Towards hardware-software co-design for data analytics

The Wisconsin Quickstep Project

Jignesh M. Patel

Blog: http://bigfastdata.blogspot.com
Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs

Spyros Blanas, Yinan Li, Jignesh M. Patel
University of Wisconsin-Madison
{sblanas, yinan, jignesh}@cs.wisc.edu

1. INTRODUCTION

Large scale multi-core processors are imminent. Modern processors today already have four or more cores, and for the past few years Intel has been introducing two more cores per processor roughly every 15 months. At this rate, it is not hard to imagine running database management systems (DBMSs) on processors with hundreds of cores in the near future. In addition, memory prices are continuing to drop. Today LTE of memory costs as little as $25,000. Consequently, many databases now either fit entirely in main memory, or their working set is main memory resident. As a result, many DBMSs are becoming CPU bound.

In this evolving architectural landscape, DBMSs have the unique opportunity to leverage the inherent parallelism that is provided by the relational data model. Data is exposed by declarative query language to user applications and the DBMS is free to choose its execution strategy. Coupled with the trend towards impeding very large multi-cores, this implies that DBMSs must carefully rethink how they can exploit the parallelism that is provided by the modern multi-core processor, or DBMS performance will stall.

A natural question to ask then is whether there is anything new here. Beginning about three decades ago, at the inception of the field of parallel DBMSs, the database community thoroughly examined how a DBMS can use various forms of parallelism. These forms of parallelism include pure shared-nothing, shared-memory, and shared disk architectures [17]. If the modern multi-core architectures resemble any of these architectural template, then we can simply adopt the methods that have already been designed.

In fact, to a large extent this is the approach that DBMSs have taken towards dealing with multi-core machines. Many commercial DBMSs simply treat a multi-core processor as a symmetric multi-processor (SMP) machine, leveraging previous work that was done by the DBMS vendors' two-re search teams. In reaction to the increasing popularity of SMP machines decades ago. These methods leverage the task of a single operation, such as an equijoin, into disjoint parts and allow each processor (as an SMP box) to work on each part independently. At a high-level, these methods resemble variations of query processing techniques that were developed for parallel shared-nothing architectures [19], but adapted for SMP machines. In most commercial DBMSs, this approach is reflected across the entire design process, ranging from system internals (join processing, for example) to their pricing model, which is frequently done by scaling the SMP pricing model. On the other hand, open-source DBMSs have...
Results – uniform dataset

- **Intel Xeon**
  - **Radix**: 3.3x more cache misses
  - 70x more TLB misses on load than Radix
  - 24% faster than WiscJ

- **Sun T2**
  - **Radix**: Instruction path length is 58% of Radix
  - 47% more cache misses than Radix
  - 51% faster than Radix
Skew in partitioning-based hash join algorithms causes partition size skew → work imbalance

Non-partitioned (Wisconsin) hash join improves with higher skew!

© 2012, Jignesh M. Patel, University of Wisconsin
Hash Joins (2011): Summary

- Hash join algorithm started simple, and with each architectural turn, it adapted.

- We have come full circle: The simple hash join is now very competitive. And, in many cases more efficient than the more complex methods!
Data Processing Kernels

Fast & Scalable

"Relational-like" Data Processing Primitives

- \( \pi \) (project)
- \( \sigma \) (select)
- \( \gamma \) (aggregate)
- \( \tau \) (sort)
- \( \natural \) (join)
- \( \delta \) (bag->set)
- \( \triangleright \) (anti-join)
- \( \_ \) (minus)
- \( \cup \) (union)

Applications

Platforms

Deep Learning • MapReduce • K/V/Document Store • Operational • Graph
Disruptive hardware trends

Want

High Performance
Low Cost

Constraint

Power

Low-powered, lower latency, higher bandwidth, persistent stores

Multicores, multi-socket, heterogeneous cores

Magnetic Hard Disk Drives

CPU caches

NVRAM (e.g., SSDs)

