Vectorized Query Processing using Apache Arrow

Strata Data Conference, San Jose
March 7, 2018
Siddharth Teotia
Who?

Siddharth Teotia
@siddcoder
loonytek
Quora

• Software Engineer @ Dremio
• Committer - Apache Arrow
• Formerly at Oracle (Database Engine team)
Apache Arrow Project

• Top-level Apache Software Foundation project
  – Announced Feb 17, 2016

• Focused on Columnar In-Memory Analytics
  1. 10-100x speedup on many workloads
  2. Designed to work with any programming language
  3. Flexible data model that handles both flat and nested types

• Developers from 13+ major open source projects involved.
Arrow goals

• Columnar in-memory representation optimized for efficient use of processor cache through data locality.
• Designed to take advantage of modern CPU characteristics by implementing algorithms that leverage hardware acceleration.
• Interoperability for high speed exchange between data systems.
• Embeddable in execution engines, storage layers, etc.
• Well-documented and cross language compatible.
High Performance Interface for Data Exchange

- Each system has its own internal memory format
- 70-80% CPU wasted on serialization and deserialization
- Functionality duplication and unnecessary conversions

- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (e.g., Parquet-to-Arrow reader)
Focus on CPU Efficiency

- Maximize CPU throughput
  - SIMD
  - Cache Locality
- Vectorized operations.
- Minimal Structure Overhead
- Constant value access
  - With minimal structure overhead
- Use efficient lightweight compression schemes on a per column basis.

<table>
<thead>
<tr>
<th>session_id</th>
<th>timestamp</th>
<th>source_ip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1331246351</td>
<td>3/8/2012 2:38PM</td>
<td>65.87.165.114</td>
</tr>
<tr>
<td>1331244570</td>
<td>3/8/2012 2:09PM</td>
<td>71.10.106.181</td>
</tr>
<tr>
<td>1331261196</td>
<td>3/8/2012 6:46PM</td>
<td>76.102.156.138</td>
</tr>
</tbody>
</table>

**Traditional Memory Buffer (row format)**

<table>
<thead>
<tr>
<th>session_id</th>
<th>timestamp</th>
<th>source_ip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1331246351</td>
<td>3/8/2012 2:38PM</td>
<td>65.87.165.114</td>
</tr>
<tr>
<td>1331244570</td>
<td>3/8/2012 2:09PM</td>
<td>71.10.106.181</td>
</tr>
<tr>
<td>1331261196</td>
<td>3/8/2012 6:46PM</td>
<td>76.102.156.138</td>
</tr>
</tbody>
</table>

**Arrow Memory Buffer (columnar format)**
Arrow Data Types

- **Scalars**
  - Boolean
  - [u]int[8,16,32,64], Decimal, Float, Double
  - Date, Time, Timestamp
  - UTF8 String, Binary

- **Complex**
  - Struct, List, Union
# Columnar Data

<table>
<thead>
<tr>
<th>Validity Buffer (bitmap)</th>
<th>Data Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>0</td>
<td>'x'</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>'x'</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
</tr>
<tr>
<td>0</td>
<td>'x'</td>
</tr>
</tbody>
</table>

- **Fixed Width 4 Byte INT Vector**

<table>
<thead>
<tr>
<th>Validity Buffer (bitmap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>'s'</td>
</tr>
<tr>
<td>'i'</td>
</tr>
<tr>
<td>'d'</td>
</tr>
<tr>
<td>'v'</td>
</tr>
<tr>
<td>'a'</td>
</tr>
<tr>
<td>'i'</td>
</tr>
<tr>
<td>'s'</td>
</tr>
<tr>
<td>'h'</td>
</tr>
</tbody>
</table>

**VARCHAR Vector**
Columnar Data

```javascript
persons = [{
    name: 'sidd',
    id: 1234,
    addresses: [
        {number: 2, street: 'a'}],
}, {
    name: 'vaish',
    iq: 5678,
    addresses: [
        {number: 4, street: 'cc'},
        {number: 5, street: 'ddd'}
    ]
}]
```
Messaging

- Fast language agnostic metadata messaging layer (using Google Flatbuffers library).
- Self-describing binary wire formats (streaming and file like) for remote procedure calls (RPC) and interprocess communication (IPC).
- Schema Negotiation
  - Logical Description of structure
  - Identification of dictionary encoded Nodes
- Dictionary Batch
  - Dictionary ID, Values
- Record Batch
  - Batches of records up to 64K
- Streaming Format: sequence of encapsulated Arrow messages
- File Format: Similar to streaming format. Supports random access.


