WeChat’s Feature Compute Engine for Real-time Recommender Systems

Jin Shang, Software Engineer, Tencent WeChat

FEATURE STORE SUMMIT 2023

Organized by HOPSWORKS

Who am I?

Present:

- Software Engineer @ WeChat's ML Platform Team
- Lead developer of our vectorized feature compute engine
- Active community contributor of Apache Arrow

Previous:

- SWE Intern @ Google Cloud Vertex AI Feature Store
- Masters in Computer Science @ CMU

Background

Machine learning at WeChat

- Recommender Systems
  - Ads, Articles, Videos, Live Streams, Feeds, Search Ranking...
- Trust and Safety
  - Fraud Detection
  - Content Moderation
- Internal Usage
  - Code Generation
  - Documentation Chatbot

Articles

Videos

Mini-apps

WeChat Pay

Features Engineering For Recommendation

- Offline feature *generation*
  - Fixed-interval Batch jobs: **Apache Spark**
  - Event Triggered Streaming Jobs: **Apache Flink**
- Online feature *extraction*
  - Transform data in DB to features for models
  - OLAP-like operations on small batches: <1000 items per request
  - Requires low latency: <50ms for a request
  - Facebook F3 / OpenMLDB Online Engine

Recommender System: Online Feature Extraction

Operations that are impossible or expensive offline:

- Filtering: Eliminate invalid/abnormal values
- Joining: Cross user and item features online to save key space
- Sorting and Aggregation: # of request < # of actions
- Numerical transformations: Different parameter for each model
    - Discretization
    - Hashing

Recommender System: Online Feature Extraction

Workload pattern:

- Each request contains a single user and multiple items
- The same set of features are computed for each item
- Features have fixed computation logic once deployed
- The same computation tasks are repeated over and over

Conclusion:

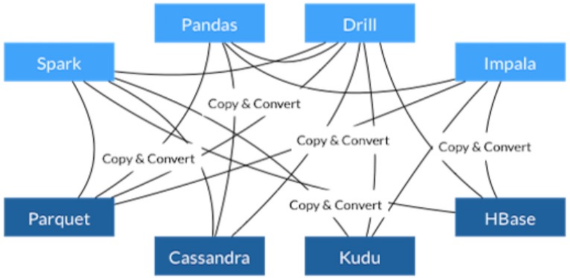
- Perfect scenario for **vectorization** and **compilation**

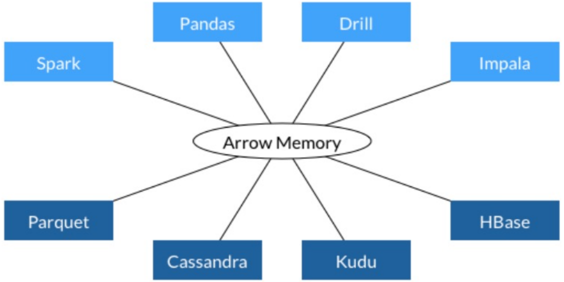
Technical Details

APACHE ARROW

Apache Arrow

- De facto standard for in memory columnar (vector) data layout
- Most widely used vector format in data analysis world
  - Ad hoc analysis: PyArrow, Pandas, Polars...
  - OLAP engine: DuckDB, Velox, Datafusion...
  - ML data: Huggingface Dataset, Ray Dataset
- A complete toolbox for vector data:
  - Primitive, List, Map, Struct, Union, Extensions...
  - IO, Serialization, Compute...
  - C, C++, Rust, Python, Go, Java, Matlab, Julia, JS...





