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

Presented by Matthew Cote

Overview

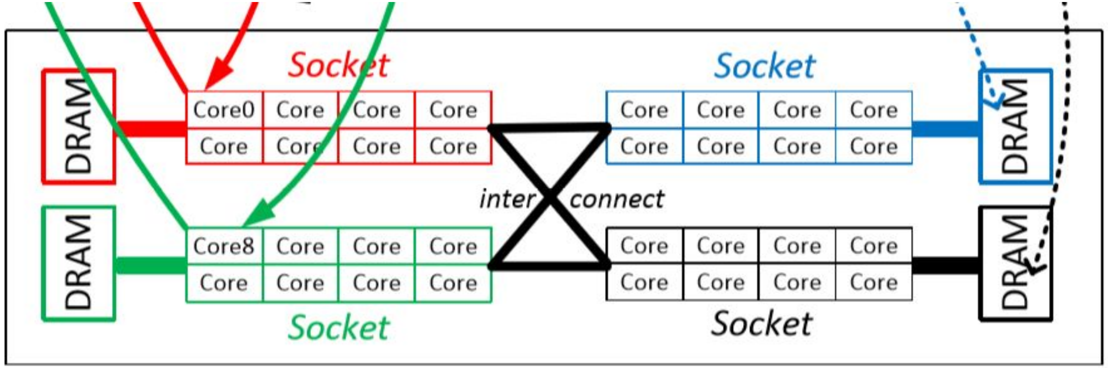
Setting the Stage:

- Rise of many-core architectures
  - More parallelism, less single-thread optimizations
  - ~10s to 100s of threads in modern architectures
- Main memory capacities have increased
  - Query processing not always I/O bound
- Rise of NUMA architecture
  - Needed to scale throughput from large memories
  - Allows multiple cores to access different memory banks simultaneously

NUMA Architecture

NUMA = Non-Uniform Memory Access

- Memory access cost varies depending on which chip the accessing thread and memory are located



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

The Problem: Part 1

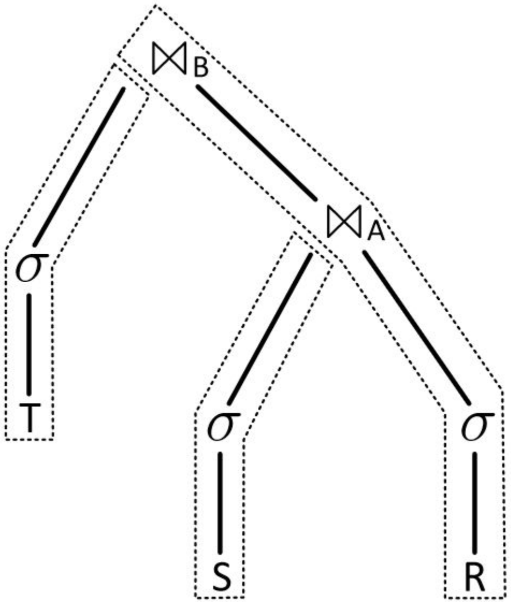
- Single threaded database applications have already been significantly optimized.
- Modern processor design focuses more on adding more cores instead of improving each core.

What about multi-core database designs?

Previous Approach to Parallelism

Parallel Volcano Model

- Operators unaware of parallelism
- Tuple streams routed among threads executing a pipeline
- Statically determine # of threads, operation for each thread



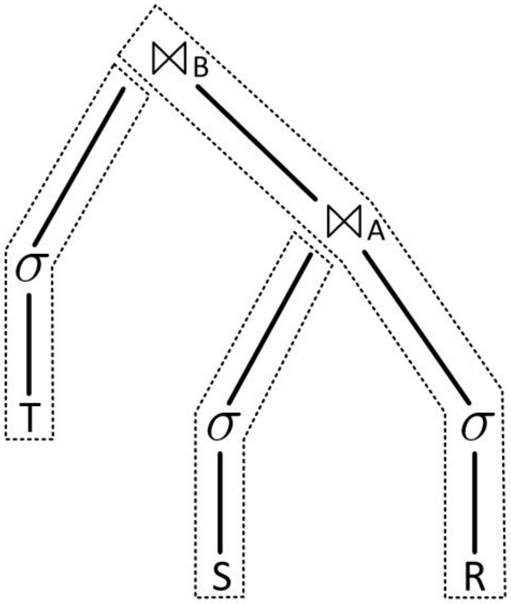
Previous Approach to Parallelism

Parallel Volcano Model

- Operators unaware of parallelism
- Tuple streams routed among threads executing a pipeline
- Statically determine # of threads, operation for each thread

