Performance in the Multicore Era

Gustavo Alonso
Performance in the multicore era
Most of the slides prepared by Cagri Balkesen

Work in this talk mostly done in collaboration with Cagri Balkesen, Jens Teubner, and Tamer Ozu (ICDE 2013)

Work part and/or related to
- Avalanche Project (Swiss SNF) – J. Teubner
- SwissBox Project (ECC) – G. Alonso
- Rapid Project (Oracle Research) – C. Balkesen, G. Alonso
SWISSBOX

Gustavo Alonso, Donald Kossmann, Timothy Roscoe: SwissBox: An Architecture for Data Processing Appliances. CIDR 2011: 32-37

Performance in the multicore era
The SwissBox project

- Build an open source data appliance
  - Hardware
  - Software

- What is a DB appliance?
  - Database in a box
    - Funny database
    - Funny box
The many sides to SwissBox

- From multicore to clusters
- Main memory
  - Crescando
  - SharedDB
- Hardware acceleration
  - FPGAs on data path (intelligent storage)
  - Specialized processors
- DB/OS codesign
  - Resource allocation, deployment, and scheduling
- Low latency networking (RDMA, Infiniband, etc.)
BACKGROUND

- The joy of joins
Working on Swissbox and interested in modern hardware, we started to look at the implementation of data operators on:

- Multicores
- Clusters
- Using processing pipelines
- Using RDMA
- Using specialized hardware
- etc.

Interesting things happen with multicores ...
Curious result

P. Roy, J. Teubner, G. Alonso
Efficient Frequent Item Counting in Multi-Core Hardware
KDD 2012
Joins are a complex and demanding operation
Lots of work on implementing all kinds of joins
In 2009
   Kim, Sedlar et al. paper (PVLDB 2009)
      Radix join on multicore
      Sort Merge join on multicore
      Claim fastest implementation to date
Key message: when SIMD wide enough, sort merge will be faster than radix join
In 2011

- Blanas, Li et al. (SIGMOD 2011)
  
  No partitioning join
  
  (vs. radix join version of Kim paper)

Claim: **Hardware is good enough, no need for careful tailoring to the underlying hardware**
In 2012

– Albutiu, Kemper et al. (PVLDB 2012)

Sort merge joins

(vs join version of Blanas)

Claim: Sort merge already better and **without using SIMD**
I will also discuss on the side some of the problems one encounters when doing performance analysis work in a research community that does not enforce any standards for reporting and comparing to related work.

- You will think serious researchers would
  - use and publish results based on the same metrics
  - compare results of papers on the same inputs
  - do sanity checks across published numbers
  - provide access to the code for verification
  - do not invent new metrics or play ration games
  - enforce these rules when reviewing
The basic hash join
Canonical Hash Join

1. Build phase

2. Probe phase

Complexity: $O(|R|+|S|)$, i.e., $O(N)$

Easy to parallelize
Need for Speed

Hardware-Conscious
Hash Joins
**Partitioned Hash Join** (Shatdal et al. 1994)

- **Idea:** Partition input into disjoint chunks of cache size

![Diagram of Partitioned Hash Join]

- **Problem:** $p$ can be too large!

> "Cache conscious algorithms for relational query processing", Shatdal et al, VLDB ’94

Performance in the multicore era
**Problem:** Hardware limits fan-out, i.e. \( T = \#\text{TLB-entries} \) (typically 64-512)

**Solution:** Do the partitioning in multiple passes!

---

- **TLB & Cache** efficiency compensates multiple read/write passes

---

Parallel Radix Join

Parallelizing the Partitioning: Pass - 1

Relation

Thread-1

Thread-2

Thread-3

Thread-N

Local Histograms

Global Histogram & Prefix Sum

Each thread scatters out its tuples based on the prefix sum

Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs, VLDB ‘09

Performance in the multicore era
Parallel Radix Join

Parallelizing the Partitioning: Pass - (2 .. i)

Each thread individually partitions sub-relations from pass-1

Performance in the multicore era
Trust the force

Hardware-Oblivious Hash Joins
Parallel Hash Join („no partitioning join“ of Blanas et al.)

1. Build Phase
   1. Acquire latch
   2. Store in bucket

2. Probe Phase
   Compare & match

Performance in the multicore era
Improving on Hardware-Oblivious Hash Joins
The main claim behind no partitioning is
- Hardware is good enough
  To hide cache misses
  To hide internal details of the architecture
- The code is multithreaded, uses latches, and pays no attention to NUMA
Cache Efficiency

Hash table in Blanas et al.