Feature Representation with Apache Arrow

- Each feature value is represented as an Arrow Array or Scalar

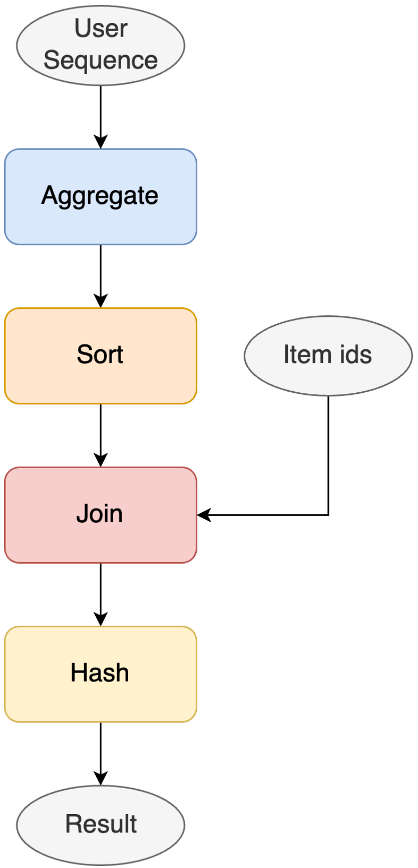
	User	Item
Scalar Feature	Scalar	Primitive Array
Sequence Feature	Primitive Array	List Array

- Item ratings: [9, 8, 9, null, 7], representing the value for item 1-5
  - Arrow arrays natively support null values
- Item tags: [["action", "horror"], ["romance"], [], null, ["comedy"]]
  - Nuanced difference between empty list and null list
  - Number of features: 0 vs null

FEATURE STORE SUMMIT 2023

Feature Computation with Apache Arrow

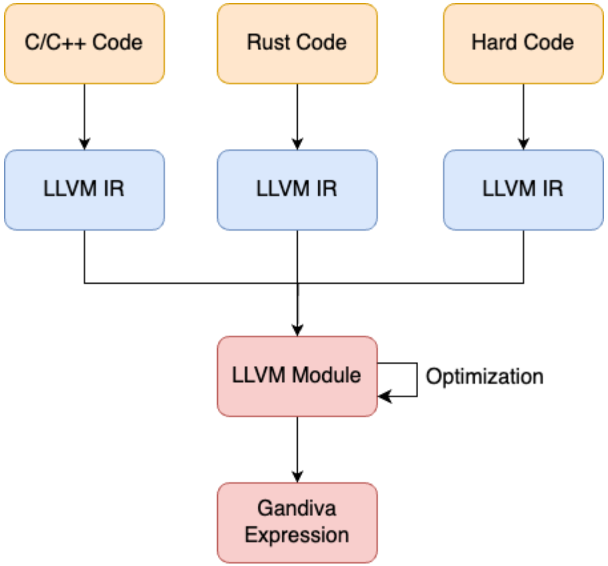
- We provide common operators such as Join, Sort, Math Expressions...
- User define a feature by combining operators
- Several ways to achieve SIMD vectorization on Arrow array
    - LLVM JIT Engine
    - Arrow native compute functions
    - Compiler auto-vectorization
    - Hand-written SIMD intrinsics



LLVM JIT Engine

Gandiva Expression Compiler

- Developed by Dremio, maintained by Arrow team
- Provides arithmetic functions pre-compiled to LLVM IR
- Combine functions into expression at LLVM IR level
- Leverages LLVM for various optimizations
  - loop vectorization
  - function in-lining
  - instruction combination
  - ...
- Used for Projection and Filtering



