Vectorized Query Execution in Apache Spark at Facebook

Chen Yang

Spark Summit | 04/24/2019

About me

Chen Yang

- Software Engineer at Facebook (Data Warehouse Team)
  - Working on Spark execution engine improvements
  - Worked on Hive and ORC in the past

Agenda

- Spark at Facebook
- Row format vs Columnar format
- Row-at-a-time vs Vector-at-a-time processing
- Performance results
- Road ahead

Agenda

- Spark at Facebook
- Row format vs Columnar format
- Row-at-a-time vs Vector-at-a-time processing
- Performance results
- Road ahead

Spark at Facebook

- Largest SQL query engine at Facebook (by CPU usage)
- We use Spark on disaggregated compute/storage clusters
  - Scale/upgrade clusters independently
- Efficiency is top priority for Spark at Facebook given the scale
  - Compute efficiency: Optimize CPU and Memory usage
  - Storage efficiency: Optimize on disk size and IOPS

Spark at Facebook

- Compute efficiency : Optimize CPU and Memory usage
  - Significant percentage (>40%) of CPU time is spent in reading/writing
- Storage efficiency : Optimize on disk size and IOPS
  - Storage format have big impact on on-disk size and IOPS
  - Facebook data warehouse use ORC format

Agenda

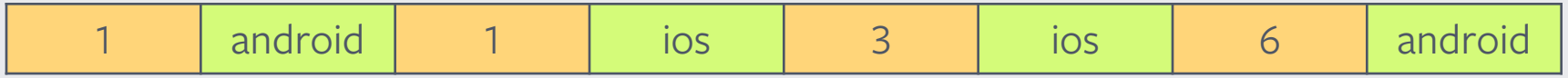
- Spark at Facebook
- Row format vs Columnar format
- Row-at-a-time vs Vector-at-a-time processing
- Performance results
- Road ahead

Row format vs Columnar format

Logical table

user_id	os
1	android
1	ios
3	ios
6	android

Row format



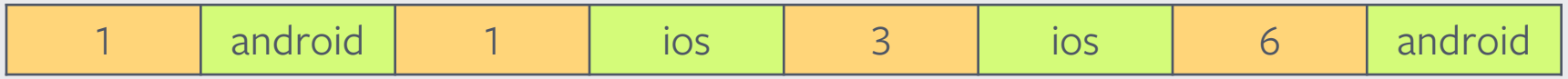
Columnar format

Row format vs Columnar format

Logical table

user_id	os
1	android
1	ios
3	ios
6	android

Row format



On disk row format:

csv

In memory row format:

UnsafeRow, OrcLazyRow

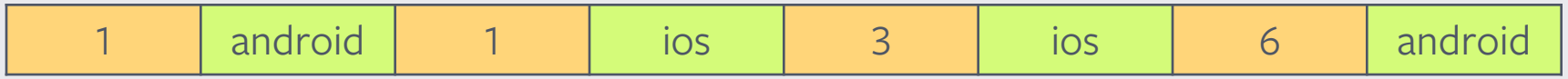
Columnar format

Row format vs Columnar format

Logical table

user_id	os
1	android
1	ios
3	ios
6	android

Row format



On disk row format:

csv

In memory row format:

UnsafeRow, OrcLazyRow

Columnar format

On disk columnar format:

Parquet, ORC

In memory columnar format:

Arrow, VectorizedRowBatch

Columnar on disk format

Logical table

user_id	os
1	android
1	ios
3	ios
6	android

- Better compression ratio
  - Type specific encoding
- Minimize I/O
  - Projection push down (column pruning)
  - Predicate push down (filtering based on stats)

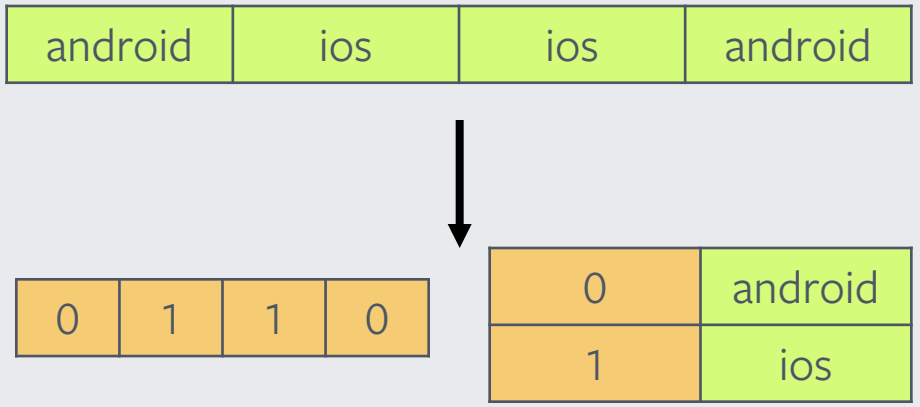
Columnar format

1	1	3	6	android	ios	ios	android

Better compression ratio

- Encoding
  - Dictionary encoding
  - Run Length encoding
  - Delta encoding
- Compression
  - zstd
  - zlib
  - snappy

