DATA+AI SUMMIT 2022

UIMeta:

A 10X Faster Cloud-Native Spark History Server

Lantao Jin

Head of Spark Engine Team, ByteDance

ORGANIZED BY databricks

About Me

I am Head of Spark Engine Team at ByteDance, focusing on SQL engine kernel and efficient platform building. I am a contributor of Apache Spark and Apache Hadoop and I am familiar with a variety of distributed systems. Prior to ByteDance, I worked for eBay Inc., Meituan Group, and Alibaba Group, where I worked on data platform and data warehouse infrastructure efforts.

Past sessions:

Summit Europe 2020: Using Delta Lake to Transform a Legacy Apache Spark to Support Complex Update/Delete SQL Operation

Summit 2019: Managing Apache Spark Workload and Automatic Optimizing

Linkedin:

https://www.linkedin.com/in/lantaojin/

Agenda

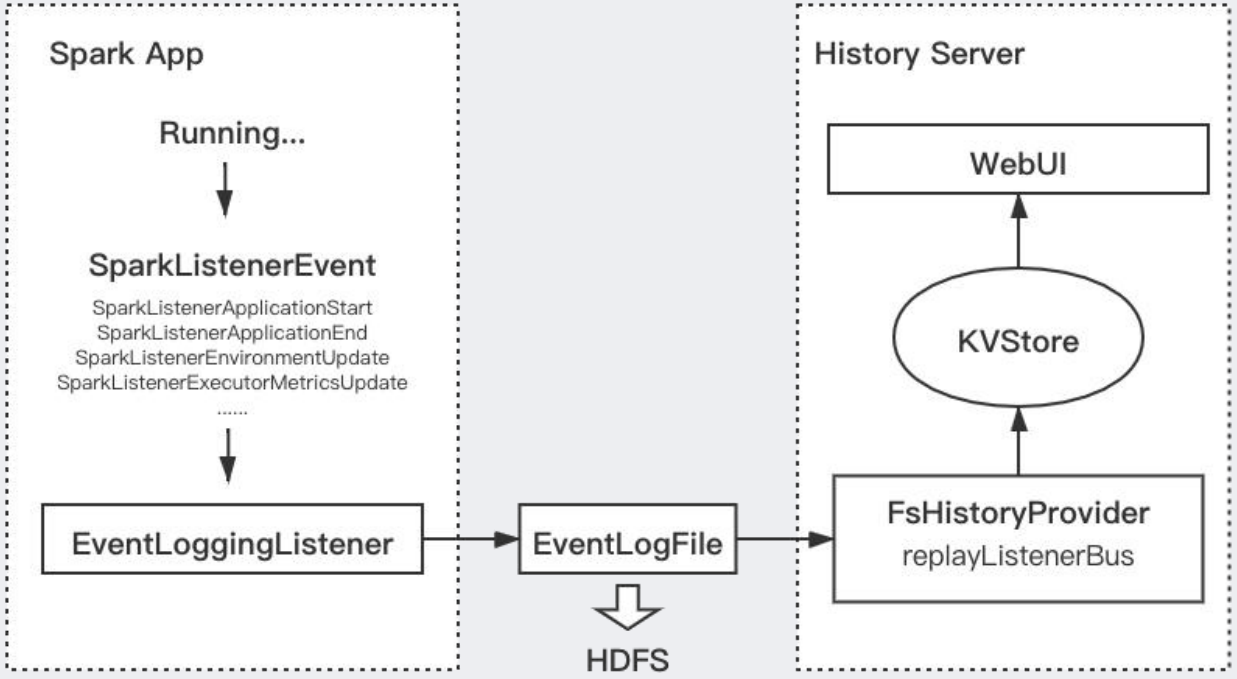
UIMeta: A 10X Faster Cloud-Native Spark History Server

1. Background
2. Design
3. Implementation
4. Results

Background

Spark History Server

The Spark history server (SHS) which is used to provide a web UI to display the historic Spark information.



Problems

of Spark History Server

Mass Storage

The Spark event log records almost everything. For UI display, most events are useless.

And the event log is stored in JSON plaintext, which takes up a lot of space.

The 7-days event log within ByteDance occupies about 3.2 PB in HDFS.