Gandiva Expression Compiler

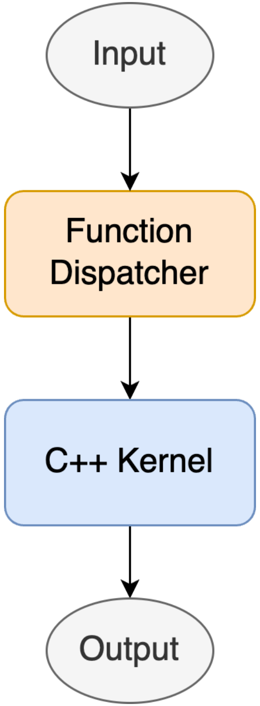
LLVM IR for “(a+b)*c” on ArmV8 Neon instruction set

```
vector.body:                                      ; preds = %vector.body, %vector.ph
  …
  %lsr.iv4042 = bitcast double* %lsr.iv40 to <2 x double>*
  %lsr.iv4547 = bitcast double* %lsr.iv45 to <2 x double>*
  %lsr.iv5052 = bitcast double* %lsr.iv50 to <2 x double>*
  %lsr.iv5557 = bitcast double* %lsr.iv55 to <2 x double>*
  …
  %21 = fadd <2 x double> %wide.load, %wide.load28
  %22 = fadd <2 x double> %wide.load27, %wide.load29
  %scevgep58 = getelementptr <2 x double>, <2 x double>* %lsr.iv5557, i64 -1
  %wide.load30 = load <2 x double>, <2 x double>* %scevgep58, align 8, !alias.scope !16
  %wide.load31 = load <2 x double>, <2 x double>* %lsr.iv5557, align 8, !alias.scope !16
  %23 = fmul <2 x double> %21, %wide.load30
  %24 = fmul <2 x double> %22, %wide.load31
  …
```

Arrow Compute Functions

Natively supported vectorized functions

- Gandiva is very fast, but...
  - Hard to develop and maintain
  - Hard to debug
  - Only used for simple operations such as math expressions
- Arrow Compute
  - C++ functions that are dynamically dispatched at runtime
  - Provides vectorized kernels for many functions
  - Supports various complex operations, e.g. Sort, Aggregate
  - All functions are exported to Python, easier to experiment with



Arrow Compute Functions

Common complex operations e.g. Sort, Aggregate, Join...

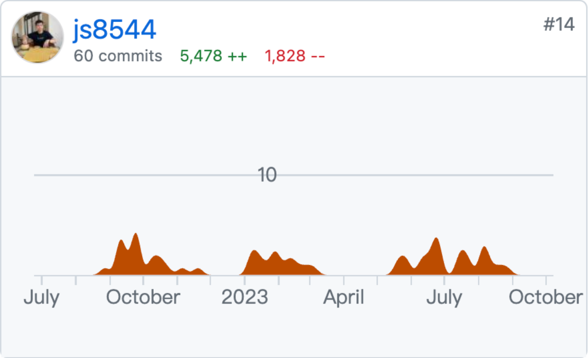
```
>>> import pyarrow as pa
>>> import pyarrow.compute as pc
>>> a = pa.array([5, 3, 4, 1, 2])
>>> sorted_indices = pc.sort_indices(a)
>>> pc.take(a, sorted_indices)
<pyarrow.lib.Int64Array object at 0x127a5d1e0>
[
  1,
  2,
  3,
  4,
  5
]
>>> pc.sum(a)
<pyarrow.Int64Scalar: 15>
```

```
>>> user_history_ids = pa.array([999, 777, 555])
>>> user_history_watch_time = pa.array([30, 50, 10])
>>> request_items = pa.array([111, 222, 777, 888, 999])
>>> join_index = pc.index_in(request_items,
user_history_ids)
>>> request_items_user_watch_time =
pc.take(user_history_watch_time, join_index)
>>> request_items_user_watch_time
<pyarrow.lib.Int64Array object at 0x127a5d4e0>
[
  null,
  null,
  50,
  null,
  30
]
```

Arrow Compute Functions

We have contributed several functions to Arrow

- cumulative_sum/prod/min/max
- rolling_sum/min/max
- adjoin_as_list
- pairwise_diff
- integer round functions
- arithmetic for temporal types
- ...