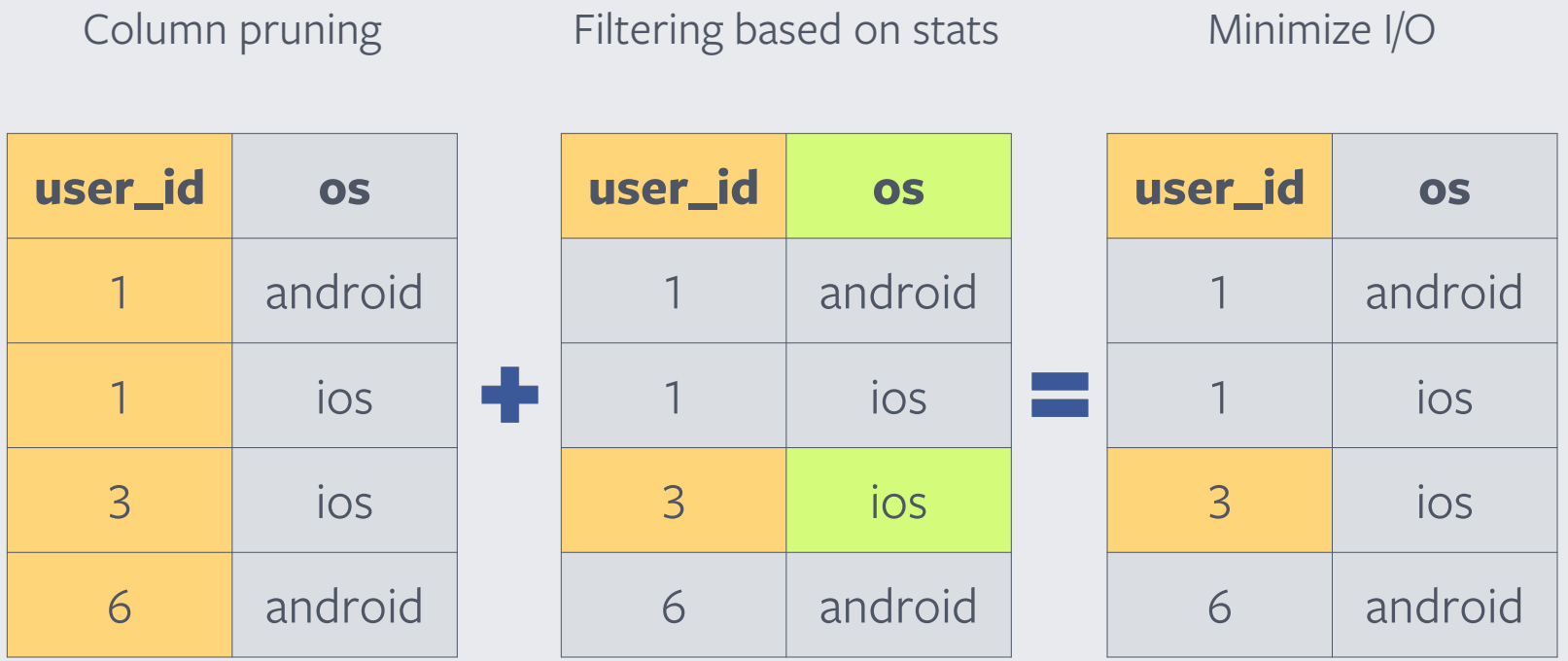
Dictionary encoding



Minimize I/O

- Projection push down
  - Column pruning
- Predicate push down
  - Filtering based on stats

```
SELECT
   COUNT(*)
FROM user_os_info
WHERE user_id = 3
```



ORC in open source vs at Facebook

Open source:

- ORC(Optimized Row Columnar) is a self-describing type-aware columnar file format designed for Hadoop workloads.
- It is optimized for large streaming reads, but with integrated support for finding required rows quickly.

