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

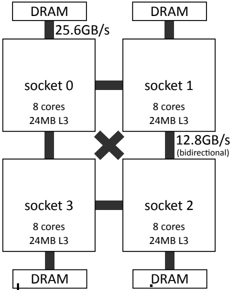
Viktor Leis, Peter Boncz*, Alfons Kemper, Thomas Neumann

Technische Universität München *CWI

with some modifications by: S. Sudarshan

Introduction

- Number of CPU cores keeps growing:
  4-socket Ivy Bridge EX with 60 cores, 120 threads, 1TB RAM (50,000$)
- These systems support terabytes of NUMA RAM: disk is not a bottleneck
- For analytic workloads intra-query parallelization is necessary to utilize such systems



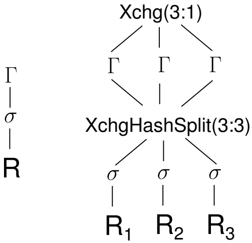
- Number of CPU cores keeps growing:
  4-socket Ivy Bridge EX with 60 cores, 120 threads, 1TB RAM (50,000$)

Contributions

- We present an architectural blueprint for a query engine incorporating the following
  - Morsel-driven query execution (work is distributed between threads dynamically using work stealing)
  - Set of fast parallel algorithms for the most important relational operators
  - Systematic approach to integrating NUMA-awareness into database systems
- Lots of prior work on algorithms for main-memory databases
  - Focus on storage, and on individual operations (hash join, merge join, aggregation, ...)
  - NUMA has been addressed by quite a few papers
  - Focus of this paper is on efficiently evaluating a full query, and on algorithms that support pipelined evaluation

Related Work: Volcano-Style Parallelism (1)

- *Encapsulation of Parallelism in the Volcano Query Processing System*, Goetz Graefe, SIGMOD 1990
  SIGMOD Test of Time Award 2000
- *Plan-driven* approach:
  - optimizer statically determines at query compile time how many threads should run
  - instantiates one query operator plan for each thread
  - connects these with exchange operators, which encapsulate parallelism and manage threads
- Elegant model which is used by many systems

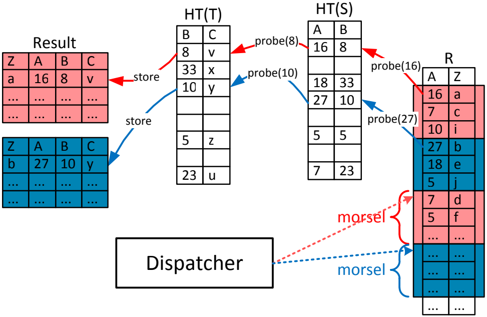


Volcano-Style Parallelism (2)

- + Operators are largely oblivious to parallelism
- + Great for shared-nothing parallel systems
- – But can do better for shared memory parallel systems with all data in-memory
- – Static work partitioning can cause load imbalances
- – Degree of parallelism cannot easily be changed mid-query
- – Not NUMA aware
- – Overhead:
    - Thread oversubscription causes context switching
    - Hash re-partitioning often does not pay off
    - Exchange operators create additional copies of the tuples

Morsel-Driven Query Execution (1)

- Break input into constant-sized work units (“morsels”)
- Dispatcher assigns morsels to worker threads
- # worker threads = # hardware threads
- Operators are designed for parallel execution



Result

Z	A	B	C
a	16	8	v
...	...	...	...
...	...	...	...

Z	A	B	C
b	27	10	y
...	...	...	...
...	...	...	...

HT(T)

B	C
8	v
33	x
10	y
5	z
23	u

HT(S)

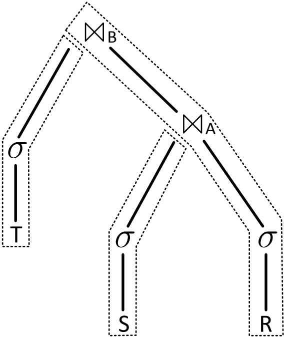
A	B
16	8
18	33
27	10
5	5
7	23

R

A	Z
16	a
7	c
10	i
27	b
18	e
5	j
7	d
5	f
...	...
...	...
...	...
...	...
...	...

