

# 从 Spark 做批处理到 Flink 做流批一体

原创 张晨娅@LinkedIn Apache Flink 2021-10-12 21:30

收录于话题

#行业案例

57个 >

▼ 关注「**Flink 中文社区**」，获取更多技术干货 ▼



**Apache Flink**

原「Flink 中文社区」。Apache Flink 官微，Flink PMC 维护  
263篇原创内容



公众号

**摘要：**本文由社区志愿者苗文婷整理，内容来源于 LinkedIn 大数据高级开发工程师张晨娅在 Flink Forward Asia 2020 分享的《从 Spark 做批处理到 Flink 做流批一体》，主要内容为：

1. 为什么要做流批一体？
2. 当前行业已有的解决方案和现状，优势和劣势
3. 探索生产实践场景的经验
4. Shuffle Service 在 Spark 和 Flink 上的对比，以及 Flink 社区后面可以考虑做的工作
5. 总结

**Tips：**FFA 2021 重磅开启，点击「**阅读原文**」即可报名～

 GitHub 地址 

欢迎大家给 Flink 点赞送 star～



## 一、为什么要做流批一体

## Benefits of Stream-Batch Unification to BI / AI / ETL Users

### Shared Code

Avoid code duplication  
and enable reuse of core  
processing logic

### Two Directions

Streaming first → Batch  
OR  
Batch first → Streaming

### Maintain Less

Avoid operating on  
different systems or  
mitigate inconsistency

### Learn Once

Avoid ramping up on  
different frameworks with  
multiple learning curves

FLINK FORWARD #ASIA 2020

做流批一体到底有哪些益处，尤其是在 BI/AI/ETL 的场景下。整体来看，如果能帮助用户做到流批一体，会有以上 4 个比较明显的益处：

- 可以避免代码重复，复用代码核心处理逻辑

代码逻辑能完全一致是最好的，但这会有一些的难度。但整体来讲，现在的商业逻辑越来越长，越来越复杂，要求也很多，如果我们使用不同的框架，不同的引擎，用户每次都要重新写一遍逻辑，压力很大并且难以维护。所以整体来讲，尽量避免代码重复，帮助用户复用代码逻辑，就显得尤为重要。

- 流批一体有两个方向

这两个方向要考虑的问题很不一样，目前 Flink 做 Streaming、Spark 做 Batch 等等一些框架在批处理或流处理上都比较成熟，都已经产生了很多的单方面用户。当我们想帮助用户移到另外一个方向上时，比如一些商业需求，通常会分成两类，是先从流处理开始到批处理，还是从批处理开始到流处理。之后介绍的两个生产实践场景案例，正好对应这两个方向。