Facebook:

- Fork of Apache ORC, Also called DWRF
- Various perf improvements (IOPS, memory etc)
- Some special format for specific use case in Facebook (FlatMap)

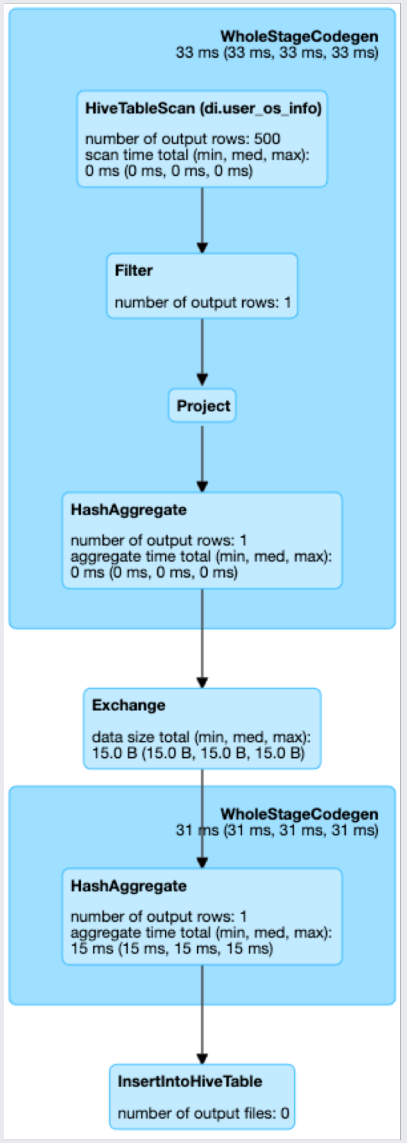
Agenda

- Spark at Facebook
- Row format vs Columnar format
- Row-at-a-time vs Vector-at-a-time processing
- Performance results
- Road ahead

Row-at-a-time vs vector-at-a-time

```
SELECT
    COUNT(*)
FROM user_os_info
WHERE user_id = 3
```

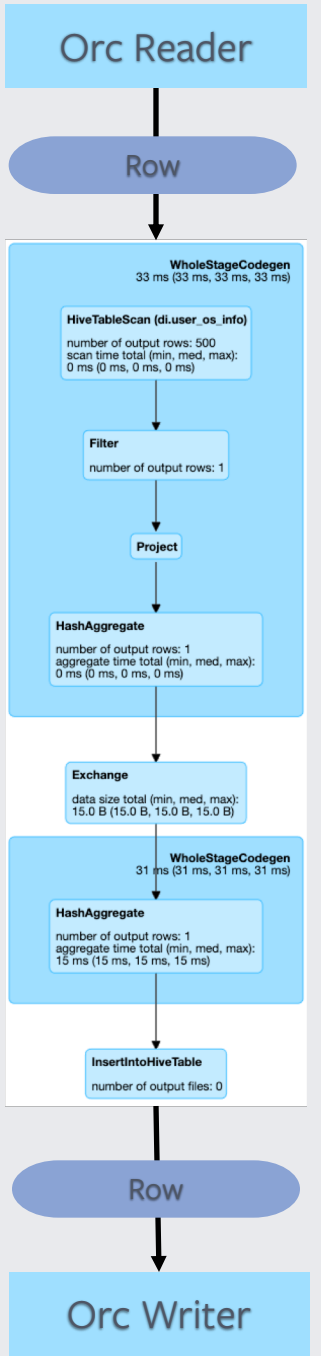
user_id	os
1	android
1	ios
3	ios
6	android



Row-at-a-time processing

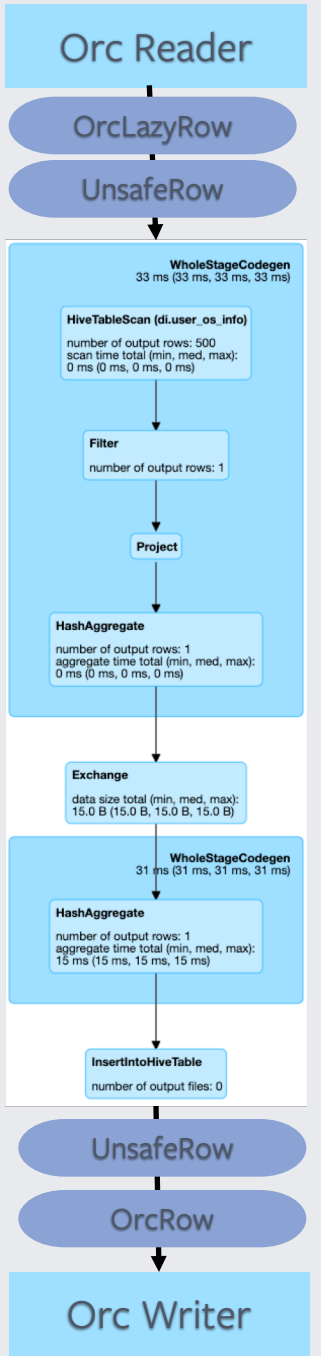
```
SELECT
    COUNT(*)
FROM user_os_info
WHERE user_id = 3
```