Schema consists of sequence of fields which are metadata describing the columns

© 2018 Dremio Corporation  @DremioHQ
Record Batch Construction

Record Batch contains a depth-first (pre-order) flattened set of field metadata along with physical memory buffers for each vector.

```json
{
  name: 'Joe',
  age: 18,
  phones: [
    '555-111-1111',
    '555-222-2222'
  ]
}
```

Each box (vector) is contiguous memory.
Language Bindings

• Java Library
• C++ Library
• C Library
• Ruby Library
• Python Library
• JavaScript Library
Arrow in Dremio

• Dremio is an OSS self-service data platform
• The core engine is “Sabot”
  – Built entirely on top of Arrow libraries, runs in JVM.
Sabot: In-Memory Columnar Query Execution Engine

- Memory Management
- Vectorized Query Processing
  - Vector sizing
  - SIMD processing
  - Vectorized hash aggregation, columnar pivot
  - Vectorized copy
  - Columnar compression
  - Vectorized Parquet Reader
  - Late Materialization
  - Unnesting lists
Memory Management

• Arrow includes chunk-based managed allocator
  – Built on top of Netty’s JEMalloc implementation

• Create a tree of allocators
  – Support both reservation and local limits
  – Include leak detection, debug ownership logs and location accounting
  – Each operator in query plan gets its own allocator and child allocators are created for specific pieces of work within the operator.

• Size allocators (reservation and maximum) based on workload management, when to trigger spilling, etc.

• Reference counted off-heap buffers in each vector.
Memory Management Cont’d

• Data moves through data pipelines
• Ownership needs to be clear (to plan/control execution
  – Allocated memory can be referenced by many consumers
  – One allocator ‘owns’ the accounted memory
  – Consumers can use Vector’s transfer capability to leverage transfer semantics and handoff data ownership for a batch of records.
  – In the diagram, a record batch (set of vectors for all columns) is transferred as output of scan operator into aggregation operator.
Vectorized Execution

• Traditional:
  – Tuple at a time iterator based model.
  – Push tuple-at-a-time through query plan tree
• Vectorized:
  – Columnar processing.
  – A vector containing fixed number of values from a column.
  – Improves the overall utilization of both CPU-Memory and disk I/O bandwidth.
  – Better utilization of CPU cache by passing blocks (vectors) of tuples between operators.
  – Cache lines are potentially filled with data of interest.
  – Efficient tight for-loop access.
Vectorized Execution

SELECT AVG (COL) FROM FOO WHERE COL > 1000
Vectorized Execution: Vector Sizing

- Batches are the smallest work unit
  - A batch comprises of column vectors
- Batches of records can be 1..64k records in size.
- Optimization Problem
  - Larger improve processing performance
  - Larger causes pipeline problems
  - Smaller causes more heap overhead
- Execution-Level Adaptive Resizing for wide records (100-1000s fields)
Vectorized Execution: Vector Sizing Cont’d

• With batch size, we control the amount of memory allocated for the vectors
  – Helps in memory intensive operations like **external sort**, aggregation, join.
  – Not to exceed the memory reservations the task is working under.
  – Very important in **spilling code** in order to gracefully handle out of memory conditions.

`vector.setInitialCapacity(records) -> vector.allocate(records)`
Vectorized Execution - SIMD Acceleration

Accelerate SCAN, FILTER, SUM, and other operations through SIMD.
Vectorized Execution: Hash Aggregation

For generating hash table, maintaining a columnar structure for keys slows hashing insertion and lookup

- Break hash table into fixed and variable blocks
  - Contiguous memory regions.
  - Called as pivot space.
  - Store row-wise representation of keys.

- Pivot columnar data into pivot space
  - Vector at time for fixed values
  - All variable at same time for variable vectors.