Weaknesses:

- Poor load-balancing
- Not NUMA-Aware
- Not dynamically elastic



Proposed System

Morsel Driven Query Execution Framework for Main Memory Databases

- Multiple threads execute each pipeline (as before)
- Flexible and dynamic dispatcher assigns tasks at runtime
- Tasks organized by what memory they access
- Tasks work on small morsels of data

Solving the Problem

Perfect Load Balancing

- Threads for a given pipeline finish at the same time
- Possible due to morsel-size work increments and work stealing

NUMA - Awareness

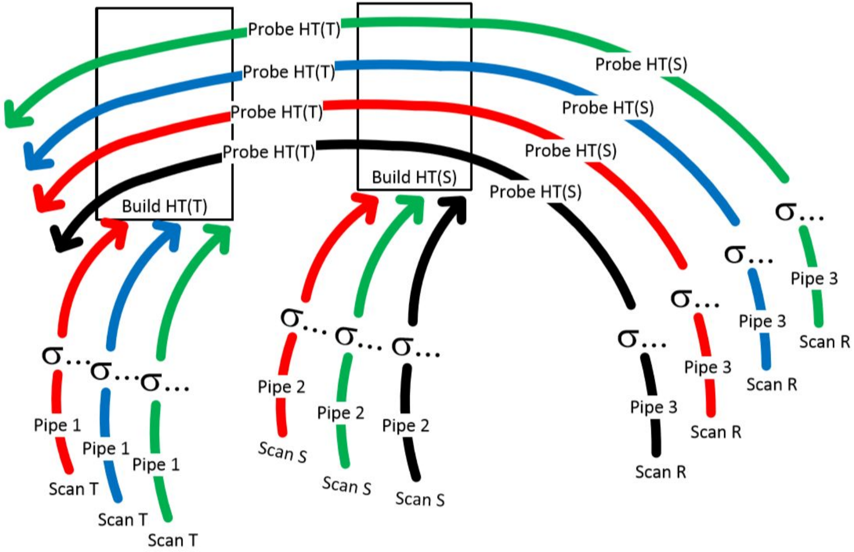
- Thread (primarily) reads/writes to memory in own socket

Elasticity

- Dispatcher determines at run-time what each thread will work on
- Threads can be moved to other pipelines upon finishing a morsel

Depiction of Three-Way Hash Join

- Color = Socket
- Line = Thread
- Line Group = Pipeline



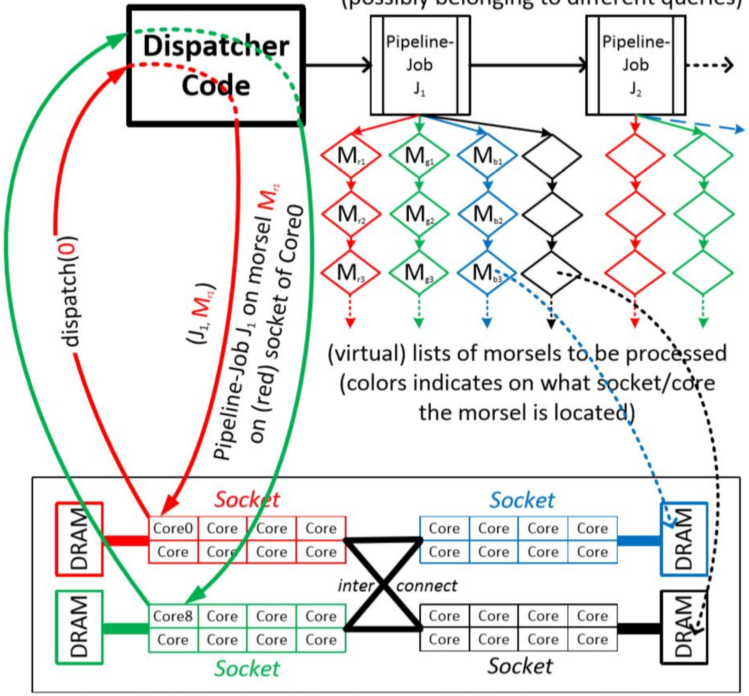
Implementation Details

Execution Overview: Scheduling Pipelines

- QEPobject controls the execution of a query
  - Transfers executable pipelines to the dispatcher
    - Only pipelines with no dependencies
  - Allocates storage for results from threads
  - Creates new morsels from results of past operations

