) (WB)

Because ALU takes so much space we can replicate all other parts of the CPU and re-use ALU.



This is going to be counted as two cores. So, we have 32 cores and 32 virtual cores with 64 hardware threads

Additional Experiment - NUMA Awareness

Remove NUMA awareness and perform memory placement by the operating system and "interleaved", where **all memory is allocated round robin over all nodes**,

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×

The geometric mean and maximum speedup of NUMA-aware approach on TPC-H

- Simply interleaving the memory is a reasonable, though not optimal strategy.
- Loss of Locality cause by disable NUMA-Awareness contributes to **more often data exchange between sockets**, and then the cross-sockets interconnect becomes the bottleneck.

Additional Experiment - NUMA Awareness

Micro Benchmark: 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

- The Sandy Bridge EP has smaller throughput and higher latency with a random mix, which means NUMA- Awareness is much more important for its good performance, because NUMA-Awareness can help it reduce data exchange between sockets.
- **The importance of NUMA-awareness clearly depends on the speed and number of the cross-socket interconnects(QPI) in the hardware platform.**

Experiment - Elasticity

- **Experiment**: varied the number of parallel queries so as to adjust threads used by each query to test whether the throughput will be interfered by other processes.

- **Feature:** fully-fledged HyPer can keep a high throughput.

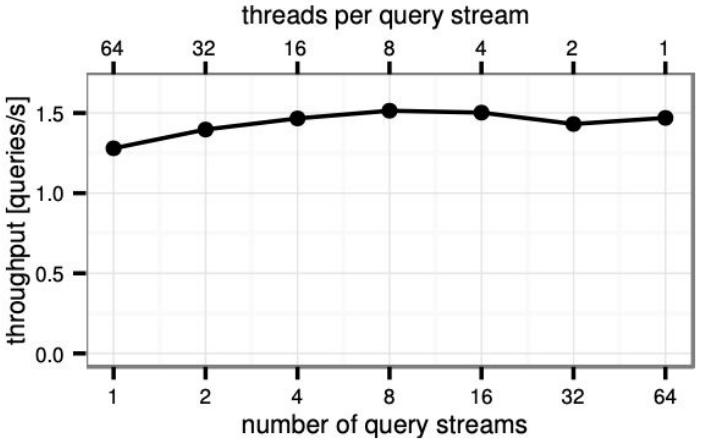


Figure 12: Intra- vs. inter-query parallelism with 64 threads

Experiment - Elasticity

- **Explain**: Fully-fledged HyPer can dynamically **re-assign worker threads to other queries** with morsel-wise processing. Morsel-wise processing ensures that we can **smoothly switch tasks on threads**, ensuring load Balance, so it can keep a stable and high throughput.

- **Example**

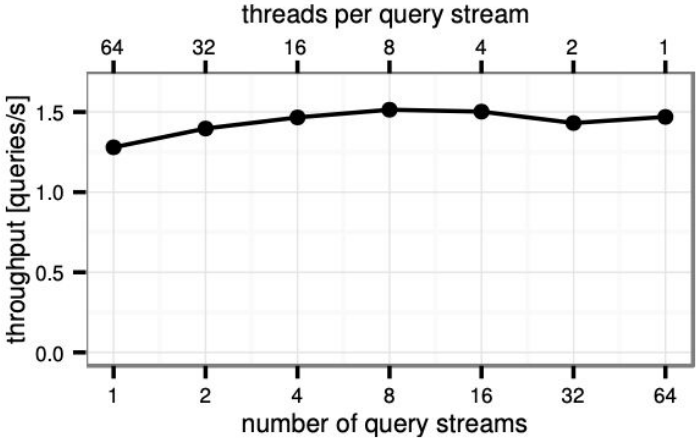


Figure 12: Intra- vs. inter-query parallelism with 64 threads

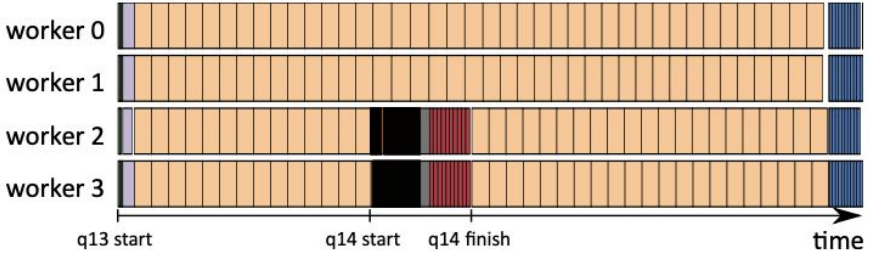


Figure 13: Illustration of morsel-wise processing and elasticity

Star Schema Benchmark

Except TPC-H, fully fledged HyPer also works well on Star Schema Benchmark with a speedup of over 40 for most queries

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

Conclusion & Future Work

- Conclusion
- Future Work
- Improvement
- Technical Question

Conclusion

Fully-fledged HyPer with **NUMA Awareness** and **morsel-wise processing** can get good scalability and make fully use of multi-core systems.

Future Work

- The design and evaluation of a scheduler, which takes quality-of-service constraints into account
- Investigate algorithms that take knowledge of the underlying hardware 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.

What the Paper Could Improve

- Should including more detail on lock-free data structures - especially the dispatcher
- Talk more details about how queries were interwoven during testing
- Should calculate overhead of scheduling

Technical Question:
The authors mention in the paper that "the morsel size is not very critical for performance". What do you think is the reason behind this? Do the authors present any experiment to support their claim?