- Compute hash on pivoted data (both fixed and variable width keys in row)
- Insert into hash table.
- Avoids excessive indirection.
- Maintain Aggregation tables in columnar format
  - Both input and output vector
  - Set of bits from hash table insertion ordinal point to the target computed value in accumulator vector.
Vectorized Execution: **Hash Aggregation Cont’d**

- **Fixed Block Vector**
  - `validity | fixed1 | fixed2 | varlen | varoffset`

- **Variable Block Vector**
  - `len | data | len | data | len | data | len | data | varlen | varoffset`

**Aggregation Tables**

```
pivot fixed
```

```
pivot variable
```

```
unpivot
```

```
direct projection
```

© 2018 Dremio Corporation  @DremioHQ
Vectorized Execution: Vectorized Copy

• Copy records between input and output record batches
• **Straight copier:**
  – use vector-to-vector transfer.
• **Selection vector based copier:**
  – Use the 2 byte offset from the selection vector to index the input vector and get the source data for copy.
  – Use the 4 byte selection vector to index multiple batches and records within a batch.
• **High performance C/C++ style code:**
  – Directly work with underlying memory of vector buffers.
  – Keep shifting the source and target addresses in a for loop.
  – `PlatformDependent.putInt(addr, value)`
  – `PlatformDependent.putLong(addr, value)`
  – `PlatformDependent.getByte(addr)`.....
Vectorized Execution: Efficient Columnar Compression

• General purpose compression algorithms optimize for compression ratio
  – LZO, ZLIB give good compression ratios but typically hurt CPU efficiency.
• Several lightweight compression schemes are preferred
  – Favor query performance over compression ratio.
• Use compression on per column basis. Determine most effective method
  – Cardinality
  – Sorted?
  – Data type
• Compress column values into fixed width dense arrays.
  – Encoded column values can be packed into arrays.
  – Amenable to SIMD acceleration
• Dictionary Encoding
  – Rewrite FILTER on strings as FILTER on fixed width dictionary values.
  – Dictionary encoded values can be further compressed using bit packing etc.
Vectorized Execution: Operating on compressed data

```
SELECT .... FROM T
WHERE COUNTRY='FRANCE'
```

```
<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Encoded values in column</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNITED STATES</td>
<td>1</td>
</tr>
<tr>
<td>CHINA</td>
<td>2</td>
</tr>
<tr>
<td>INDIA</td>
<td>3</td>
</tr>
<tr>
<td>FRANCE</td>
<td>4</td>
</tr>
<tr>
<td>UNITED KINGDOM</td>
<td>5</td>
</tr>
<tr>
<td>CANADA</td>
<td>6</td>
</tr>
<tr>
<td>GERMANY</td>
<td>7</td>
</tr>
<tr>
<td>ITALY</td>
<td>8</td>
</tr>
</tbody>
</table>
```

```
4 | 2
4 | 5
4 | 4
4 | 3
4 | 1
4 | 7
......
```

PARALLEL COMPARE

© 2018 Dremio Corporation

@DremioHQ
Vectorized Execution: **Vectorized Parquet Reader**

- **Parquet <-> Arrow conversion**
  - Sabot leverages the on-disk columnar format for storing highly optimized materialized views.
  - Compliments the in-memory columnar format (Arrow) for execution.
- **Columnar operations while reading and writing Parquet files.**
- **Support filter push down for Parquet scans.**
- **Read plain, dictionary encoded, RLE, BitPacked pages.**
- **Decode the run lengths in definition levels to bulk load column data.**
Vectorized Execution: **Late Materialization**

- Work on columnar format till late in the query plan
  - Delay tuple reconstruction.
  - The idea is not to read columnar data and stitch the values together to perform row-store operations on tuples.
- Improves utilization of memory bandwidth.
  - Potentially avoid constructing unnecessary tuples.
- Use selection vectors (intermediate ordinal lists) like structures:
  - FILTER on one column followed by PROJECT on second.
  - Use a 2 byte selection vector for FILTER column.
- Late Materialization along with ability to directly operate on compressed data heavily improves utilization of memory bandwidth.
Unnesting List Vectors

- Common Pattern: List of objects that want to be unrolled to separate records.
- Arrow’s representation allows a direct unroll (no inner data copies required)
- Since leaf vectors can be larger (up to 2B), may need to split apart inner vectors
  - SplitAndTransfer vector contents as necessary.
  - SplitAndTransfer as cheap as possible
    - NO-OP for data buffers (both fixed and variable width vectors).
    - Offset rewrite for variable width vectors.
    - Bit rewrite & shifting for validity buffer
Get Involved

• Join the Arrow community
  – dev@arrow.apache.org
  – Slack:
    • https://apachearrowslackin.herokuapp.com/
    – http://arrow.apache.org
    – Follow @ApacheArrow, @DremioHQ, @siddcoder

• Join the Dremio community
  – https://community.dremio.com/
  – https://github.com/dremio/dremio-oss
Free advance copy of our book!

No BI User Left Behind*
by Tomer Shiran

Available in booth 1520. Please visit our booth to learn more about our platform.

*No data scientists left behind either