user_id	os
1	android
1	ios
3	ios
6	android



Row-at-a-time processing

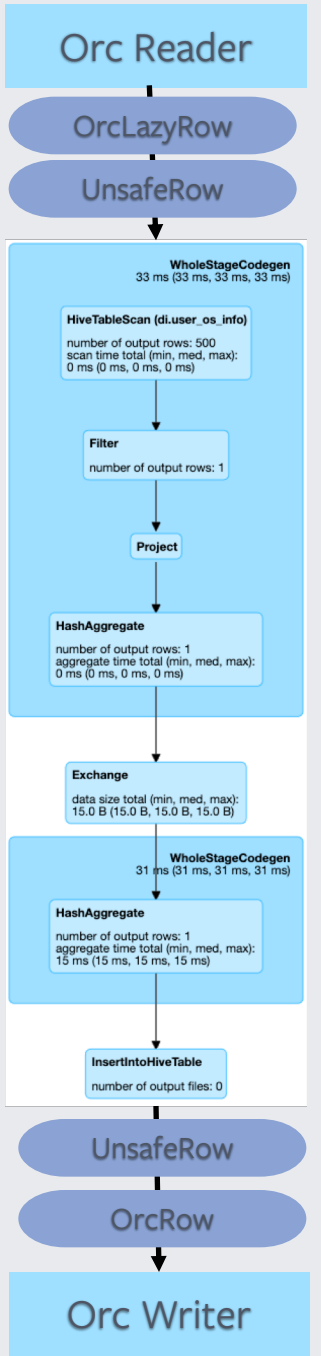
```
SELECT
    COUNT(*)
FROM user_os_info
WHERE user_id = 3
```



Row-at-a-time processing

```
SELECT
    COUNT(*)
FROM user_os_info
WHERE user_id = 3
```

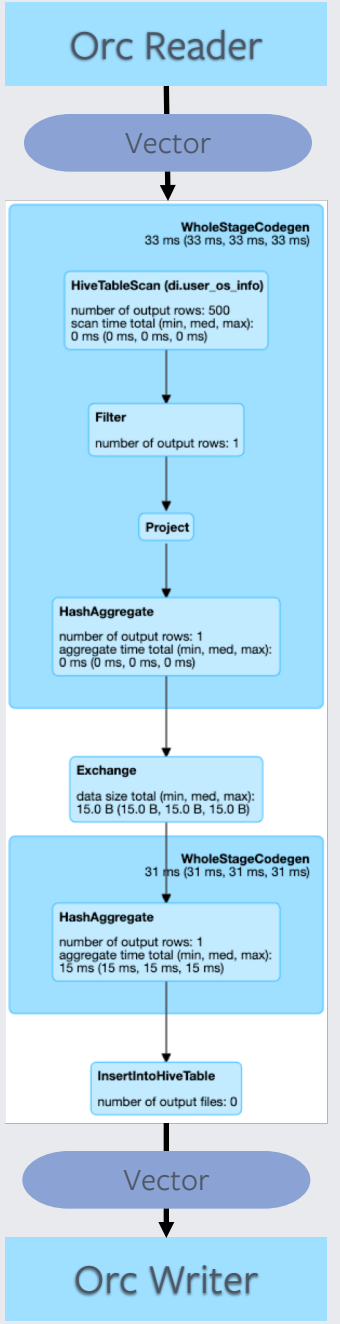
```
private void agg_doAggregateWithoutKey() {
    while (inputadapter_input.hasNext()) {
        inputadapter_row = inputadapter_input.next();
        value = inputadapter_row.getLong(0);
        // do aggr
    }
}
```



Vector-at-a-time processing

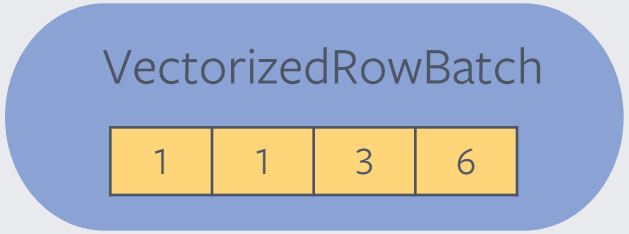
```
SELECT
    COUNT(*)
FROM user_os_info
WHERE user_id = 3
```