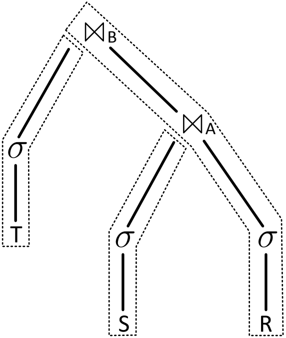
Morsel-Driven Query Execution (2)

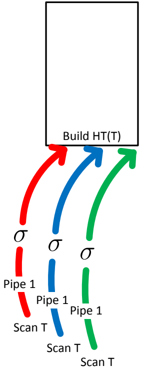
- Each pipeline is parallelized individually using all threads



Morsel-Driven Query Execution (2)

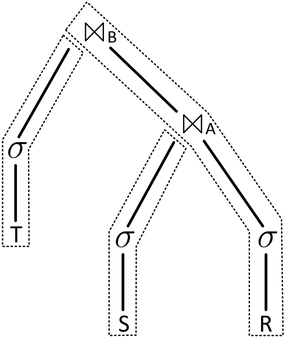
- Each pipeline is parallelized individually using all threads

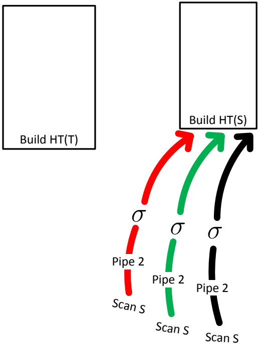




Morsel-Driven Query Execution (2)

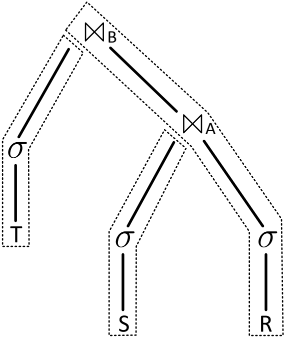
- Each pipeline is parallelized individually using all threads

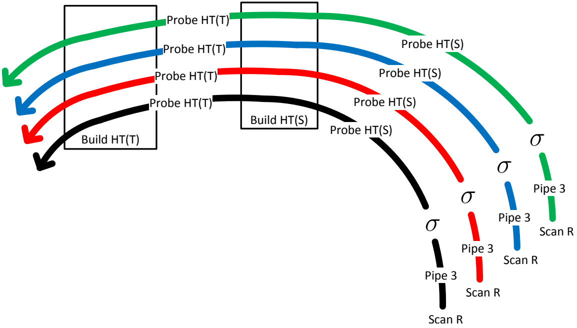




Morsel-Driven Query Execution (2)

- Each pipeline is parallelized individually using all threads

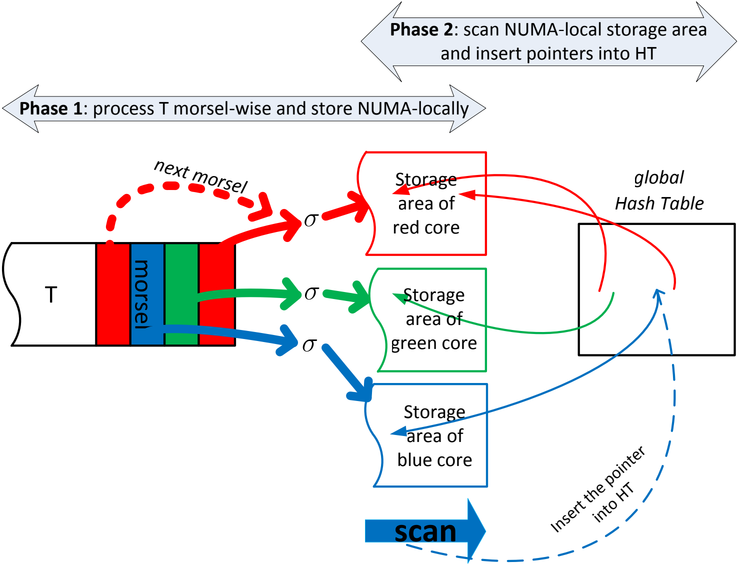




Parallel In-Memory Hash Join

1. Several algorithms proposed earlier for parallel in-memory hash join
2. Option 1: partition relation and process each partioning in parallel
3. Option 2: build a global hash table on build relation, but parallellize both building and probing
4. Earlier work shows Option 2 is better
5. Key issues: maximize locality, minimize synchronization

