How Architectural Knowledge Influences Algorithm Design for Databases

Kenneth A. Ross
Computer Science
Columbia University

DBIR Day
October 14, 2008
Commercial Database Systems

“Today, most DBMS installations are ‘compute bound’, at least after sufficient disk drives and indexes have been added. However, the bottlenecks are actually not CPU cycles for instruction execution but cache faults and pipeline stalls.”

Outline

• What are the performance-critical issues?
  – Hazards: Cache misses, branch mispredictions, ILP, etc.
  – Opportunities: SIMD, Graphics cards, Multi-core chips, etc.

• Examples of how to design algorithms that are architecture-aware:
  – Hash Tables
  – Multicore Aggregation (joint work with J. Cieslewicz)
  – (more in Proceedings of the IEEE survey paper)

• Future outlook
h = hash(key) % TableSize;

m = &table[h]; // m points to a hash bucket

do {
    for(i=0 ; i < m->keys_in_bucket; i++)
    {
        if (m->key[i] == key) return m->payload[i];
    }
    m = m->next_bucket;
} while (m != NULL);

return 0;
Performance Hazards for Hash Probes

• Cache miss
  – Hundreds of cycles latency on modern machines

• Conditional branch
  – Branch mispredictions
  – Barrier to code reordering
    • Reordering by compiler, or at run-time in an out-of-order processor
    • Reordering necessary to get good Instruction-Level Parallelism and avoid dependency stalls by overlapping latencies
Code Reordering Example

A = B + C
D = E + F

Load r1, B
Load r2, E
Add r1, C
Add r2, F
Store A, r1
Store D, r2

If there’s a cache miss on B, can overlap latency with load of E: two misses for the price of one

Conditional branch creates a control dependency prevents overlapping of work

A = B + C
if (x) goto …
D = E + F

Load r1, B
Add r1, C
Store A, r1
...
Load r2, E
Add r2, F
Store D, r2

If there’s a cache miss on B, can overlap latency with load of E: two misses for the price of one
Prior Work: Bucketized Cuckoo Hashing [Erlingsson et al, 2006]

Insert (R)
• Use H hash functions on R.key to give H candidate slots. (Slot capacity = B.)
• If some of the H slots are not full, put R in one of them. (Least full is a good choice.)
• Otherwise, evict a prior occupant P of one of the slots and replace it with R; Reinsert (P).

Reinsert (R)
• Like insert, but avoid previous slot.

Probe (R)
• Look in H slots for R.key.
Reinsertion

- A reinsertion may eventually **fail**.
  - Thresholding behavior: failure happens consistently around a given load threshold (random graph property)

![Graph showing reinsertion failure load vs. Log10(N)]

### Quantile

- **H=2, B=4**: 97.5% full before failure
- **H=3, B=4**: 99.8%
- **H=4, B=4**: 99.97%
- **H=2, B=8**: 99.7%
Bucketized Cuckoo Hash Summary

• **Very space-efficient**
  – 97% and up
  – At best 75% using conventional hashing methods to avoid degenerate probe behavior

• **But, what about all that extra work for probes?**
  – $H$ hash functions, not one
  – $H$ slots means $H$ cache misses, not one
  – $HB$ key comparisons rather than $B$
Optimizing the probes

• H hash functions, not one
  – Use SIMD to compute multiple hash values at once (needs the right kind of hash function)

• H slots means H cache misses, not one
  – The slot accesses are independent, so (if we’re careful) the miss latencies can be overlapped

• HB key comparisons rather than B
  – Use SIMD to compare multiple keys at once

• Branch-free code
  – Control dependencies become data dependencies
  – Better ILP, overlapping probes
Bucket 1, 4 keys, 4 payloads

Bucket 2, 4 keys, 4 payloads

Hash multipliers (little endian)