- latch array
- hashtable pointer array
- buckets (as linked list)

Hash table in our code

- hashtable as array of buckets

1-Byte 8-Bytes 48-Bytes

Expect 3 cache miss / tuple

L

head*
free  nxt  tuple-1  tuple-2
.
.
.
free  nxt  tuple-1  tuple-2

Expect 1 cache miss / tuple

L

tuple-1  tuple-2  nxt

Performance in the multicore era
SMT vs code efficiency

- SMT helps to hide cache miss latencies
- ... but for code with less redundancy, only negligible benefit
Comparison of Hardware-Oblivious Implementations

Workload: 16M \(\bowtie\) 256M, 16-byte tuples; Machine: Intel Xeon L5520, 2.26GHz, 4-cores, 8-threads

Performance in the multicore era
Improving on Hardware Conscious Hash joins
Configuration Parameters in Radix

- Radix join is fairly robust against a parameter misconfiguration
- Trade-off between partitioning and join costs

Workload: 977MiB × 977MiB, 8-byte tuples; Intel Nehalem: 2-passes, AMD Bulldozer: 1-pass

Performance in the multicore era
SIMD in Hash Join?

- Classical bucket chained hash table [Manegold et al., TKDE 2002]
- Relation re-ordering and histogram-based [Kim et al., VLDB 2009]

Contiguous tuples in \( R' \) can be compared with SIMD
Software prefetching for potential matches
Classic bucket chaining has an edge over other optimizations

SIMD improves hash joins

Workload: 977MiB $\bowtie$ 977MiB, 8-byte tuples; Intel Nehalem, 8 threads, 2-passes
Order of Build and Probe in Radix Join

- Order of build & probe is important for cache efficiency!

Wrong order: build all then probe  ❗ Hash tables will be evicted from cache!

Correct order: build and immediately probe → no more cache misses! ✔
Comparison of Hardware-Conscious Implementations

- Difference code efficiency and optimal configurations
- Order of building and probing hash tables

Workload: 16M \times 256M, 16-byte tuples; Machine: Intel Xeon L5520, 2.26GHz, 4-cores, 8-threads
And the winner is ...
Effect of Workloads – Case 1

- **Workload A**: 16M $\bowtie$ 256M, 16-byte tuples, i.e., 256MiB $\bowtie$ 4096MiB
  
  Hardware –oblivious vs. –conscious with our optimized code
  
  $\approx$ -30% may seem in favor of hardware-oblivious, but …

- 1. Effective on-chip threading
- 2. Efficient sync. primitives (ldstub)
- 3. Larger page size
Effect of Workloads – Case 2

- **Workload B**: Equal-sized tables, 977MiB $\bowtie$ 977MiB, 8-byte tuples

- Picture radically changes: Hardware-conscious is better by **3.5X** on Intel and **2.5X** on others

- With larger build table, overhead of not being hardware-conscious is clearly visible
Scalability

- **Workload B**: Equal-sized tables, 977MiB $\bowtie$ 977MiB, 8-byte tuples

![Graph showing throughput vs. number of threads]

- Intel Sandy Bridge 2.7GHz, 8 cores/16 threads
- Fastest reported join performance to date!

196 M/sec
14 cy/tpl
3.5X
55 M/sec
Hardware-Conscious or Not?

✗ Hardware-oblivious algorithms work well only under a narrow parameter window and on particular hardware

✔ Hardware-conscious algorithms
  – are significantly faster under a wider range of setup
  – can be tuned to the hardware
  – and are robust to wider set of parameters
Sort or Hash?
Sort or Hash?

- Input size is an important parameter!
- Optimizations on radix: NUMA-aware data placement, partitioning with software-managed buffers

Workload: 12.8GB \(\bowtie\) 12.8GB

- 375M/sec, 6.41 cy/tpl
- 299M/sec, 8.03 cy/tpl

Workload: 1GB \(\bowtie\) 1GB

- 619M/sec, 3.87 cy/tpl
- 305M/sec, 7.99 cy/tpl

Machine: Intel Sandy Bridge E4640, 2.4GHz, 32-cores, 64-threads

Performance in the multicore era
This is the end ...  
(or maybe just the beginning)
Conclusions

- We are fast, we are cool, we are clueless ...
  - Each machine is a different world (getting worse)
  - Many core changes the picture completely
  - Heterogeneous cores and interconnects

- We need
  - Benchmarks
  - Deeper analysis
  - More people working in the area
  - More open discussions on systems and implementations