Columnar processing for SQL-on-Hadoop: The best is yet to come

Henry Robinson (Cloudera) | Zuo Wang (Intel)
“Columnar? Isn’t that, like, so three years ago?”
Columnar formats store all the same values of a column consecutively on disk.

CREATE TABLE tbl (a int, b string, c timestamp, d string) STORED AS PARQUET

Row-oriented storage

<table>
<thead>
<tr>
<th>a1</th>
<th>b1</th>
<th>c1</th>
<th>d1</th>
<th>a2</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c2</td>
<td>d2</td>
<td>a3</td>
<td>b3</td>
<td>c3</td>
<td>d3</td>
</tr>
<tr>
<td>a4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Column-oriented storage

<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>a4</th>
<th>a5</th>
<th>a6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>aN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>aN</td>
<td></td>
<td>b1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b2</td>
<td>b3</td>
<td>b4</td>
<td></td>
</tr>
<tr>
<td>b5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
There’s always another bottleneck.
Columns always are eventually transposed into rows
Does it make sense to keep data in columnar format beyond the initial scan?
CPUs are faster than memory, and good at doing the same thing many times in parallel.
Columnar formats help efficient memory usage.

Row orientation would mean four memory fetches, and maybe one or more cache misses.
Columnar formats help the query engine parallelise.

**Compute column A + column B**

SIMD Register 1

a1  a2  a3  a4

+  +  +  +  

SIMD Register 2

b1  b2  b3  b4

Compute A + B for four rows in one operation

SIMD Register 1

a1 + b1  a2 + b2  a3 + b3  a4 + b4

Then save results with a single store to memory.
Vectorization is widely applicable

```sql
SELECT tblA.proj, tblB.proj  
  FROM tblA  
  JOIN tblB  
    ON tblA.col = tblB.col  
WHERE tblA.expr > 10  
ORDER BY tblA.sortCol
```
Vectorization is widely applicable

```sql
SELECT tblA.proj, tblB.proj
FROM tblA
JOIN tblB
ON tblA.col = tblB.col
WHERE tblA.expr > 10
ORDER BY tblA.sortCol
```
Experimental results: vectorized scans in Impala
Optimize Impala with AVX2 — Experimental

October, 2015

Henry Robinson, Zuo Wang
Outline

- Intel’s AVX Instruction set
- AVX2 optimized Parquet scanner
- Benchmark results
Intel® Advanced Vector Extensions

- **AVX1**
  - Expand 128-bit SIMD registers to 256 bits
  - Three-operand, nondestructive operations
  - Memory alignment requirements for operands are relaxed.

- **AVX2 (AVX-256)**
  - 256-bit integer SIMD instructions
  - FMA instruction
  - Gather instruction
  - Permute instruction

- **AVX3 (AVX-512)**
  - Expand 256-bit SIMD registers to 512 bits
  - Conflict Detection Instructions
  - Exponential and Reciprocal Instructions
  - Prefetch Instructions
  - Byte and Word Instructions

AVX Registers getting wider, instruction set getting richer

Performance/core
Optimize Database Operations Using SIMD Instructions

- Partition
- Join
- Aggregate
- Sort
- Index
- Predicates evaluation
- Compression
- Scan
- …
Outline

- Intel’s AVX2 Instruction set
- AVX2 optimized Parquet scanner
- Benchmark results
Scan

Research papers on vectorized scan

Johnson et al, VLDB’08
Willhalm et al, PVLDB,09
Willhalm et al, ADMS’13
Li and Pater, SIGMOD’13
Ziqiang Feng and Eric Lo, ICDE’15
### Column Store

- **shipdate**
  - Mar-12-2013
  - Jan-08-2013
  - Apr-29-2013
  - May-14-2013
  - ...
  - Feb-28-2013

- **discount**
  - 5%
  - 2%
  - 10%
  - 0%
  - ...
  - 5%

- **quantity**
  - 5
  - 4
  - 3
  - 6
  - ...
  - 0

### Ordered dictionary

- **Order preserved compression**
  - **Jan-01-2013:** 0%
  - **Jan-02-2013:** 1%
  - **Jan-03-2013:** 2%
  - **Jan-04-2013:** 3%
  - **Jan-05-2013:** 4%
  - **Jan-06-2013:** 5%
  - **Jan-07-2013:** 6%
  - **Jan-08-2013:** 7%
  - **Jan-09-2013:** 8%
  - **Jan-10-2013:** 9%
  - **Jan-11-2013:** 10%

---

**Column Codes**

- **128 bits**
  - **shipdate**
  - **32 bits**
  - **discount**
  - **32 bits**
  - **quantity**

- **4 bits**
  - 5
  - 4
  - 3
  - 6
  - ...
  - 0

- **3 bits**
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8

- **9 bits**
  - 70
  - 7
  - 118
  - 133
  - ...
  - 58
9 bits

4 bits

3 bits

64 bits
Less Than Predicate

<table>
<thead>
<tr>
<th>Column Quantity Codes</th>
<th>Bit vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4 3 6 ... 0</td>
<td>0 1 1 0 ... 1</td>
</tr>
</tbody>
</table>

MSB

<table>
<thead>
<tr>
<th>3 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 ... ?</td>
<td>1 1 1 1 ... 1</td>
</tr>
<tr>
<td>0 0 1 1 ... ?</td>
<td>0 0 0 0 ... 0</td>
</tr>
<tr>
<td>1 0 1 0 ... ?</td>
<td>1 1 1 1 ... 1</td>
</tr>
</tbody>
</table>