Table size

Table slots

Hash values

SIMD cmp
- eq

Search Key K

SIMD COPY

K K K K K

K K K K

K K K K

P4 P3 P2 P1

K4 K3 K2 K1

SIMD cmp-eq

mask

SIMD AND

SIMD OR

SIMD OR-ACROSS

P7

P1 P2 P3 P4

K1 K2 K3 K4

SIMD MULT

M1 M2

H1 H2

S1 S2

T T

SIMD MULT

S2 S1

Table slots

Hash values

Table size

SIMD MULT

S2 S1

H2 H1

M1 M2

Hash multipliers (little endian)
SIMD probe properties

- 9 SIMD operations altogether for H=2
- B=4 for only slightly more work than B=1
  - (B=1 can avoid the SIMD-COPY and OR-ACROSS)
- No branches!
- Most work for each bucket is independent of other buckets
  - Flexibility to reorder instructions
Multiplicative Hashing

• Classical hashing technique [Knuth]
  – Choose random odd 32-bit multiplier m
  – Compute the low-order 32 bits of m*K
• Fast on modern CPUs [Thorup 2000]
• Universal
  – i.e., with high probability, adversary can’t create degenerate dataset if m is chosen secretly
• Can be SIMDized
  – Compute 2 or 4 hash functions at once (using 64-bit or 32-bit arithmetic) in 128-bit registers
• Arbitrary table size using multiplication, not division
  – Don’t force size to be a power of 2
  – Rebuild is efficient: scan through old & new tables
A Fair Comparison?

• Can’t we also use SIMD and eliminate branches in conventional hashing?
  – Yes, but performance is worse
• E.g., for chaining
  – To eliminate branches, need a bound on length of chain
  – Even with a bound of 2, do more work in the common case just to handle rare overflow cases
  – Second cache miss is dependent on first, so no overlap
Experimental Framework

• Probing the Table
  – What is the probe cost, and what does the processor spend it’s time doing during a probe?
  – How does a splash table compare with conventional hash tables?
  – Are cache misses really overlapped?

– Insertion cost.
– How does probe time depend on B and H?
– Alternative Key/payload sizes
– Are there fundamental differences between a Pentium 4 and a Cell SPE that influence the results?

Hardware performance counters. Intel icc compiler.

8 SPEs per Cell Chip. Each SPE has 256KB of 6-cycle local memory. 3-5 GHz.

Spusim simulator (close to cycle accurate). IBM xlc compiler. Also, run on Cell Blade.
Conventional Hash Probes: Pentium

Chaining

Quadratic Probing

Load 0.75
Splash Table Probes (H=2, B=4): Pentium

- Total time
- (overestimated) L2 miss
- (overestimated) L1 miss
- (overestimated) TLB miss

Cycles/probe vs. Total Splash Table Size

- 2x faster
- 4x faster
- Ignore
Splash vs. Conventional Hash on Cell
(H=2, B=4)

Cycles on Cell: Splash vs. Hash

21 cycles. Validated on Blade.

Tables fit in SPE local memory (256KB)
Cache Latency Overlap

Pentium 4 (1.8GHz, 1GB RAM, 256K L2, latency 154ns)
Pentium 4 (3.0GHz, 2GB RAM, 1ML2, latency 112ns)
Cell PPU (2.1GHz, 512M RAM, 384K L2, latency 274ns)
Opteron (2.0GHz, 4GB RAM, 1ML2, latency 66ns)
Related Work

• Zukowski et al 2006
  – Similar motivation using a simpler variant of cuckoo hashing with <50% space utilization
  – Branches eliminated
  – No SIMD
  – Simpler (non-universal) hash functions
  – Scaling only to cache-sized tables
Splash Table Summary

• Space efficient
  – 97 – 99.9% utilization;
  – 100% utilization possible for small tables

• Time efficient relative to conventional hashing; for 32-bit keys/payloads:
  – Factor of 2-4 faster on Pentium 4
  – Factor of 10 faster on Cell SPE

• For more info
  – ICDE 2007 proceedings; IBM Research Report
Multicore Aggregation: Motivation

- Aggregation is a well understood database operation
- Seemingly easy to implement
- Chip multiprocessors introduce new challenges and opportunities
- Picking the wrong aggregation technique can result in a performance penalty of more than an order of magnitude.
Outline

• Chip multiprocessors
• Aggregation Strategies
• Modeling Performance
• Adaptive Aggregation
• Experimental Results
• Discussion and Future Work
The Multicore Future

- ILP is tapped out
- Heat dissipation and power consumption problems
- Thread Level Parallelism (TLP) is the future of performance gains