© 2013, Jignesh M. Patel, University of Wisconsin
Quickstep

Goal

• Run data analytics @ hardware speeds

Short-term

• Run @ the speed of hardware today

Long-term

• Hardware-software co-design for data kernels

http://quickstep.cs.wisc.edu
## Scan: A Key Data Processing Kernel

### What?
- Scan a column of a table applying some predicate

### Why?
- A key primitive in database
- “The” critical kernel in main memory analytic systems

### How?
- Conserve memory bandwidth: BitWeaving the data
- Use every bit of data that is brought to the processor efficiently using **intra-cycle parallelism**
Focus on Column Scan (can be generalized)

Traditional Row Store

<table>
<thead>
<tr>
<th>shipdate</th>
<th>...</th>
<th>discount</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mar-12-2013</td>
<td></td>
<td>5%</td>
<td>5</td>
</tr>
<tr>
<td>Jan-08-2013</td>
<td></td>
<td>2%</td>
<td>4</td>
</tr>
<tr>
<td>Apr-29-2013</td>
<td></td>
<td>10%</td>
<td>3</td>
</tr>
<tr>
<td>May-14-2013</td>
<td></td>
<td>0%</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Feb-28-2013</td>
<td></td>
<td>5%</td>
<td>0</td>
</tr>
</tbody>
</table>

One big file

Column Store

<table>
<thead>
<tr>
<th>shipdate</th>
<th>discount</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mar-12-2013</td>
<td>5%</td>
<td>5</td>
</tr>
<tr>
<td>Jan-08-2013</td>
<td>2%</td>
<td>4</td>
</tr>
<tr>
<td>Apr-29-2013</td>
<td>10%</td>
<td>3</td>
</tr>
<tr>
<td>May-14-2013</td>
<td>0%</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Feb-28-2013</td>
<td>5%</td>
<td>0</td>
</tr>
</tbody>
</table>

File: 1

File: n-1

File: n

Order-preserving compression

Column Codes: 3 bits

5 4 3 6 2 7 1 0 ...
First batch of Processor Words

(batch size = code size in bits)
**Framework – Example**

```sql
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
AND l_quantity < Quantity
```
# Column Codes:

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>12</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>7</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Word 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Word 3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Word 4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **The first (most significant) bits of 8 consecutive codes**
- **The second bits of 8 consecutive codes**
- **The third bits of 8 consecutive codes**
- **The last (least significant) bits of 8 consecutive codes**
**BitWeaving/V - early pruning**

**Column Codes:**

<table>
<thead>
<tr>
<th>10</th>
<th>12</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>7</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Constant**

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
</table>

**Predicate**

\( a < 5 \)

**Result Bit Vector**

0 0 1 0 0 0 1 1

---

**Early Pruning:** terminate the predicate evaluation on a segment, when all results have been determined.
BitWeaving/V - Early Pruning Model

- Early pruning probability: 96% at bit position 4
- Early pruning probability: 98% at bit position 8

This cut-off mechanism allows for efficient evaluation of conjunction/disjunction predicates
BitWeaving/H - Example

Segment 1

Memory space

Code size (3 + 1 bits)

Predicate evaluation is done on the 4 codes in parallel

Word size (16 bits)

© 2014, Jignesh M. Patel, University of Wisconsin
BitWeaving/H: Less Than Predicate

Uses only 3 instructions! Without the delimiter, we would need ~12 instructions...

\[ X = (c_1 c_5 c_9 c_{13}) \]

\[ Y = (5555) \]

\[ (Y + (X \oplus M1)) \land M2 \]

\[ M1 = 0111 \ 0111 \ 0111 \ 0111 \]

\[ M2 = 1000 \ 1000 \ 1000 \ 1000 \]

