Building Composable Data Microservices with Apache Arrow

Community Over Code 2023
October 7, 2023

Peter Leicht, Engineering Team Lead
Shoumyo Chakravorti, Software Engineer
Bloomberg Indices Engineering

TechAtBloomberg.com
Who are we? (Bloomberg)

- 8,000+ software engineers

- Product areas:
  - Data
  - Analytics
  - News
  - Communication
  - Electronic trading

- Lots of teams who need to process large datasets and publish analytics
Exponential Growth in Market Data Ingestion
Who are we? (Bloomberg Indices)

1. Ingest bond data
   Daily bond data is acquired from various internal and external data sources into our data stores, after which it is cleaned, massaged, and enriched.

2. Compute indices
   For each index, identify the members and their relative weights in the index, and compute index-level statistics.

3. Validate results
   Product/Data/Operations validates any potentially anomalous data points.

4. Publish to clients
   Once the data has been validated, the indices are published to the respective clients.
Who are we? (Bloomberg Indices)

- **Fixed Income Indices**
  - $50 Trillion dollar market value
  - Batch processing
  - Tight SLAs – clients are expecting reports ASAP

- **Data scale**
  - 200K+ bonds (rows), 1000s of columns of data per day
  - 25K+ indices produced each day
  - Up to 60-70K constituents (rows) per index
Why do we care about microservices?
Where did we come from?
Monolith: Pros & Cons

😄 Pros
- Performance
- Programming practices

😔 Cons
- Coupling
- Impact radius
- Ownership
  - Who owns the Monolith?
- Release cadence
- Choice of languages
- Build processes
Splitting things out into services
Going all the way

😊 Pros
- Decoupled
- Impact radius
- Ownership
- Release cadences
- Choice of language
- Build processes
New complexities…

😊 Cons
- Performance
- Development costs
Problem 1: Performance
Problem: CPU encoding and decoding

- CPU time spent encoding/decoding data for large requests/responses (~64MB)
  - C++ to C++: 50%
  - C++ to Python: 90%
Solution: Scaling out?

😄 Pros
- Latency

😊 Cons
- Band-Aid solution
  - Encode / decode still exists
  - Resource wastage
- Operational complexity
Solution: Libraries?

**Pros**
- Latency
- No encoding/decoding

**Cons**
- Becoming a monolith
- Multiple versions
- Access control
The “composability theorem”

- Must have composability
- Trading simplicity for performance
Problem 2: Fragmentation
Fragmentation is inevitable
Problem: Lack of standardization

- Every service defines its own schema and internal data model
- Lots of time spent designing schemas for new systems
- Lots of time spent connecting systems
Solution: Shared data model?

😊 Pros
- Some consistency across service schemas

😔 Cons
- Increased schema complexity
  - Variant/union types
- In-memory format != on-wire format
Solution: Organic localized standardization

😊 Pros
- Some reusability
- Overall less fragmentation

😢 Cons
- Local optimum
- Performance problems unsolved
## In Summary

<table>
<thead>
<tr>
<th></th>
<th>Monolith</th>
<th>Services only</th>
<th>Services + Libraries</th>
<th>Services + shared data models and converters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>✔️</td>
<td>❌</td>
<td>✔️</td>
<td>❌</td>
</tr>
<tr>
<td><strong>Isolation</strong></td>
<td>❌</td>
<td>✔️</td>
<td>!</td>
<td>✔️</td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❗</td>
</tr>
</tbody>
</table>
A theme emerges

● A good solution to composability would:
  ○ Standardize both in-memory and on-wire formats
  ○ Reduce the cost of integration
  ○ Not add unnecessary complexity
  ○ Not add significant performance penalties
In the meantime...
Offline analysis

- **Goal**: Using daily bond data for offline analysis
  - Compliance / troubleshooting
  - Product ideation
- Apache Parquet for analytics with Trino
- Apache Arrow as an intermediate format
What is Apache Arrow?

- Tabular
- Columnar
- Fast data transfer
- IPC ≈ In-memory
- Cache friendly
- SIMD friendly
Example uses of Apache Arrow

Data Transfer
- Apache Spark

Compute Engines
- Velox
- Polars

Visualization
- PERSPECTIVE
- Streamlit
New use-case

- Python application for data validation / reconciliation

- Python app experiencing significant slowness
  - Culprit: `ser/de` was taking 90% of the time!
  - Brute force was not an option…
Revisiting the status quo
How about Arrow?
How did we do this?