Thread level parallelism helps mitigate the memory bottleneck found in uniprocessor database performance.
Sun UltraSPARC T1

- 1 GHz
- 8 cores
- 4 threads / core
- 8 KB Data cache / core
- 16 KB Instr. cache / core
- 3 MB Shared L2
- “Lean camp”
Aggregation Overview

• Group tuples by zero or more attributes
• Compute an aggregate for each group
  – SQL standards: COUNT, SUM, AVERAGE, MIN, MAX
• Hash-based implementation
  – Conventional hashing, not Cuckoo-based hashing. (T1 has no hardware SIMD, no penalty for branch mispredictions, and does not process instructions out-of-order within a thread.)
Hash Aggregation

Key = 5
Value = 2
Hash Aggregation

Key = 5
Value = 6
Hash Aggregation

Key = 9
Value = 4

Hash

5 | 8

9 | 4
Aggregation Implementation

• All threads share input stream
  – Read contiguous chunks

• All threads work on the same operation
  – Intra-operator sharing and conflicts are easier to reason about than inter-operator
  – Instructions shared by threads (Instruction cache misses are expensive)
Option 1: Independent Tables

- Each thread has its own hash table
- No coordination between threads
- No sharing
- Capacity and conflict cache misses
- Huge memory requirement
Option 2: Global Tables, Mutex

- Threads share 1 table
- Shared table means more unique aggregate values fit in the cache
- Hash buckets must be locked to prevent race conditions
- Contention for common keys
Option 2: Global Tables, Atomic

- Same as mutex, but...
- No locking, use atomic operations for updates
- Provided by many microarchitectures
- More efficient than locking, longer latency than comparable non-atomic operation
- All standard aggregates implementable using atomic compare-and-swap
Option 3: Hybrid

- Independent tables fit in L2 cache
- Fixed size buckets
- “Spill” to global table
What influences performance?

• Runs
  – Consecutive tuples with same key can be aggregated directly

• Locality
  – If keys repeat with temporal locality, bucket will be in the cache

• Contention
  – If keys repeat too often in multiple threads, contention may occur for shared hash buckets
Aggregation Performance

Throughput

Millions of Records/Second

Runs

Contention

L2 Cache Size (locality)

Log2(|Group By|)
Modeling Performance

• Start with Hybrid
• Sample the input stream to gather statistics
• Each thread gathers independent statistics
  – No coordination overhead
  – Each thread proceeds as fast as possible
  – Local decision may not be globally optimal
    E.g., “If every thread saw input like mine, there would be contention.”
Why can’t the optimizer choose?

- Statistics might be wrong.
- Static choice cannot adapt to change in distribution.
- Multiple operators to choose from versus one that works well; reduces plan space.
- Aggregation occurs late in plans, other operators may have introduced skew.
Sampling for Runs

• Count the number of runs seen during the sampling window, find average run length.
• On uniform input, run optimization is beneficial up to $|\text{Group By}| = 8$
• Expected run length $= 1 + (1/8)^2 + (1/8)^3 \ldots = 8/7$
• Use run optimization if average run length exceeds $8/7$
Sampling for Locality

• Count “hits” in the local table
  – A “hit” is when a key is found (no insertion)
  – Tables sized to fit in L2 (likely to be a cache hit)
• Avoid compulsory misses with “warm-up”
• Locality if miss rate is less than 50%
  – Derived by the relative cost of processing a tuple in the local table compared to the cost of using a global table
Miss rate

The graph shows the miss rate as a function of the logarithmic scale of group-by size, for different access patterns and workloads. The access patterns include Uniform, Sorted, Heavy hitter, Sequential, Self-similar, Zipf, and Moving cluster. Each pattern is represented by a different line color or marker style.
Contention

• Often subsumed by locality, except...
• Distributions with “heavy hitters” have contention without locality
• Global table must be avoided if there is contention because overhead dominates execution time
The time for a contentious tuple is a linearly decreasing function of $1/p$, where $p$ is the fraction of the input that’s contentious. (See paper for mathematical derivation)

- Adding more threads can increase baseline contention and, therefore, processing time.
Contestation Test

• If a value occurs in more than 1/8\textsuperscript{th} of the input, it contributes to contention.