Works for arbitrary code sizes & word sizes!
### BitWeaving/H - Example

<table>
<thead>
<tr>
<th>Segment 1</th>
<th>Code size (3 + 1 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Memory space</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment 1</td>
</tr>
<tr>
<td>Word 1</td>
</tr>
<tr>
<td>c1 c5 c9 c13</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Word 2</td>
</tr>
<tr>
<td>c2 c6 c10 c14</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Word 3</td>
</tr>
<tr>
<td>c3 c7 c11 c15</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Word 4</td>
</tr>
<tr>
<td>c4 c8 c12 c16</td>
</tr>
</tbody>
</table>

**Word size** (16 bits)

**Result bit vector computed efficiently with this layout!**

![Result bit vector](image)
## Evaluation

<table>
<thead>
<tr>
<th>SYSTEM</th>
<th>WORKLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Intel Xeon X5650</td>
<td>1. Synthetic</td>
</tr>
<tr>
<td>• 64 bits ALU</td>
<td>• SELECT COUNT(*)</td>
</tr>
<tr>
<td>• 128 bits SIMD</td>
<td>FROM R</td>
</tr>
<tr>
<td>• 12MB L3 Cache</td>
<td>WHERE R.a &lt; C</td>
</tr>
<tr>
<td>• 24GB memory</td>
<td>• 1 billion tuples</td>
</tr>
<tr>
<td>• Single threaded</td>
<td>• Uniform distribution</td>
</tr>
<tr>
<td>execution</td>
<td>• Selectivity: 10%</td>
</tr>
</tbody>
</table>

2. TPC-H @ SF=10
   • scan only with materialized join results
Evaluation: Micro-benchmark

![Graph showing cycles per code size vs size of code (bits)]

Cycles / code

Size of code (# bits)

Naive
**Evaluation: Micro-benchmark**

- **Cycles / code**
- **Size of code (# bits)**

**SIMD Paper:** T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner.
**SIMD-Scan:** Ultra fast in-memory table scan using on-chip vector processing units. PVLDB'09

2X: SIMD parallelism

© 2014, Jignesh M. Patel, University of Wisconsin
Evaluation: Micro-benchmark

3X-4X speedup over BL:
1) Use the extra (delimiter) bit
2) Easy to produce the result bit vector with the HBP layout
Evaluation: Micro-benchmark

Many more experiments in the paper

2X speedup: Early pruning

- Naive
- SIMD
- BL
- BitWeaving/H
- BitWeaving/V

Cycles / code vs. Size of code (# bits)
### Customer

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>gender</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Andy</td>
<td>M</td>
<td>100 Main st.</td>
</tr>
<tr>
<td>2</td>
<td>Kate</td>
<td>F</td>
<td>20 10th blvd.</td>
</tr>
<tr>
<td>3</td>
<td>Bob</td>
<td>M</td>
<td>300 5th ave.</td>
</tr>
</tbody>
</table>

### Product

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Milk</td>
</tr>
<tr>
<td>2</td>
<td>Coffee</td>
</tr>
<tr>
<td>3</td>
<td>Tea</td>
</tr>
</tbody>
</table>

### Buy

<table>
<thead>
<tr>
<th>cid</th>
<th>pid</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>S</td>
</tr>
</tbody>
</table>

- **WideTable**

© 2014, Jignesh M. Patel, University of Wisconsin
Now we can run analytical workloads (e.g. TPC-H) using simple BitWeaved scans
**WideTable Techniques**

**Quickstep/WT**

- **Query Transformer**
  - $R \times S \equiv \pi_{R_{eq}}(R \times S)$
  - $R \times \ldots \times S \equiv \sigma_{R.p \neq \text{NULL}} \ldots \sigma_{S.p \neq \text{NULL}}(R \times \ldots \times S)$
  - $\gamma(R) \times S \equiv \gamma(R \times S)$
  - $R \equiv \pi_{R_{eq}}(R \times S)$
  - $\pi_{a_1}(R) \equiv \pi_{a_1}(\ldots(\pi_{a_n}(R)))$, where $a_1 \subseteq \ldots \subseteq a_n$