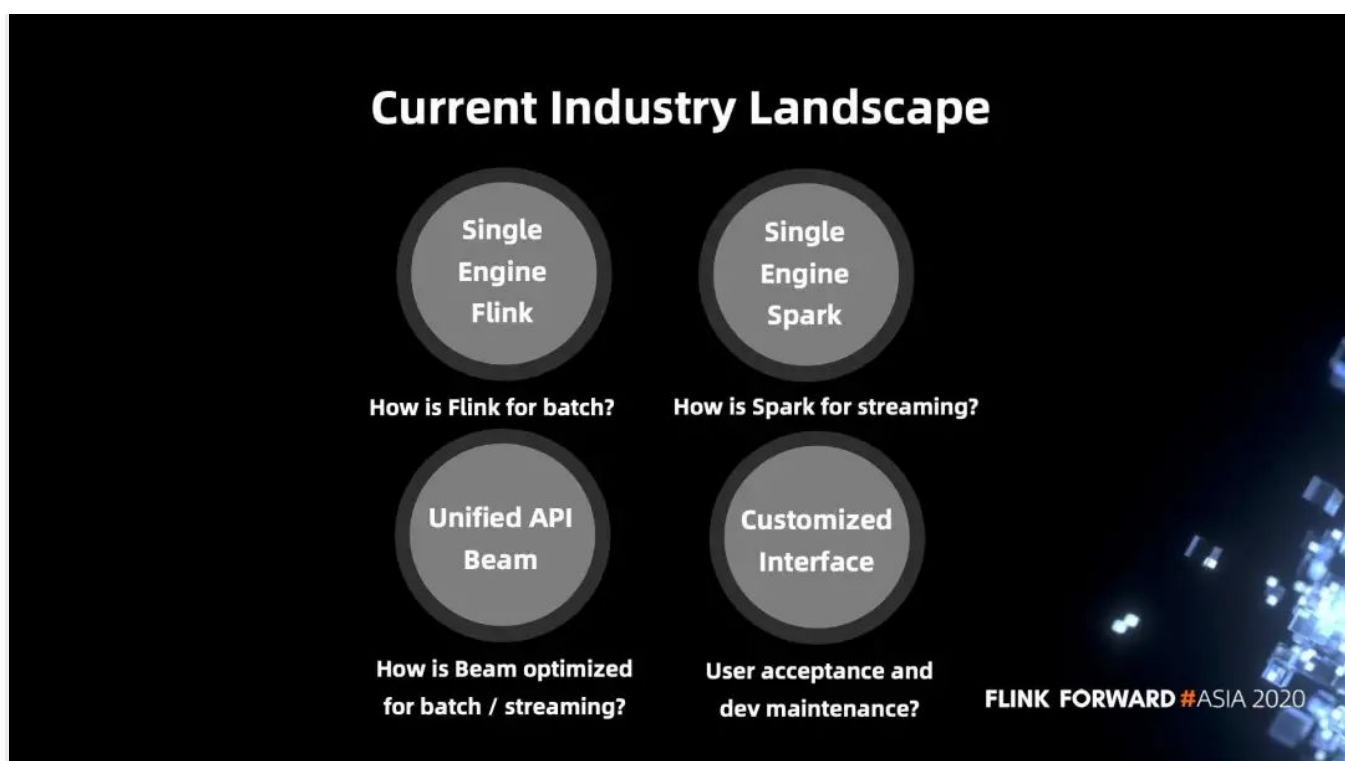
- 减少维护工作量

避免维护多套系统，系统之间的差异可能非常大，框架和引擎都不一样，会带来比较多的问题。如果公司内部有多条 pipeline，一个实时一个离线，会造成数据不一致性，因此会在数据验证、数据准确性查询、数据存储等方面做很多工作，尽量去维护数据的一致性。

- 学习更多

框架和引擎很多，商业逻辑既要跑实时，也要跑离线，所以，支持用户时需要学习很多东西。

## 二、当前行业现状



Flink 和 Spark 都是同时支持流处理和批处理的引擎。我们一致认为 Flink 的流处理做的比较好，那么它的批处理能做到多好？同时，Spark 的批处理做的比较好，那么它的流处理能不能足够帮助用户解决现有的需求？

现在有各种各样的引擎框架，能不能在它们之上有一个统一的框架，类似于联邦处理或者是一些简单的 physical API，比如 Beam API 或者是自定义接口。

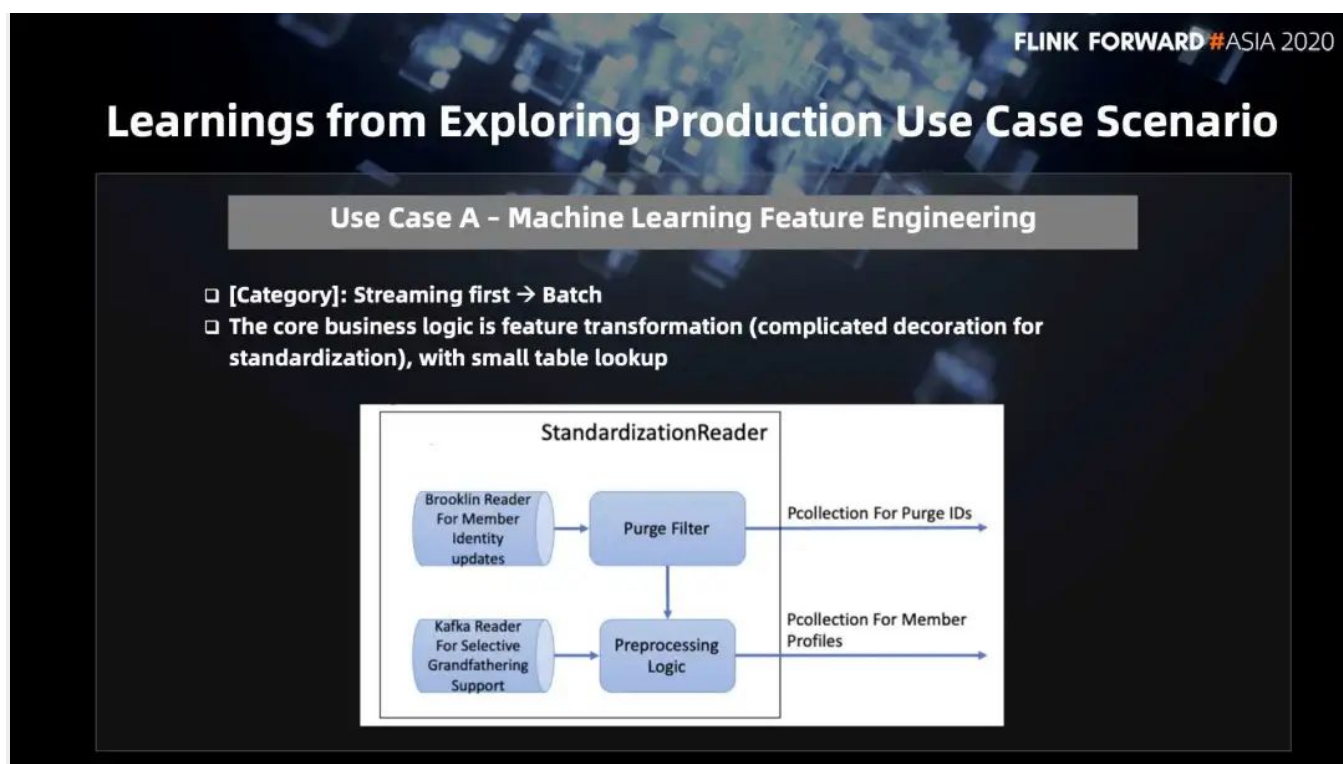
Beam 方面需要考虑的问题，是它在批处理和流处理上的优化能做到多好？Beam 目前还是偏物理执行，之后的计划是我们需要考究的。