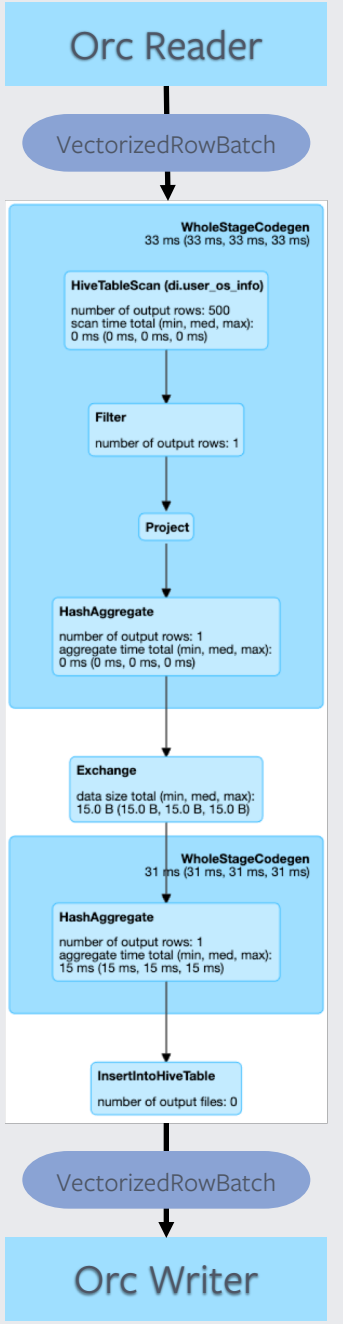
user_id	os
1	android
1	ios
3	ios
6	android



Vector-at-a-time processing

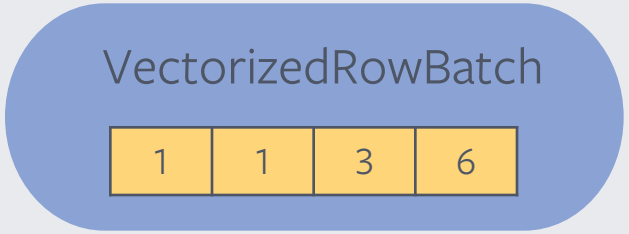
```
SELECT
    COUNT(*)
FROM user_os_info
WHERE user_id = 3
```



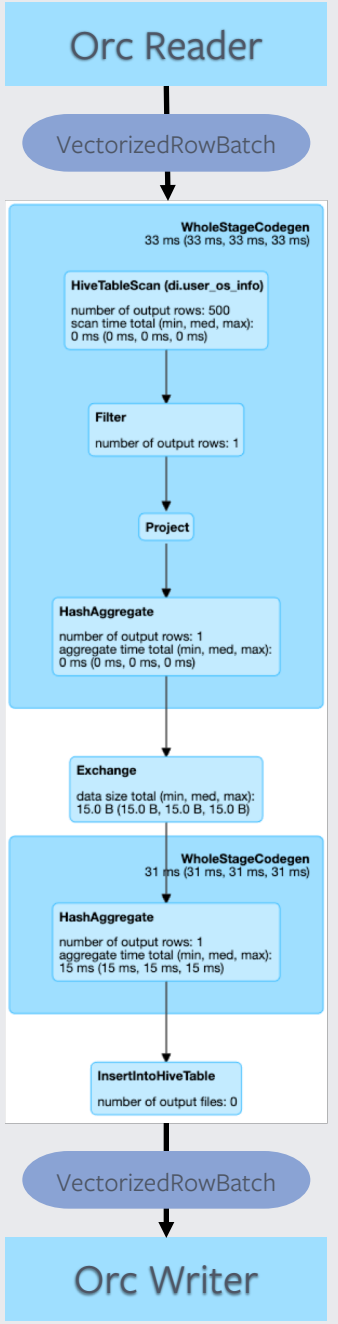


Vector-at-a-time processing

```
SELECT
    COUNT(*)
FROM user_os_info
WHERE user_id = 3
```




```
private void agg_doAggregateWithoutKey() {
    while (orc_scan_batch != null) {
        int numRows = orc_scan_batch.size;
        while (orc_scan_batch_idx < numRows) {
            value = orc_scan_col_0.vector[orc_scan_row_idx];
            orc_scan_batch_idx++;
            // do aggr
        }
        nextBatch();
    }
}
private void nextBatch() {
    if (orc_scan_input.hasNext()) {
        batch = orc_scan_input.next();
        col_0 = orc_scan_batch.cols[0];
    }
}
```