NUMA-aware Processing of Build Phase



Phase 2: scan NUMA-local storage area and insert pointers into HT

Phase 1: process T morsel-wise and store NUMA-locally

next morsel

T

morsel

σ

σ

σ

Storage area of red core

Storage area of green core

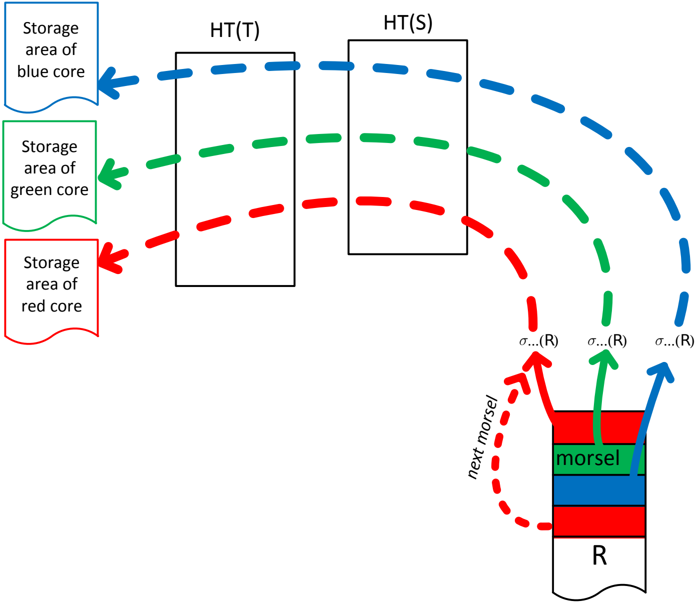
Storage area of blue core

global Hash Table

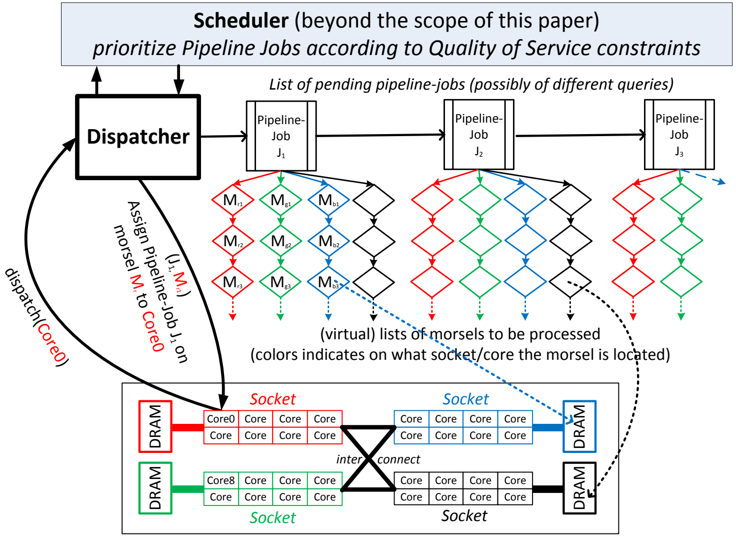
scan

Insert the pointer into HT

Morsel-Wise Processing of Probe Phase



Dispatcher



Scheduler (beyond the scope of this paper)

prioritize Pipeline Jobs according to Quality of Service constraints

List of pending pipeline-jobs (possibly of different queries)