Compiler Auto Vectorization

- Arrow arrays are always stored in contiguous memory
- Gandiva/Compute too heavy for some light weight operations
- Rely on compilers to automatically vectorize
- GCC: -ftree-vectorize
- Clang: Enabled by default
- Example: Null bitmap bitwise and

```
for (int64_t i = 0; i < length; ++i) {
    arr1[i] &= arr2[i];
}
```

```
vmovdqu xmm0, XMMWORD PTR [rsi+rax]
add     rcx, 1
vinserti128     ymm0, ymm0, XMMWORD PTR [rsi+16+rax], 0x1
vpand   ymm0, ymm0, YMMWORD PTR [rdx+rax]
vmovdqa YMMWORD PTR [rdx+rax], ymm0
```

Handwritten SIMD with Intrinsics

Call SIMD intrinsic functions provided by Intel

- Compilers fail to vectorize complex operations, e.g. hashing
- Manual vectorization by calling SIMD intrinsics

```
template <int shift>
__attribute__((always_inline)) inline uint64 Rotate(uint64 val) {
  // Avoid shifting by 64: doing so yields an undefined result.
  if constexpr (shift == 0) {
    return val;
  } else {
    constexpr int kLeftShift = 64 - shift;
    return ((val >> shift) | (val << kLeftShift));
  }
}
```

```
template <int shift>
__attribute__((always_inline)) inline __m256i AVXRotate(__m256i val) {
  // Avoid shifting by 64: doing so yields an undefined result.
  if constexpr (shift == 0) {
    return val;
  } else {
    constexpr int kLeftShift = 64 - shift;
    auto val_sr_reg = _mm256_srli_epi64(val, shift);
    auto val_sl_reg = _mm256_slli_epi64(val, kLeftShift);
    return _mm256_or_si256(val_sr_reg, val_sl_reg);
  }
}
```

Performance Issue for Small Batch Size

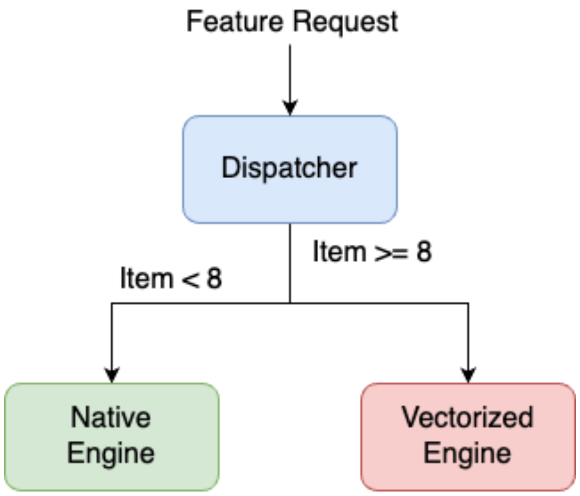
- Sometimes scenarios only have very few candidate items
  - Online training with one sample
  - Push notifications

Vectorized operators perform badly when batch size is small

- Array overhead
  - Need to store array metadata
  - Arrow array always aligned at 64 bytes
- Vectorization overhead
  - Compute more values than needed
- Compute Function overhead

Native Operators

- Provide a native implementation for all operators
  - Each item feature is stored separately as single values
- Dynamically dispatch to Native (<8) and Vectorized (>=8)



Consistency is ensured:

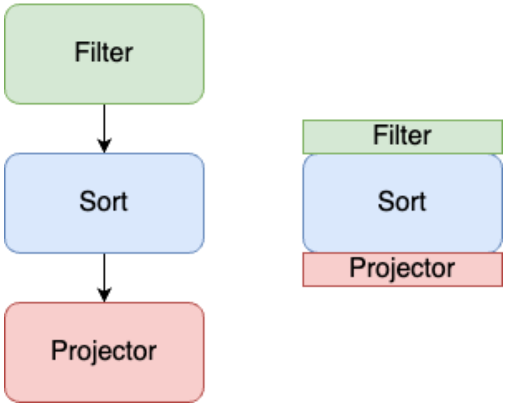
- Full test coverage
  - Operator level unit tests
  - Engine level E2E tests
- Daily pipeline to examine diffs from raw feature logs