High Lantency

History Server builds the Spark UI by replaying and parsing the event log. Replaying a large task will cause a significant lantency.

After the job ends, users may have to wait 10 minutes or even half an hour to see the job in History Server, which greatly affects the user experience.

Scale-up

History Server traverses all event logs and load meta information for all files into memory, which makes it a stateful service. Therefore, every time the service is restarted, the entire path needs to be reloaded before it can be served.

This makes it hard to scale-out.

No Cloud-Native

History Server is not a pure cloud-native service. In public cloud scenario, it requires multi-tenancy and variety of workload.

We need a cloud-native history server for public cloud environment.

UIMeta & Volengine & LAS

UIMeta: a new Could-Native Spark history server

Data Engine team is responsible for LakeHouse Analytics Service (LAS).

LAS is one of products in Volengine, which is a Chinese public cloud platform.

You can try LAS via https://www.volcengine.com/product/las.

Spark is one of the engines in LAS. The UIMeta is the default Spark history service in LAS.

Design

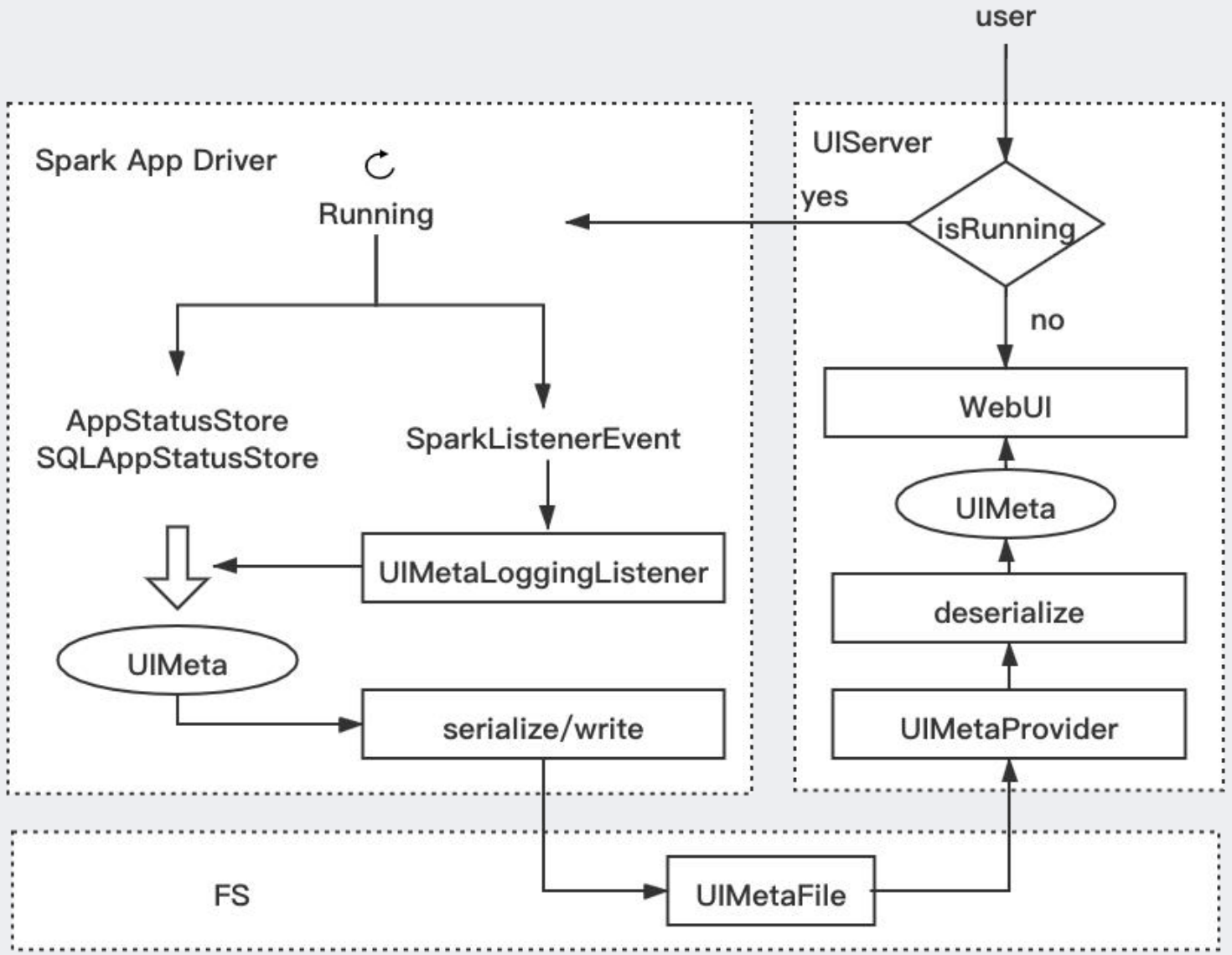
High Level Design

UIMeta abandons the event-log, attempts to loading SNAPSHOT file as an alternative.

On one hand, a running Spark application dumps a snapshot file in batch. On the other hand, the UIMeta server deserialized the snapshot file and build the page on demand.

UIMeta

Architecture



Key Points

of design

1. History Server only needs to care about the final state of app, not the events that cause the state change. Therefore, we can only persist KVStore without storing a lot of redundant event information.

2. KVStore stores all information required for UI display, and supports Kryo serialization, which its storage is significantly smaller than JSON.

3. Storing the KVStore async as a snapshot in a new listener.

4. For each access, the new UIMeta can find the corresponding snapshot HDFS path with the appId in URI according to a rule, and load it directly.

Implementation

KVStore vs UIMetaStore

A UIMetaStore is a collection of all UI information.

KVStore

AppStatusStore

SQLAppStatusStore

UIMetaStore

```
Array(
  classOf[JobDataWrapper],
  classOf[ExecutorStageSummaryWrapper],