Scheduling Threads

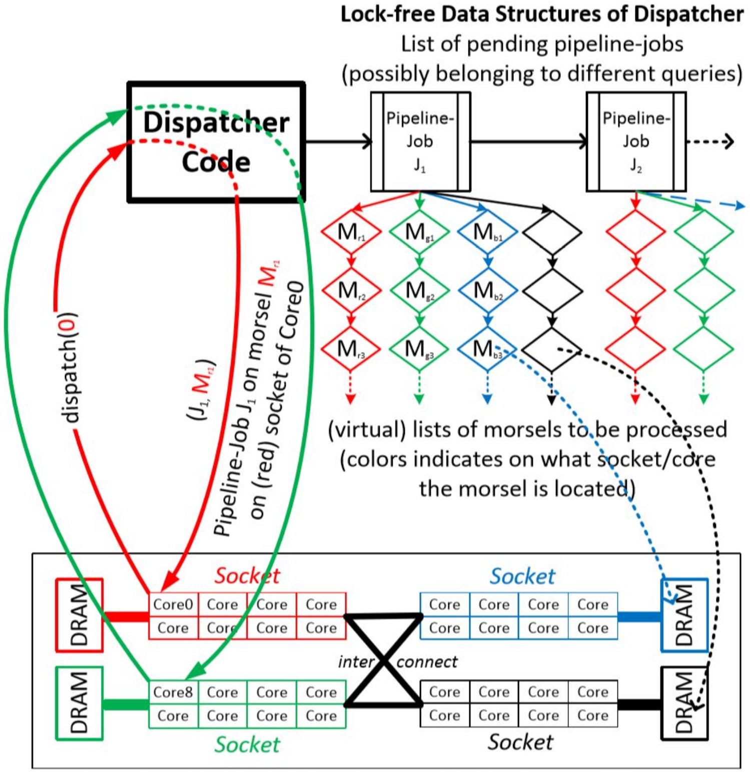
- Dispatcher has list of available pipelines
- Allows for inter-query parallelism
- Pipeline has list of morsels to process, one list per memory bank.
- Assigns each thread a task (pipeline job + morsel)
- Dispatcher = lock free, run by the work-requesting thread.



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

Scheduling Threads

- Queries dynamically assigned threads
- Preemption occurs at morsel boundary
- All cores working on a pipeline finish at same time
    - Prevents fast threads from idling
    - Requires work stealing



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

Solving the Problem

Perfect Load Balancing

- Threads for a given pipeline finish at the same time
- Possible due to morsel-size work increments and work stealing

NUMA - Awareness

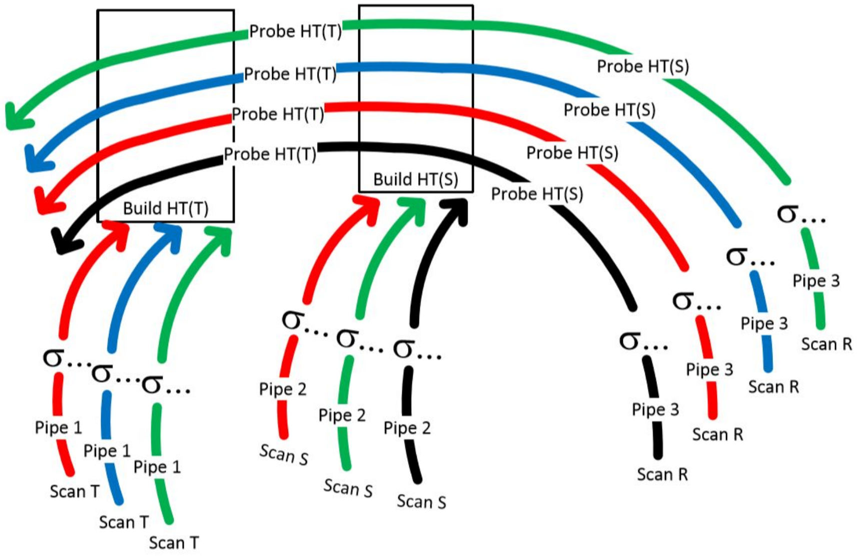
- Thread (primarily) reads/writes to memory in own socket

Elasticity

- Dispatcher determines at run-time what each thread will work on
- Threads can be moved to other pipelines upon finishing a morsel

