Database Architecture Optimized for the new
Bottleneck: Memory Access

Peter Boncz
Data Distilleries B.V.
Amsterdam - The Netherlands
P.Boncz@ddil.nl

Stefan Manegold
CWI
Amsterdam - The Netherlands
S.Manegold,M.Kersten@cwilib.nl

1 Introduction

Custom hardware -- from workstations to PCs -- has been experiencing tremendous improvements in the past decades. Unfortunately, this growth has not been equally distributed over all aspects of hardware performance and capacity. Figure 1 shows that the speed of commercial microprocessors has been increasing roughly 75% every year, while the speed of commodity DRAM has improved by little more than 50% over the past decade [Mo98]. Part of the reason for this is the direct tradeoff between capacity and speed in DRAM chips, and the highest priority has been for increasing capacity. The result is that from the perspective of the processor, memory has been getting slower at a dramatic rate. This affects all computer systems, making it increasingly difficult to achieve high processor efficiencies.

Three aspects of memory performance are of interest: bandwidth, latency, and address translation. The only way to realize effective memory latency for applications is to provide effective access to all aspects of memory performance. In order to understand this problem, we require an architectural model that incorporates memory access cost. Experiments that validate this model were performed on the Meret database system. We obtained exact statistics on events like TLB misses, L1 and L2 cache misses, by using hardware performance counters found in modern CPUs. Using our cost model, we show how the carefully tuned memory access pattern of our radix algorithms make them perform well, which is confirmed by experimental results.

*This work was carried out when the author was at the University of Amsterdam, supported by FOM grant 060-SB-80.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.


Figure 1: Bandrate trends in DRAM and CPU speed
Agenda

Hardware background
In-memory partitioned hash join
Radix join
Experimental results
Radix join vs. non-partition hash join
Column-store and encoding
Agenda

**Hardware background**

- In-memory partitioned hash join
- Radix join
- Experimental results
- Radix join vs. non-partition hash join
- Column-store and encoding
Memory Wall

The growth of memory speed is slower than the growth of CPU speed
- Latency
- Bandwidth

Figure 1: Hardware trends in DRAM and CPU speed
Memory/Cache Hierarchy

Figure 2: Hierarchical Memory System
Memory/Cache Hierarchy

Optimizing join in DRAM/Disk system
- GRACE hash join

Optimizing join in SRAM/DRAM system?

Higher bandwidth
Lower access latency
Smaller capacity

Figure 2: Hierarchical Memory System
Optimizing Join in Main-Memory DBMS

Intuitive solution: Partition tables into shards that fit in SRAM cache
– Like GRACE hash join

Figure 2: Hierarchical Memory System
Recap: GRACE Hash Join

Phase 1: Partition both R and S into pairs of k shards
   - Each shard of R fits in CPU cache

Phase 2: Separately join each pairs of partitions
In some modern in-memory DBMSs, the entire database can fit in memory. In such a system, can similar optimizations be applied to on-chip SRAM caches vs. DRAM? What are the key challenges compared to a DRAM vs. Disk setting?
In some modern in-memory DBMSs, the entire database can fit in memory. In such a system, can similar optimizations be applied to on-chip SRAM caches vs. DRAM? What are the key challenges compared to a DRAM vs. Disk setting?

– Software does not have full control of CPU cache contents
– Disk access granularity is a block; DRAM access granularity is a cacheline
– CPU cache has very limited capacity 4KB 64B
Optimizing Join in Main-Memory DBMS

**Intuitive solution:** Partition tables into shards that fit in SRAM cache
- Like GRACE hash join

**Challenges:**
- TLB becomes a performance bottleneck if too many partitions exist
- Determine the memory layout of data partitions (e.g., fragmentation)

Figure 2: Hierarchical Memory System
Agenda

Hardware background

**In-memory partitioned hash join**

Radix join

Experimental results

Radix join vs. non-partition hash join

Column-store and encoding
Translation Lookaside Buffer (TLB)

A cache of OS page table to accelerate virtual address to physical address translation

- TLB hit has no cost
- TLB miss requires an expensive page table walk

TLB has a small number of entries

If the number of partitions is greater than the number of TLB entries, the system experience **TLB thrashing**, i.e., many accesses lead to TLB misses.
Thrashing

**TLB thrashing**: Number of accessed pages (i.e., number of partitions) is greater than the number of TLB entries in hardware

**Cache thrashing**: Number of accessed cachelines (i.e., number of partitions) is greater than the cache capacity

**Page thrashing** (in last lecture): Number of accessed pages (i.e., number of partitions) is greater than the memory capacity
Intuitive solution: Partition tables into shards that fit in SRAM cache

Challenges:
- TLB becomes a performance bottleneck if too many partitions exist
- Determine the memory layout of data partitions (e.g., fragmentation)

Do not have too many partitions per round of partitioning. Limiting factor includes cache size and TLB size.