(virtual) lists of morsels to be processed

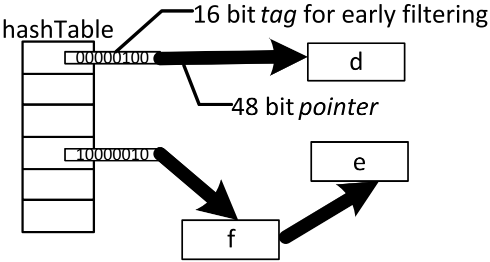
(colors indicates on what socket/core the morsel is located)

Assign Pipeline-Job J_1 on morsel M_r to Core0

dispatch(Core0)

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

Hash Table



- Unused bits in pointers act as a cheap bloom filter

Lock-Free Insertion into Hash Table

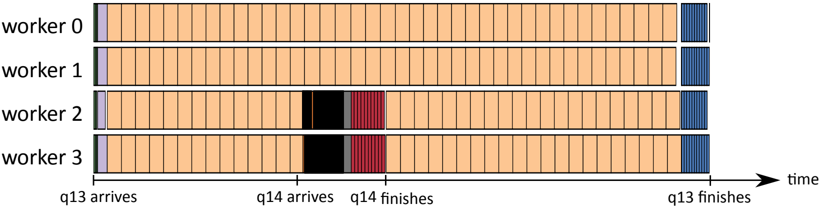
```
insert(entry) {
// determine slot in hash table
        slot = entry->hash >> hashTableShift
        do {
             old = hashTable[slot]
             // set next to old entry without tag
             entry->next = removeTag(old)
             // add old and new tag
             new = entry | (old&tagMask)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

Storage Implementation

1. Use large virtual memory pages (2MB) both for the hash table and the tuple storage areas.
   1.1 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.
2. Allocate the hash table using the Unix mmap system call, if available.
   2.1 Page gets allocated on first write, initialized to 0's
   2.2 Pages located on same NUMA node as thread that first writes the page, ensuring locality if only single NUMA node is used.
3. May be a good idea to partition table using primary/foreign key
   3.1 e.g. order and lineitem on orderkey

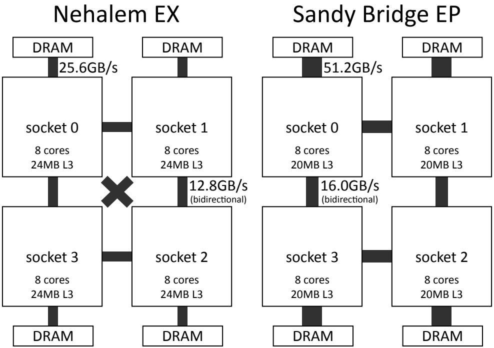
Morsels

- No load imbalances: all workers finish very close in time
- Morsels allow to react to workload changes: priority-based scheduling of dynamic workloads possible



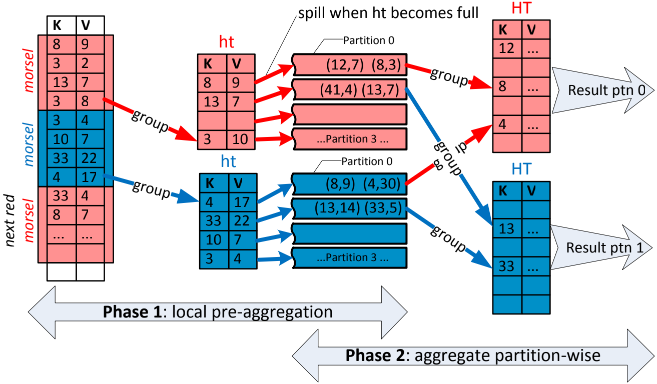
NUMA Awareness

- NUMA awareness at the morsel level
- E.g., Table scan:
  - Relations are partitioned over NUMA nodes
  - Worker threads ask for NUMA-local morsels
  - May steal morsels from other sockets to avoid idle workers



Parallel Aggregation

- Aggregation: partitioning-based with cheap pre-aggregation