Example Query Execution

Three Way Hash Join



Important First Step: Distributing a Table

- Tables partitioned semi-evenly among the memory banks
- Partition by hashing the primary key/foreign key

Building the Hash Table

- Threads processes morsel at a time
  - Write to own socket's memory
- All threads finish phase one before going to next.
- Hash table = distributed, lock-free, perfect size

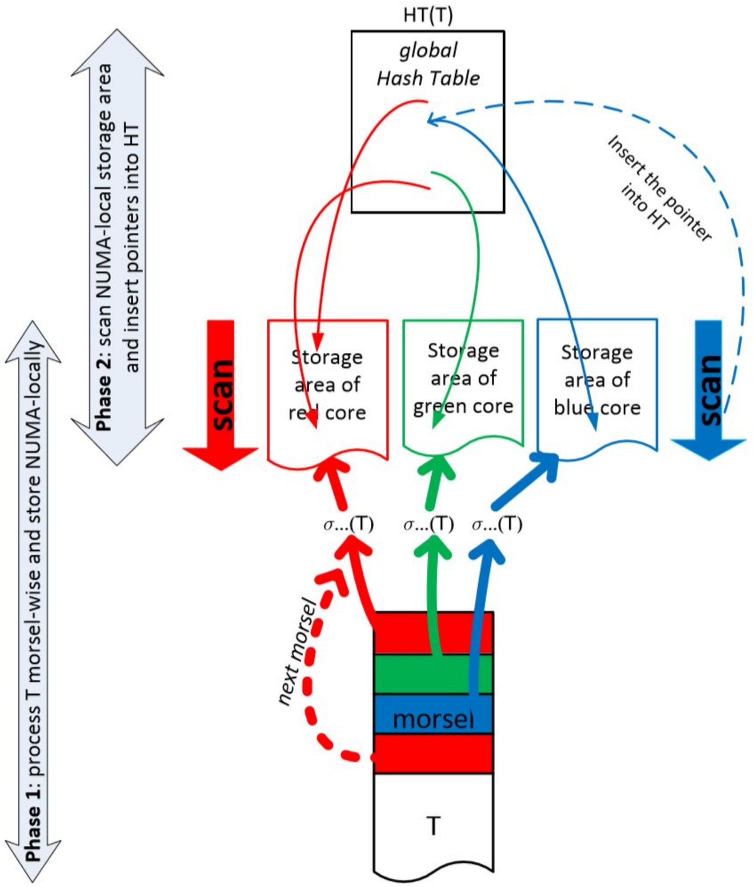
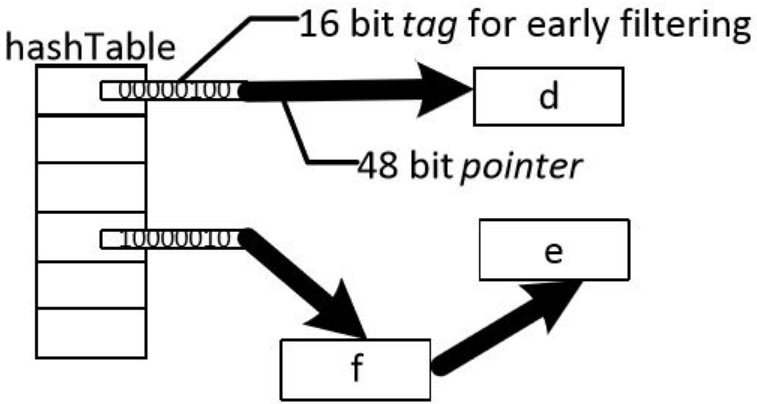


Figure 3: NUMA-aware processing of the build-phase

Building the Hash Table

- Each hash bucket has 16 bit hash tag summarizing contents.
- Probe checks tag before searching list.
- Compare and Set used to atomically grow the hash table



Finishing the Join

Repeat similar steps as before, but now probing the hash tables.