- Integrate Arrow IPC with middleware
  - e.g., Arrow Flight
- Got back almost all of the 90% ser/de time
- Just use pandas!
Adoption: slow but steady

- Headline features (in Indices)
  - For us, data transport is no longer a bottleneck!
  - Interoperability with pandas

- Growing usage across Bloomberg
  - In-memory data format for custom analytics engines
  - Faster data transport across applications
Can Apache Arrow solve our problems?

- Standardizes both in-memory and on-wire formats ✔
- Does not add significant performance penalties ✔
- Reduces the cost of integration? 🤔
- Does not add unnecessary complexity? 🤔
Taking it a step further
Kinds of business logic

- Enrichment
- Aggregation
- Validation
- Searching

<table>
<thead>
<tr>
<th>SecId</th>
<th>Currency</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBG001</td>
<td>USD</td>
<td>2.00</td>
</tr>
<tr>
<td>BBG002</td>
<td>GBP</td>
<td>9.50</td>
</tr>
<tr>
<td>BBG003</td>
<td>JPY</td>
<td>954.30</td>
</tr>
<tr>
<td>BBG004</td>
<td>USD</td>
<td>14.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Currency</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>1.00</td>
</tr>
<tr>
<td>JPY</td>
<td>149.30</td>
</tr>
<tr>
<td>GBP</td>
<td>0.83</td>
</tr>
</tbody>
</table>

bonds

<table>
<thead>
<tr>
<th>SecId</th>
<th>Price'</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBG001</td>
<td>2.00</td>
</tr>
<tr>
<td>BBG002</td>
<td>11.45</td>
</tr>
<tr>
<td>BBG003</td>
<td>6.39</td>
</tr>
<tr>
<td>BBG004</td>
<td>14.20</td>
</tr>
</tbody>
</table>

fx_rates

bonds’
How can we do it on Apache Arrow?
In-Memory Analytics
Compute Engines

- In-process library
- Declarative
- Query Optimization
- Computation
  - Vectorization vs. JIT
- Some are Arrow-native
Operations supported by compute engines

- Filter
- Project
- Aggregate
- Join
- Sort
- Window functions
- Pivot tables
- Unnest
- ...

TechAtBloomberg.com

© 2023 Bloomberg Finance L.P. All rights reserved.
Representing our business logic

- Can we represent our logic *declaratively*?
- Enrichment → *Join / Project*
- Aggregation → *GroupBy / Agg*
- Validation → *Project / Filter*
- Searching → *Filter*

```sql
SELECT
    b.*,
    (b.Price * fx.Rate) AS NormalizedPrice
FROM
    bonds b
LEFT JOIN
    fx_rates fx ON b.Currency == fx.Currency
```
Adopting DuckDB

- C++ API + query optimizer + vectorized engine
- Supports Arrow input/output
- Observed **order of magnitude** performance improvement
Achieving greater composability
Learnings
Economies of scale

- Lower amortized cost to build new tooling
  - A **standard** is the foundation for generalizability

- e.g., Arrow integration with custom middleware
  - No need to encode and decode anymore
  - Every new user gets the tooling for free
  - Every new user encourages other callers to onboard as well

- Lots of newer open-source tooling built around Arrow
  - Driving developer costs down

- Performance gains open up additional design options
Learning from others

- Community-building helped us standardize and build the vision
- Couldn’t have thought of all use-cases in isolation
- Alignment across teams is necessary to standardize effectively
A shift in mindset

● Arrow is not...
  ○ Just an in-memory format or...
  ○ Just an on-the-wire format

● It’s both
  ○ Selectively using Arrow for specific problems diminishes its value
    ■ e.g., Using Arrow for just data transport but not in-memory analytics
    ■ Still have to pay developer costs for implementing connector logic
Thank you!

We are hiring: https://www.bloomberg.com/careers

Contact us:
- Peter Leicht: pleicht@bloomberg.net
- Shoumyo Chakravorti: schakravorti@bloomberg.net