  classOf[ApplicationInfoWrapper],
  classOf[PoolData],
  classOf[ExecutorSummaryWrapper],
  classOf[StageDataWrapper],
  classOf[AppSummary],
  classOf[RDDOperationGraphWrapper],
  classOf[TaskDataWrapper],
  classOf[ApplicationEnvironmentInfoWrapper],
  Utils.classForName(SparkPlanGraphWrapper),
  Utils.classForName(SQLExecutionUIData)
)
```

UIMetaFile Persistence

of UIMetaStore

```
4-Byte Magic Number: "UI_S"
---------- Body ---------------
4_byte_length_of_class_name | class_name_str1 | 4_byte_length | serialized_of_class1_instance1
4_byte_length_of_class_name | class_name_str1 | 4_byte_length | serialized_of_class1_instance2
4_byte_length_of_class_name | class_name_str2 | 4_byte_length | serialized_of_class2_instance1
4_byte_length_of_class_name | class_name_str2 | 4_byte_length | serialized_of_class2_instance2
```

EventLoggingListener vs UIMetaLoggingListener

EventLoggingListener

EventLoggingListener triggers a serialized writing every time it accepts an event.

EventLoggingListener is streamed.

UIMetaLoggingListener

UIMetaLoggingListener is only triggered by some specific events such as “StageEnd” and “JobEnd”, and each write operation is batched Write, the information of the UIMetaStore in the previous stage is completely persisted.

UIMetaLoggingListener is batched, and periodically snapshots the UI state.

FsHistoryProvider vs UIMetaProvider

FsHistoryProvider

Read event log files and replay them to generate KVStore.

List all application paths to build the app list. All meta information should be loaded to memory.

UIMetaProvider

Read UIMetaFiles and deserialize them to build UIMetaStore.

According to the appId from access link, directly parses an UIMetaFile. Easy to scale-out.

Other Optimizations

Write redundancy

- Each Stage Completion event will trigger a writing of the UIMeta file. To eliminate the write redundancy, UIMeta maintains a map inside UIMetaLoggingListener to record instances that have been serialized.
- TaskDataWrapper data is largest, so only the TaskDataWrapper data whose status is Completed at the end of a stage will be persisted.

Other Optimizations

Fallback Eventlog

- Supports falling back to read the event log file when UIMeta file does not exist or an error is thrown in parsing UIMeta file.
- Supports converting event log files into UIMeta files offline.