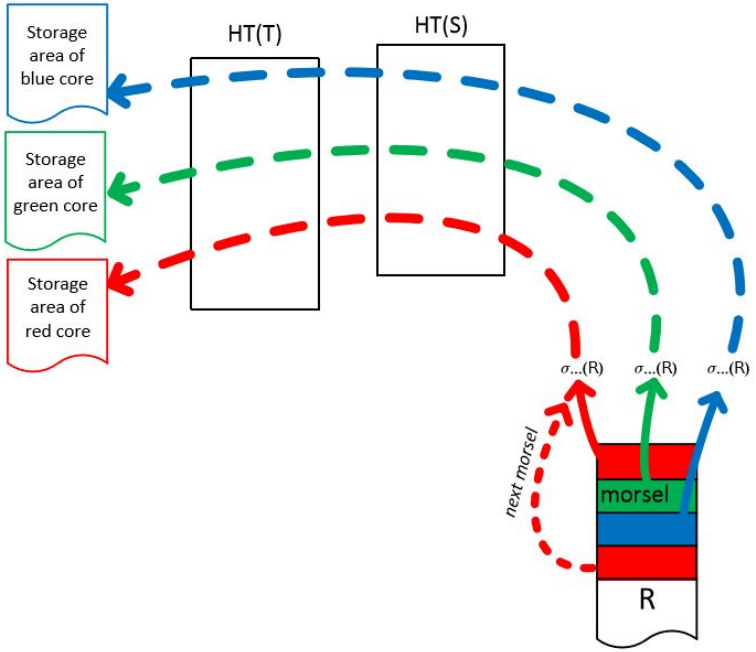


Figure 4: Morsel-wise processing of the probe phase

Testing

Evaluation Technique

Two different architectures

Different NUMA topologies/BWs

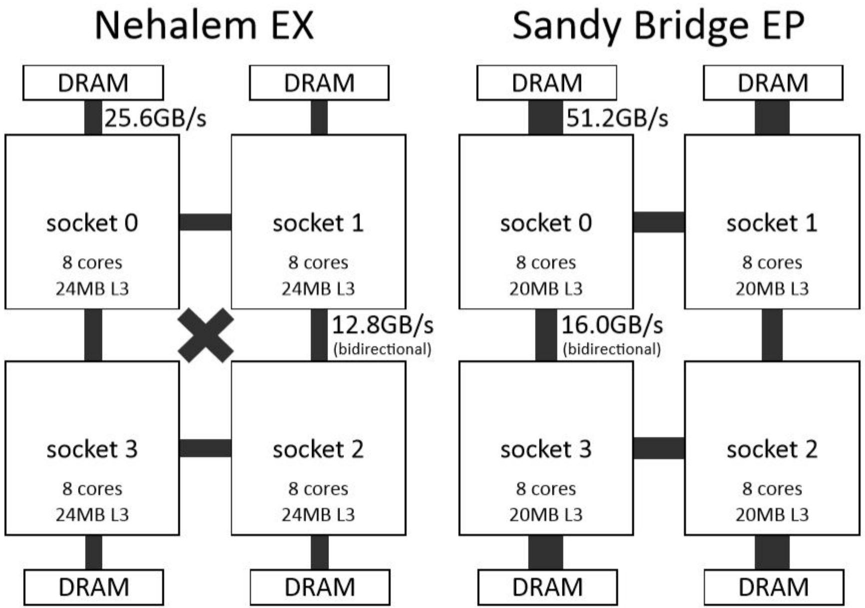


Figure 10: NUMA topologies, theoretical bandwidth

Performance Evaluation

Normalized by execution of single-threaded HyPer

Primarily compared with Vectorwise

Significantly faster than PostgreSQL and commercial column store

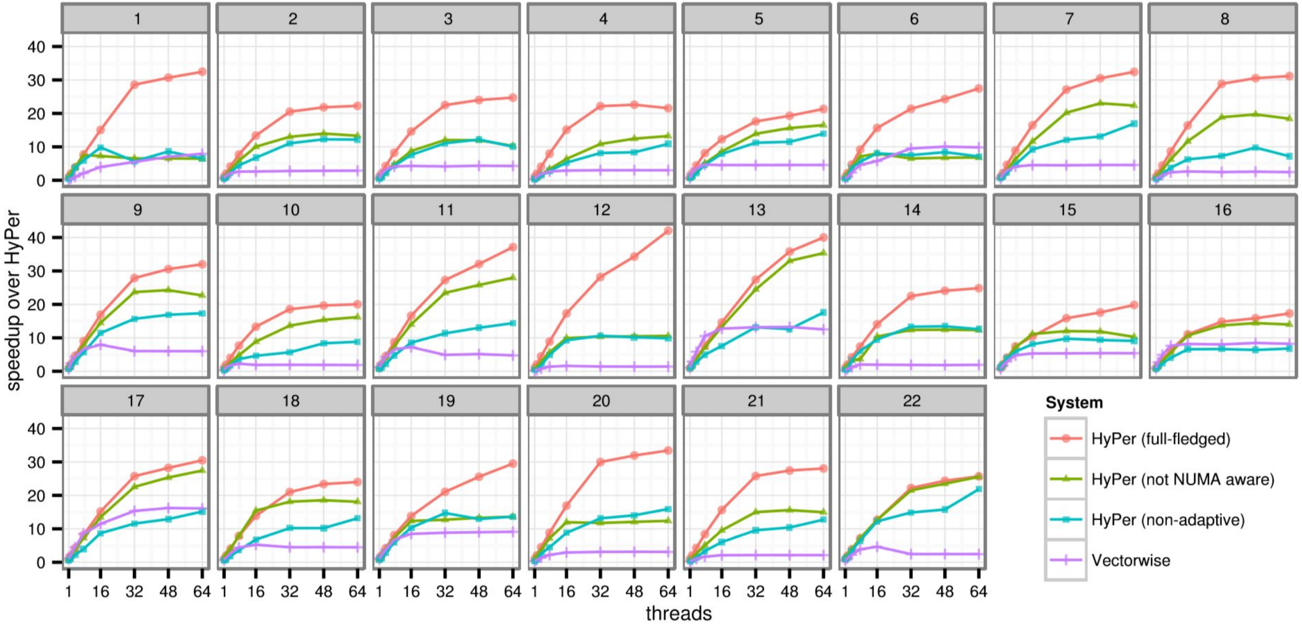


Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are "real", cores 33-64 are "virtual")

NUMA Awareness Evaluation

Near maximum bandwidth of 100 GB/s for queries 1 and 6.

% Remote is very low - indicates most memory accesses are local

QPI is lower - indicating lower congestion on most used link

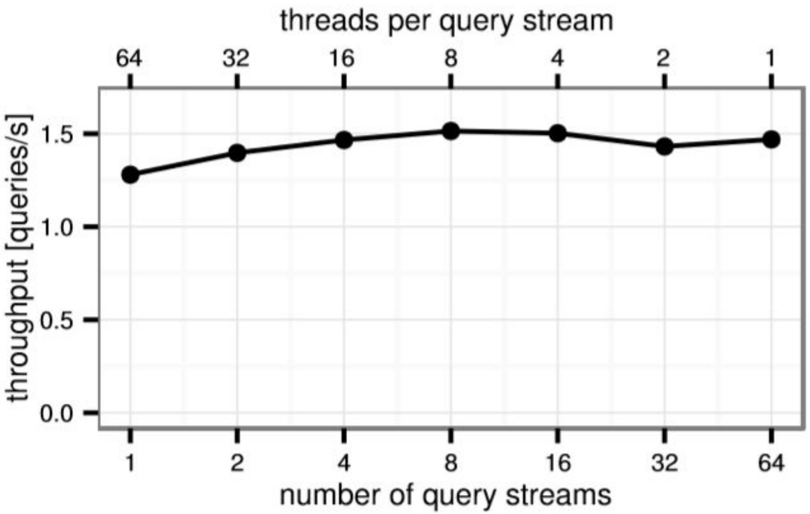
TPC-H	HyPer				[%]		Vectorwise				[%]	
	time	scal.	rd.	wr.	remote		time	scal.	rd.	wr.	remote	
#	[s]	[×]	[GB/s]			QPI	[s]	[×]	[GB/s]			QPI
1	0.28	32.4	82.6	0.2	1	40	1.13	30.2	12.5	0.5	74	7
2	0.08	22.3	25.1	0.5	15	17	0.63	4.6	8.7	3.6	55	6
3	0.66	24.7	48.1	4.4	25	34	3.83	7.3	13.5	4.6	76	9
4	0.38	21.6	45.8	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

Elasticity Evaluation

System works for a large range of number of query streams

Demonstrates threads can be dynamically scheduled and load balanced



Evaluation of the Paper

What the Paper Did Well

- Provided concrete example to follow throughout
- Used helpful and illustrative diagrams
- Clearly stated the weakness of old systems and how this system fixes it
- Justified most architecture decisions in detail and discussed alternatives
- Created experiments to test the key optimizations

What the Paper Could Improve:

- Discuss potential applications to non in-memory databases
- Including priority based scheduling
- Including more detail on lock-free data structures - especially the dispatcher
- Talk about how queries were interwoven during testing
- Present weaknesses of this system - no real sense of a trade-off
- Calculate overhead of scheduling