Row-at-a-time vs vector-at-a-time

- Lower overhead per row
  - Avoid virtual function dispatch cost per row
  - Better cache locality
- Avoid unnecessary copy/conversion
  - No need to convert between OrcLazyRow and UnsafeRow

Row-at-a-time processing

```
private void agg_doAggregateWithoutKey() {
    while (inputadapter_input.hasNext()) {
        row = inputadapter_input.next();
        value = inputadapter_row.getLong(0);
        // do aggr
    }
}
```

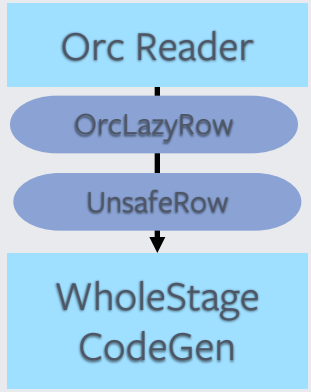
Vector-at-a-time processing

```
private void agg_doAggregateWithoutKey() {
    while (orc_scan_batch != null) {
        int numRows = orc_scan_batch.size;
        while (orc_scan_batch_idx < numRows) {
            value = col_0.vector[orc_scan_row_idx];
            orc_scan_batch_idx++;
            // do aggr
        }
        nextBatch();
    }
}
private void nextBatch() {
    if (orc_scan_input.hasNext()) {
        batch = orc_scan_input.next();
        col_0 = orc_scan_batch.cols[0];
    }
}
```

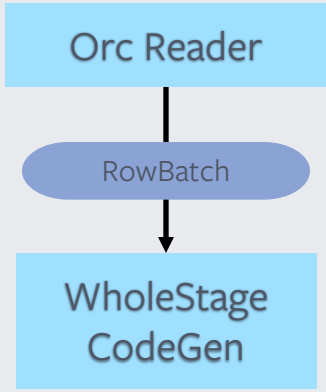
Row-at-a-time vs vector-at-a-time

- Lower overhead per row
  - Avoid virtual function dispatch cost per row
  - Better cache locality
- Avoid unnecessary copy/conversion
  - No need to convert between OrcLazyRow and UnsafeRow

Row-at-a-time processing



Vector-at-a-time processing



Vectorized ORC reader/writer in open source vs at Facebook

Open source:

- Vectorized reader only support simple type
- Vectorized writer is not supported

Facebook:

- Vectorized reader support all types, integrated with codegen
- Vectorized writer support all types, plan to integrated with codegen

Vector-at-a-time + Whole Stage CodeGen

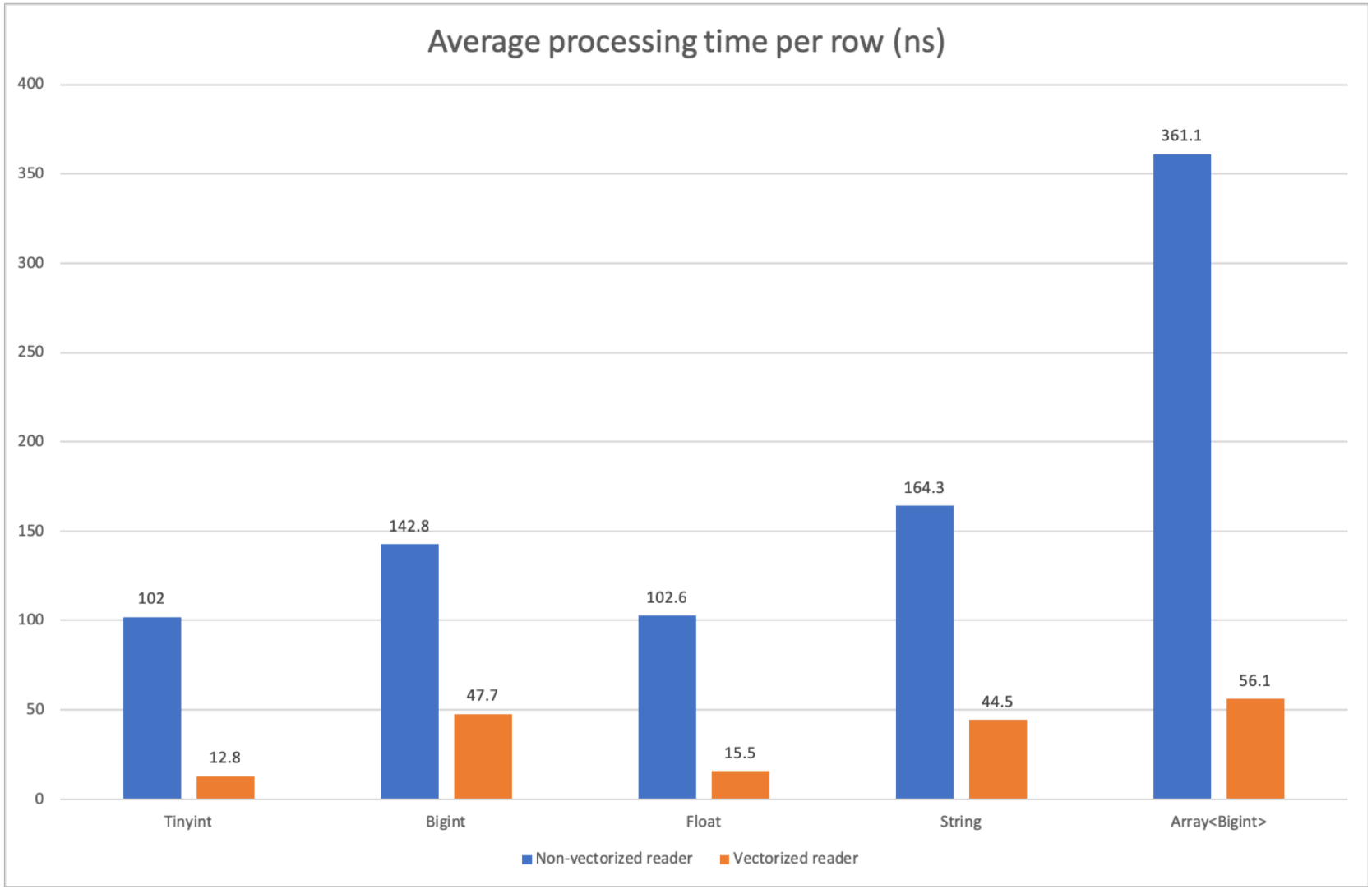
- Currently only HiveTableScan and InsertIntoHiveTable understands ORC columnar format
- Most of Spark Operators still process one row at a time