LSB

<table>
<thead>
<tr>
<th>3 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>? ? ? ? ... 0</td>
<td>1 1 1 1 ... 1</td>
</tr>
<tr>
<td>? ? ? ? ... 0</td>
<td>0 0 0 0 ... 0</td>
</tr>
<tr>
<td>? ? ? ? ... 0</td>
<td>1 1 1 1 ... 1</td>
</tr>
</tbody>
</table>

64 bits

3 bits

? ? ? ? ... 0

64 bits

3 bits

? ? ? ? ... 0

64 bits

3 bits

? ? ? ? ... 0

64 bits

64 bits
Example

SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1
AND l_discount BETWEEN Discount – 0.01 AND Discount + 0.01
AND l_quantity < Quantity
Outline

- Intel’s AVX2 Instruction set
- AVX2 optimized Parquet scanner
- Benchmark results
## Environment

### Hardware environment

<table>
<thead>
<tr>
<th>Node</th>
<th>4 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Haswell Xeon(R) CPU E5-2695 v3 @ 2.30GHz (2 CPUs, 14 cores, 56 threads)</td>
</tr>
<tr>
<td>Memory</td>
<td>384GB</td>
</tr>
<tr>
<td>Disk</td>
<td>8 SSDs(800GB)</td>
</tr>
<tr>
<td>SIMD</td>
<td>AVX2 256 bit YMM register</td>
</tr>
<tr>
<td>Network</td>
<td>10 Gigabit Ethernet</td>
</tr>
</tbody>
</table>

### Software Environment

<table>
<thead>
<tr>
<th>OS</th>
<th>Ubuntu 12.04.4 LTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>3.11.0-15-generic x86_64</td>
</tr>
<tr>
<td>Hadoop/HDFS</td>
<td>CDH 5.4.1</td>
</tr>
<tr>
<td>Impala</td>
<td>2.1.0-cdh4-INTERNAL RELEASE</td>
</tr>
<tr>
<td>JDK</td>
<td>1.6.0_31(64bit)</td>
</tr>
</tbody>
</table>
Compression

![Graph showing rate (iters/ms) vs bits for naive, avx2, and avx2/naive methods. The graph has a logarithmic scale for both x and y axes. The y-axis ranges from 0 to 4, and the x-axis ranges from 0 to 200. The graph shows a decrease in rate with increasing bits for all methods. The avx2/naive method has the highest rate at 2.792 iters/ms when bits = 1. The naive method has the lowest rate. The avx2 method is in between the two.](image-url)
Decompresion

Rate (iters/ms) vs. bits

- naïve
- avx2
- avx2/naïve
Selectivity

<table>
<thead>
<tr>
<th>Query Time(s)</th>
<th>naïve</th>
<th>avx2</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>14.49</td>
<td>14.67</td>
<td>127.65%</td>
</tr>
<tr>
<td>10%</td>
<td>15.93</td>
<td>14.67</td>
<td>101.10%</td>
</tr>
<tr>
<td>20%</td>
<td>15.70</td>
<td>69.36%</td>
<td>90.32%</td>
</tr>
<tr>
<td>30%</td>
<td>16.06</td>
<td>49.73%</td>
<td>9.27</td>
</tr>
<tr>
<td>50%</td>
<td>16.55</td>
<td>29.77%</td>
<td>10.73</td>
</tr>
<tr>
<td>80%</td>
<td>16.07</td>
<td>20.13%</td>
<td>12.76</td>
</tr>
<tr>
<td>100%</td>
<td>13.38</td>
<td></td>
<td>13.38</td>
</tr>
</tbody>
</table>
TPC-H Results

Queries Time(s)

- TPCH-Q6
- TPCH-Q13
- TPCH-Q14
- TPCH-Q15
- TPCH-Q20

naive

avx2
TPC-DS Results

<table>
<thead>
<tr>
<th>Query</th>
<th>naive</th>
<th>avx2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpcd-q3</td>
<td>2.30</td>
<td>2.19</td>
</tr>
<tr>
<td>tpcd-q7</td>
<td>4.88</td>
<td>4.92</td>
</tr>
<tr>
<td>tpcd-q27</td>
<td>2.90</td>
<td>2.82</td>
</tr>
<tr>
<td>tpcd-q96</td>
<td>4.45</td>
<td>4.66</td>
</tr>
</tbody>
</table>
TPC-DS Results Explanation

SELECT ... FROM date_dim dt,store_sales,item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
AND store_sales.ss_item_sk = item.i_item_sk
AND item.i_manufact_id = 436
AND dt.d_moy = 12
...

Predicates are all applied to dimension tables. Dimension tables are quite small. The time spend on scanning dimension tables is negligible compared to the whole query time.

<table>
<thead>
<tr>
<th>Table</th>
<th>Data_dim</th>
<th>Store_sales</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Rows</td>
<td>73049</td>
<td>864001869</td>
<td>264000</td>
</tr>
<tr>
<td>Size</td>
<td>2.17MB</td>
<td>38.63GB</td>
<td>24.93MB</td>
</tr>
</tbody>
</table>

SF=100
Thank you!
zuo.a.wang@intel.com
Interoperability, or “it takes a village to run a query”
Lots of systems participate in data processing on Hadoop.
Impala and Ibis swap data on the same machine.
Share columnar in-memory representations by adopting a standard format.
Impala and Ibis swap data on the same machine

- Impala executes UDF, shares input data
- Ibis returns results of expression, by sharing pointer to new result packet
Thank you!
Image Attributions

• Bottle by Flickr user ‘FeistyTortilla’
  https://flic.kr/p/pptq6v

• CPU by Mark Freeth
  https://flic.kr/p/pHoMms

• Sharing by Môsieur J.
  https://flic.kr/p/4K6zsd

• All images used under the Creative Commons Attribution-NonCommercial 2.0 license.
  https://creativecommons.org/licenses/by-nc/2.0/legalcode