Figure 2: Hierarchical Memory System
How to track location and size for different partitions?

- Frequent memory allocation (e.g., malloc) is expensive
- Loss of memory capacity due to fragmentation
- Problem becomes worse if multiple passes of partitioning is needed
Agenda

Hardware background
In-memory partitioned hash join

**Radix join**
Experimental results
Radix join vs. non-partition hash join
Column-store and encoding
Radix Partitioning

Cluster on the lower B bits of the integer hash-value of the partition key

- For pass $p$, use $B_p$ bits for partitioning
- Start with left most bits

Cluster on the lower B bits of the integer hash-value of the partition key

- For pass $p$, use $B_p$ bits for partitioning
- Start with left most bits

$2^B$ clusters.
Radix Partitioning

Cluster on the lower B bits of the integer hash-value of the partition key
- For pass $p$, use $B_p$ bits for partitioning
- Start with left most bits

The output array of Radix partitioning has identical structure as the input array
- No complex memory allocation
- No fragmentation
Radix Partitioning

Cluster on the **lower B bits of the integer hash-value** of the partition key
- For pass $p$, use $B_p$ bits for partitioning
- Start with left most bits

The output array of Radix partitioning has identical structure as the input array
- No complex memory allocation
- No fragmentation

Q: How to know where to write in the output array? (e.g., 47 in the example)
- Need to scan the array twice: first time collect size per partition
Radix Partitioning

Q: How to know where to write in the output array? (e.g., 47 in the example)

- Need to scan the array twice; first time collect size per partition

First scan:
- “00”: 5 records
- “01”: 2 records
- “10”: 3 records
- “11”: 2 records
Radix Partitioning

Q: How to know where to write in the output array? (e.g., 47 in the example)
   – Need to scan the array twice; first time collect size per partition

First scan
- “00”: 5 records
- “01”: 2 records
- “10”: 3 records
- “11”: 2 records

Write location in output buffer
- “00”: entry 0
- “01”: entry 5
- “10”: entry 5 + 2
- “11”: entry 5 + 2 + 3

Prefix Sum
Radix Partitioning

First scan
- “00”: 5 records
- “01”: 2 records
- “10”: 3 records
- “11”: 2 records

Write location in output buffer
- “00”: entry 0
- “01”: entry 5
- “10”: entry 5 + 2
- “11”: entry 5 + 2 + 3

Prefix Sum

Second scan: write to corresponding location in the output buffer

Q: How to know where to write in the output array? (e.g., 47 in the example)
- Need to scan the array twice; first time collect size per partition
Radix Partitioning

Fully clustering $B$ bits may require multiple passes

Number of partitions per pass is bounded by TLB and cache size
Join

Similar to GRACE hash join, join the corresponding partitions from the two relations.

Can use either **hash join** or **nested-loop join**

**Discussion Question:**
Can we use sort merge join for the two relations?
Agenda

Hardware background
In-memory partitioned hash join
Radix join

**Experimental results**
Radix join vs. non-partition hash join
Column-store and encoding
Evaluation: Radix Clustering

The machine’s TLB has 64 entries
Evaluation: Join Performance

Nested-loop join prefers small partitions

Hash-join achieves similar performance for a range of partition sizes
Evaluation: Overall

Sort-merge < Simple hash < phash L2 < phash TLB and the rest
Agenda

Hardware background
In-memory partitioned hash join
Radix join
Experimental results
Radix join vs. non-partition hash join
Column-store and encoding
A Different View Point

12 years later
Multicore processors

Two design considerations

− minimizing the number of processor cache misses => Radix Join
− minimizing processor synchronization costs => No partition hash join
Evaluation on Multicore

Uniform dataset

Highly skewed dataset

Important to minimize synchronization overhead in multicore processors
Agenda

Hardware background
In-memory partitioned hash join
Radix join
Experimental results
Radix join vs. non-partition hash join
**Column-store and encoding**
Other Topics

Column-store for analytical databases

Other Topics

Column-store for analytical databases

Dictionary encoding
- Many other encoding/compression schemes exist. E.g., bit-packing, delta encoding, RLE, etc.
Radix Join – Comments and Q/A

Radix join ensures tuples with same join key belong to same cluster?
Radix join assumes attributes stored as compact integer array?
Disadvantage of radix join?
Why having a shared hash table efficient for skewed data?
Common approach to use analytical model?
How to pick best parameters? (configurations vary across machines)
CPU speed improvement is also slowing down now.
Can radix join make use of modern hierarchical memory systems?
Core idea of radix-join portable to other operators?
Group Discussion

We want to join three tables, S \( \Join \) R \( \Join \) T. Assume S is large but R and T are relatively small (but larger than CPU cache). Assume the two joins are on different join keys. Would you use non-partitioned hash join or radix join for this query? Please justify your choice.
Before Next Lecture

Submit review for