Agenda

- Spark at Facebook
- Row format vs Columnar format
- Row-at-a-time vs Vector-at-a-time processing
- Performance results
- Road ahead

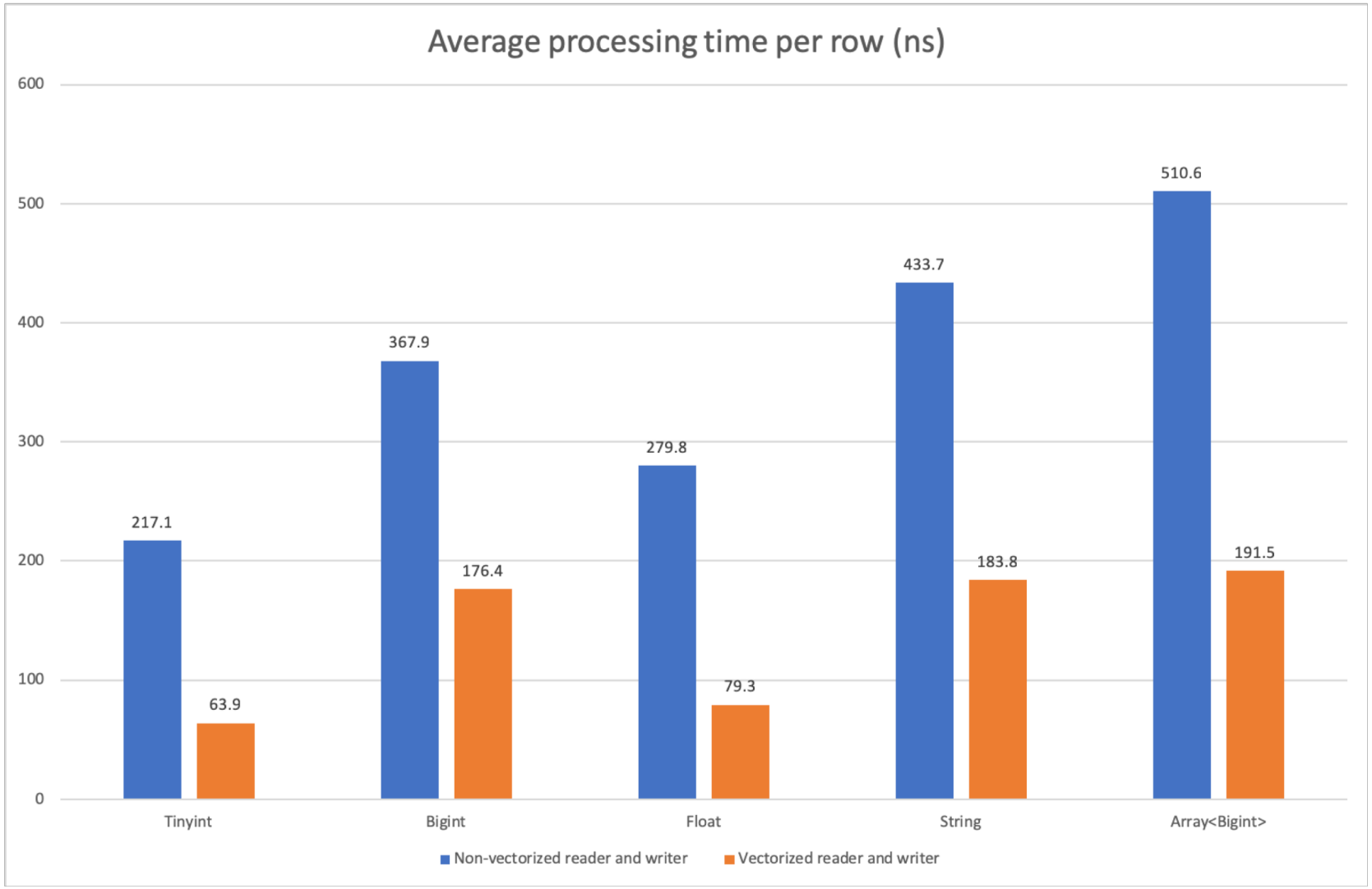
Vectorized reader + vector-at-a-time microbenchmark