• Consider all values that occur in more than 1/8\textsuperscript{th} of the input, calculate the total potential overhead.

• If overhead exceeds a threshold, flag the input as contentious.
MIN, MAX, Duplicate Elimination

• Contention Free
• Why? Answer: Updates are rare
  – E.g., given a uniform input, after 99 inputs the running minimum is in the first percentile. The chance that the 100\textsuperscript{th} value will update the minimum is 1%.
• Adversarial distributions exist, but can be handled with randomization.
Adaptive Aggregation

MIN, MAX, DUPE?

YES

Hybrid Aggregation

NO

Locality or Contention?

YES

If runs are present, add run optimization.

Atomic Aggregation

NO
Experiments

- \(2^{24} \approx 16\) Million Input Tuples
- 7 Input Distributions, 3 Queries

**Q1:** Select G, count(*), sum(V), sum(V\*V)  
From R Group By G

**Q2:** SELECT G, max(V), min(V), max(V)  
FROM R Group By G

**Q3:** Select Distinct G From R
Q1: Uniform Input

Throughput

Millions of Records/Second

Runs

Contevention

L2 Cache Size (locality)

Log2(|Group By|)
Q1: Uniform Time Breakdown
Q2: Uniform Input

Throughput

Millions of Records/Second

Log2(|Group By|)
Q3: Uniform Input

Throughput

Millions of Records/Second

Log2(|Group By|)
Q1: Self-similar Input

Throughput

Millions of Records/Second

Log2(|Group By|)

No Contention

Contention

Mutex
Atomic
Independent
Hybrid
Adaptive
Q1: Sorted Input

![Graph showing throughput vs. Log2(|Group By|)]

- **Throughput**
  - X-axis: Log2(|Group By|)
  - Y-axis: Millions of Records/Second

Lines represent different methods:
- Mutex
- Atomic
- Independent
- Hybrid
- Adaptive
Q1: Moving Cluster Input

Throughput

Millions of Records/Second

Log2(|Group By|)
Q1: Zipf Input

Throughput

 Millions of Records/Second

Log2(|Group By|)
Q1: Heavy Hitter (50%) Input

Throughput

Millions of Records/Second

Log2(|Group By|)
Scaling: $|\text{Group By}| = 4$

Contention increases as more threads are added.
Scaling: $|\text{Group By}| = 65336$
Resampling

• Resample the input distribution
• Frequency of resampling dictates:
  – Sampling overhead
  – Sensitivity to changing distributions
• Experimented with a mixed input distribution that contains segments of each input distribution.
• See paper for details
Discussion - Improving Locality

12 cycles per tuple
Discussion - TLP vs. ILP

• Zukowski et al. perform a similar duplicate elimination on 3GHz P4 (4 byte records):
  – 26 MTups/s for |Group By| = 1M
  – 81 MTups/s for |Group By| = 256

• Our numbers (8 byte records):
  – 54 MTups/s and 250MTups/s

• The T1 has an 8:3 cycle advantage.

• Comparable performance, less energy.
Adaptive Aggregation: Conclusion

• Investigated aggregation performance on a real chip multiprocessor.
• Identified locality and contention as key performance issues.
• Introduced an adaptive aggregation operator that uses lightweight sampling to choose the best aggregation strategy.
• See VLDB 2007 paper for more details
Lessons

• For future studies that measure performance:
  – Cannot ignore parallelism; must implement multithreaded algorithms
    • Good news: Databases often have embarrassing parallelism
  – Cannot ignore SIMD, cache misses, and the architectural effects of conditional branches
    • Alternative algorithms can outperform traditional ones
    • Simple metrics (e.g., counting operations) can be deceptive.
Future Directions

• Chip multiprocessors
  – Managing concurrency
  – Avoiding serialization
  – Avoiding cache interference between threads/cores
  – What kind of parallelism is most easily exploited?
  – Lean camp vs. Fat camp

• New commodity hardware
  – Flash memory; solid-state disks
  – FPGAs (e.g., Netezza)

• New metrics
  – Power
  – Performance per dollar
  – Manageability (total cost of ownership)