Results

Storage Usage

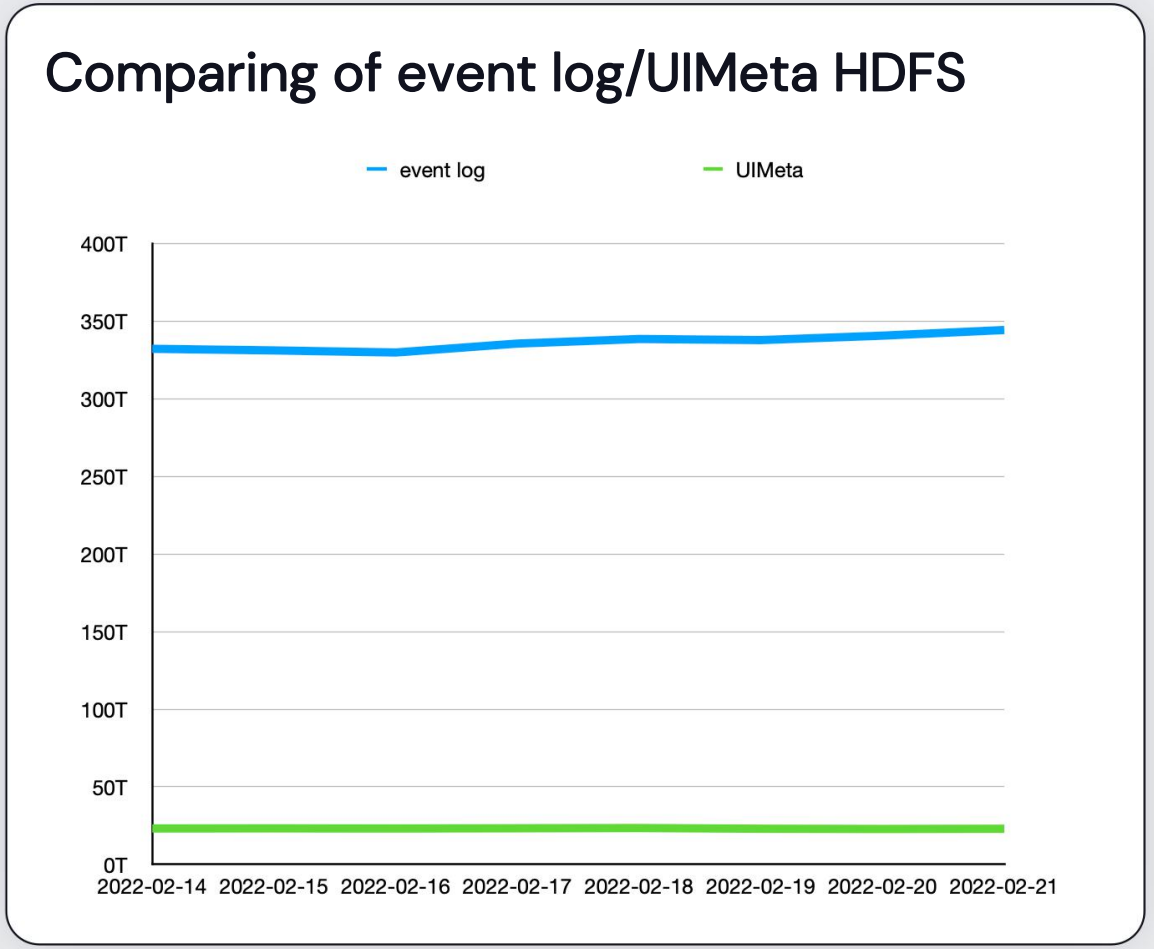
Before & After for a certain IDC

Storage was reduced by an average of 85%。

Total volume was reduced by 92.4%.

At present, the 7-day event log in ByteDance used 3.2 PB.

After switching to UIMeta, the space is only 350TB.



Access Latency

35% reduction on average.