Operator Fusion

- Overhead for each operator call
  - Virtual dispatch
  - Input validation and copy
  - Output memory allocation
- Most operators can be fused with Projection & Filtering
- Three operations done in one operator call
- Example:

log(x+1) ORDERED BY x WHERE x>0 AND x<100

PROJECTOR SORT FILTER

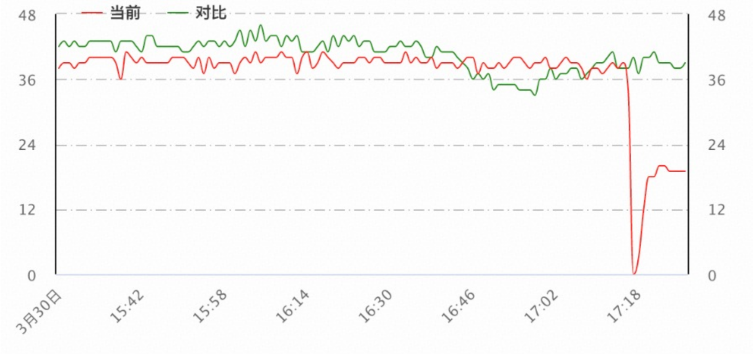


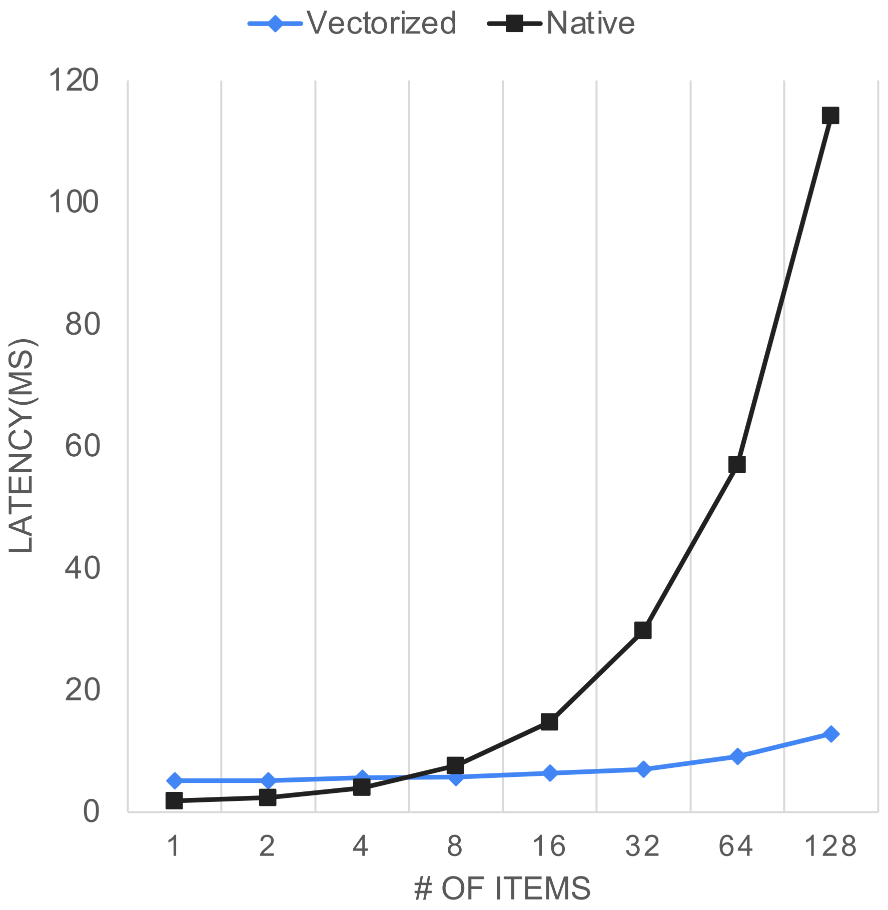
Performance Benchmark

Performance benchmark

- When item num < 8, native is better
- At 64, vectorized is 6x faster
- At 128, vectorized is 10x faster
- Overhead amortized for larger batch

- CPU usage down by 50% on deployment





Q&A