- Stage 1: Fixed size hash table per thread, overflow to partitions
- Stage 2: Final aggregation: thread per partition



morsel

morsel

next red morsel

K	V
8	9
3	2
13	7
3	8
3	4
10	7
33	22
4	17
33	4
8	7
...	...

group

group

ht

K	V
8	9
13	7
3	10

ht

K	V
4	17
33	22
10	7
3	4

spill when ht becomes full

Partition 0

(12,7) (8,3)

(41,4) (13,7)

...Partition 3 ...

Partition 0

(8,9) (4,30)

(13,14) (33,5)

...Partition 3 ...

group

group

group

group

HT

K	V
12	...
8	...
4	...

HT

K	V
13	...
33	...

Result ptn 0

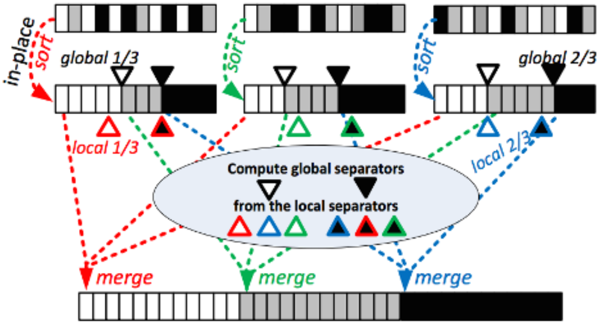
Result ptn 1

Phase 1: local pre-aggregation

Phase 2: aggregate partition-wise

Parallel Merge Sort

- Sorting for order by and top-K only, sorting for merge join not efficient
- Local sort in parallel, followed by parallel merge
- Key issue: finding exact separators. Median-of-medians algo.



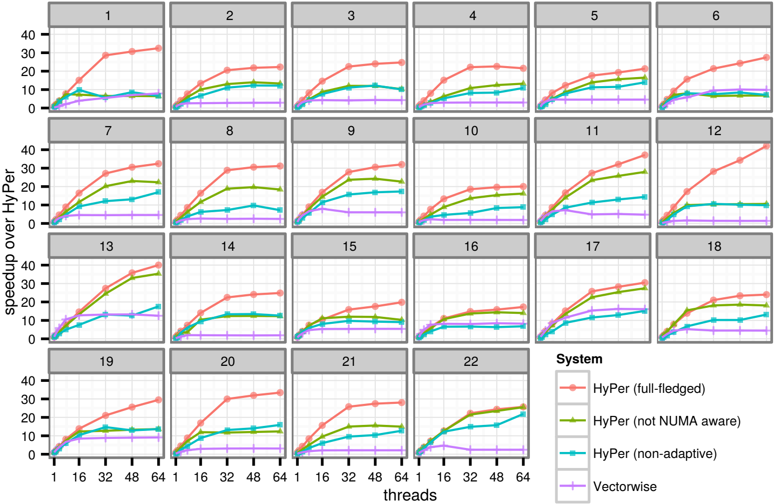
Evaluation: TPC-H (SF 100), Nehalem EX (32 cores)

TPC-H #	time [s]	speedup
1	0.28	32.4
2	0.08	22.3
3	0.66	24.7
4	0.38	21.6
5	0.97	21.3
6	0.17	27.5
7	0.53	32.4
8	0.35	31.2
9	2.14	32.0
10	0.60	20.0
11	0.09	37.1

TPC-H #	time [s]	speedup
12	0.22	42.0
13	1.95	40.0
14	0.19	24.8
15	0.44	19.8
16	0.78	17.3
17	0.44	30.5
18	2.78	24.0
19	0.88	29.5
20	0.18	33.4
21	0.91	28.0
22	0.30	25.7

- single threaded: 30x faster than PostgreSQL, 10x faster than commercial column store, similar speed as Vectorwise
- multi threaded: 5x faster than Vectorwise, 50x faster than Cloudera Impala on 20-node cluster

Scalability



Conclusions

- Getting good scalability and performance on many-core systems is challenging but possible
- However, it not possible to bolt on parallelism to an existing query engine, one must redesign it with modern hardware in mind
- With morsel-driven parallelism HyPer can finish ad hoc queries on hundreds of GBs in seconds

`www.hyper-db.com`