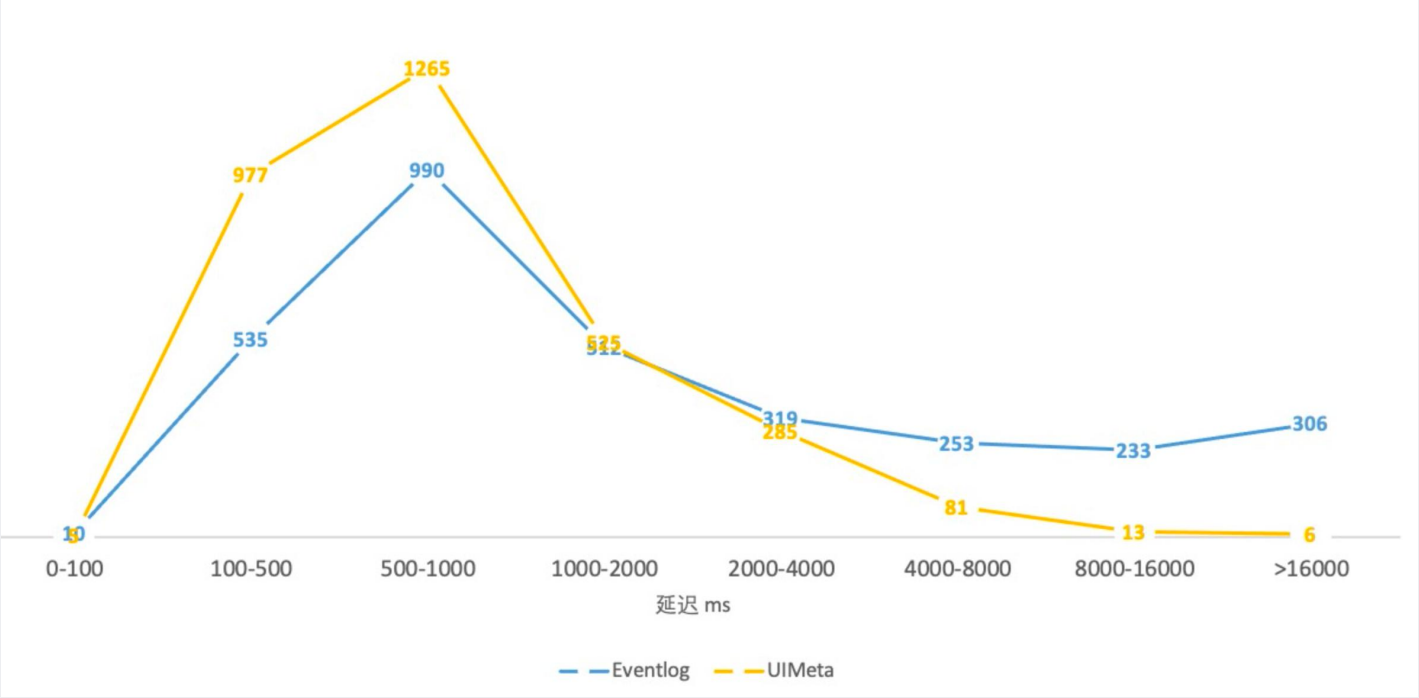
84.6%/90.8%/93.7% reduction in PCT90/95/99 respectively

	pct90	pct95	pct99	AVG
event log	15589ms	37022ms	104259ms	7217ms
UIMeta	2401ms	3410ms	6595ms	1108ms
Decline %	84.6%	90.8%	93.7%	84.6%

Access Latency

Access Latency Distribution Graph

As shown in the figure below, the overall UI access latency of UIMeta is shifted to the left compared to the event log, and the long-tail tasks are significantly reduced.



End to End Profermance

The UIMeta eliminetes traversing app paths and preloading, the duration between application comletion and accessing in the new History Server has been reduced from 10 minutes to several seconds.

Stateless and Resilient

UIMeta does not need to pre-load the event-log files directory, the loading of snapshot file is totally on demand. **It is stateless and can be scale-out.** On public Cloud, UIMeta servers are resilient according to the access traffic via load balance.

Isolation on Cloud env

Finally, it’s easy to address the requirement of multi-tenant isolation by adding corresponding tokens to the access requests in public Cloud under current framework.

Conclusions

The new Spark history server UIMeta aims at displaying the Spark historic information in a scaleable, economical, cloud-native way. The results above show the UIMeta can highly save storage, increase the access speed and improve the user experience.

At present, UIMeta has been the default history service of LAS. And you can try it in https://www.volcengine.com/product/las

DATA+AI
SUMMIT 2022

Thank you

Lantao Jin

Head of Spark Engine Team, ByteDance