Up to 8x speed up when reading 10M row from a single column table



Vectorized reader and writer + vector-at-a-time microbenchmark

Up to 3.5x speed up when reading and writing 10M row from a single column table

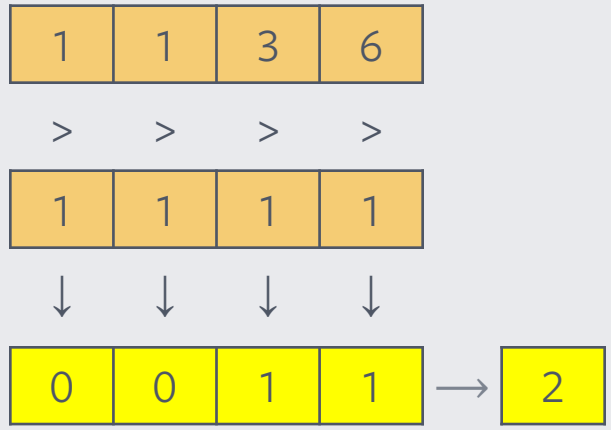


Agenda

- Spark at Facebook
- Row format vs Columnar format
- Row-at-a-time vs Vector-at-a-time processing
- Performance results
- Road ahead

Road ahead

- SIMD/Avoid branching/prefetching
  - Optimize codegen code to trigger auto-vectorization in JVM
  - Customize JVM to make better compiler optimization decisions for Spark

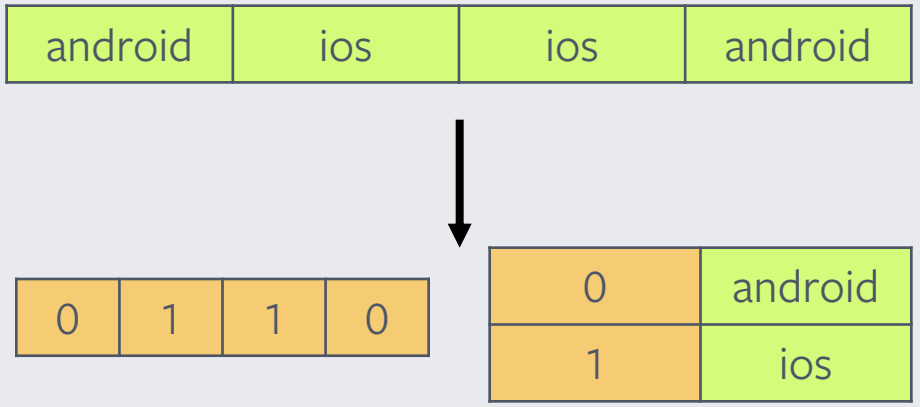



```
SELECT
   COUNT(*)
FROM user_os_info
WHERE user_id > 1
```

Road ahead

- Efficient in memory data format
  - Use Apache Arrow as in memory format
  - Encoded columnar in memory format
  - Use columnar format for shuffle
- Filter/projection push down to reader
  - Avoid decoding cost

Dictionary encoding




```
SELECT
    COUNT(*)
FROM user_os_info
WHERE os = 'ios'
```


facebook
INFRASTRUCTURE