- **Denormalizer**
  - Schema transformer
  - Flatten using $\Join$

- **Data**
- **Schema Graph**

- **Bit-Weaved Scans**

- **{WideTables}**

- **Results**

- “Non-transformable” query sent to the source system

© 2014, Jignesh M. Patel, University of Wisconsin
WideTable = (Region × Nation × Customer) × (Region × Nation × Product × Buy)

SMW = {WideTables}³
e.g. for TPC-H, SMW={lineItemWT, ordersWT, partsuppWT, customerWT}
## TPC-H Queries

<table>
<thead>
<tr>
<th>TPC-H Queries</th>
<th>Joins</th>
<th>Nested Queries</th>
<th>Non-FK joins</th>
<th>WideTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1, Q6</td>
<td></td>
<td></td>
<td></td>
<td>LineitemWT</td>
</tr>
<tr>
<td>Q3, Q5, Q7-Q10, Q12, Q14, Q19</td>
<td>×</td>
<td></td>
<td></td>
<td>LineitemWT</td>
</tr>
<tr>
<td>Q4, Q15, Q17, Q18, Q20</td>
<td>×</td>
<td>×</td>
<td></td>
<td>LineitemWT</td>
</tr>
<tr>
<td>Q21</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>---</td>
</tr>
<tr>
<td>Q2, Q11, Q16</td>
<td>×</td>
<td>×</td>
<td></td>
<td>PartsuppWT</td>
</tr>
<tr>
<td>Q13</td>
<td>×</td>
<td></td>
<td></td>
<td>OrdersWT</td>
</tr>
<tr>
<td>Q22</td>
<td>×</td>
<td>×</td>
<td></td>
<td>OrdersWT</td>
</tr>
</tbody>
</table>
Evaluation

SYSTEM

- Intel Xeon E5-2620 × 2
- 2.0 GHz
- 12 cores / 24 threads
- 15MB L3 Cache
- 32G, 1600MHz DDR3

BENCHMARK

- SF: 10 (~10GB)
- SMW =

<table>
<thead>
<tr>
<th>lineItemWT</th>
<th>5.4 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordersWT</td>
<td>0.7 GB</td>
</tr>
<tr>
<td>partsuppWT</td>
<td>0.2 GB</td>
</tr>
<tr>
<td>customerWT</td>
<td>0.05 GB</td>
</tr>
<tr>
<td>dictionaries</td>
<td>0.8 GB</td>
</tr>
<tr>
<td>filter columns</td>
<td>1.3 GB</td>
</tr>
<tr>
<td>TOTAL</td>
<td>8.5GB</td>
</tr>
</tbody>
</table>

© 2014, Jignesh M. Patel, University of Wisconsin
Speedup over MonetDB: Single Thread

WideTable over 10X faster than MonetDB for about half of the 21 queries
Speedup over MonetDB: 12 Threads

WideTable scales better

© 2014, Jignesh M. Patel, University of Wisconsin
Disruptive hardware trends

- **CPU caches**: Multicores, multi-socket, heterogeneous cores
- **DRAM**: Lower-powered, lower latency, higher bandwidth, persistent stores
- **NVRAM (e.g. SSDs)**
- **Magnetic Hard Disk Drives**

© 2013, Jignesh M. Patel, University of Wisconsin
Long Term: Raw computing and storage costs tend to zero!

The cost is in moving data and powering the circuits/devices
There are similar ways of using hardware creatively, e.g. IDISKs, ASICs, CGRA, FPGAs, or GPUs.

Basically, need hardware and software synergy!
Hardware Software Co-design: A Good Starting Point

Starting point: Two queries

Scan

• Sequential read kernel

Scattered Read/Write

• Index access kernel
Conclusions and Future Work

Transformative architectural changes at all levels (CPU, memory subsystem, I/O subsystem) is underway

Need to rethink data processing kernels
- Run @ current bare metal speed

Need to think of hardware software co-design