LinkedIn，包括其他公司，会考虑做一些自定义接口的解决方案，考虑有一个共通的 SQL 层，通用的 SQL 或 API 层，底下跑不同的框架引擎。这里需要考虑的问题是，像 Spark、Flink 都是比较成熟的框架了，已经拥有大量的用户群体。当我们提出一个新的 API，一个新的解决方案，用户的接受度如何？在公司内部应该如何维护一套新的解决方案？

### 三、生产案例场景

后面内容主要聚焦在 Flink 做 batch 的效果，Flink 和 Spark 的简单对比，以及 LinkedIn 内部的一些解决方案。分享两个生产上的实例场景，一个是在机器学习特征工程生成时如何做流批一体，另一个是复杂的 ETL 数据流中如何做流批一体。

#### 3.1 案例 A - 机器学习特征工程



第一类方向，流处理 -> 批处理，归类为流批一体。

案例 A 的主体逻辑是在机器学习中做特征生成时，如何从流处理到批处理的流批一体。核心的业务逻辑就是特征转换，转化的过程和逻辑比较复杂，用它做一些标准化。

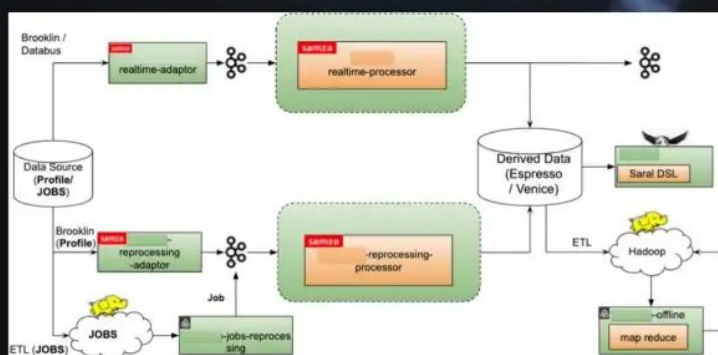
比如在 LinkedIn 的页面上输入的一些会员信息背景等，需要将这些信息提取出来标准化掉，才能进行一些推荐，帮你找一些工作等等。当会员的身份信息有更新时，会有过滤、预处理的逻辑、包括读

取 Kafka 的过程，做特征转换的过程中，可能会有一些小表查询。这个逻辑是非常直接的，没有复杂的 join 操作及其他的数据处理过程。

## Learnings from Exploring Production Use Case Scenario

### Use Case A - Machine Learning Feature Engineering

- ❑ The original workflow uses Beam on Samza for both nearline processing and backfill. Pressure on nearline cluster during backfill is very high
- ❑ Users are familiar with both Beam API and Spark Dataset API



以前它的 pipeline 是实时的，需要定期从离线 pipeline 中读取补充信息来更新流。这种 backfill 对实时集群的压力是很大的，在 backfill 时，需要等待 backfill 工作起来，需要监控工作流不让实时集群宕掉。所以，用户提出能不能做离线的 backfill，不想通过实时流处理做 backfill。

当前我们的用户是使用 Beam on Samza 做流处理，他们非常熟悉 Beam API 和 Spark Dataset API，也会用 Dataset API 去做除了 backfill 之外的一些其他业务处理。

需要特别强调的是，Dataset API 很多都是直接对 Object 操作，对 type 安全性要求很高，如果建议这些用户直接改成 SQL 或者 DataFrame 等 workflow 是不切实际的，因为他们已有的业务逻辑都是对 Object 进行直接操作和转化等。



# Learnings from Exploring Production Use Case Scenario

## Use Case A - Machine Learning Feature Engineering

Imperative API	Flink DataStream	Spark Dataset	Beam on Spark
<b>Current Status</b>	Unified, full support, not optimized (Flink DataSet)	Unified, partial support, optimized (Extension from Spark DataFrame)	Unified, partial support, not optimized (Spark RDD)
<b>Related Work</b>	<a href="#">FLIP-131: Consolidate the user-facing Dataflow SDKs/APIs (and deprecate the DataSet API)</a> <a href="#">FLIP-134: Batch execution for the DataStream API</a>	<a href="#">Databricks: Introducing Apache Spark Datasets</a> <a href="#">Spark Structured Streaming Programming Guide: Unsupported Operations</a>	<a href="#">Beam Documentation - Using the Apache Spark Runner</a> <a href="#">BEAM-8470 Create a new Spark runner based on Spark Structured streaming framework</a>

在这个案例下，我们能提供给用户一些方案选择，Imperative API。看下业界提供的方案：

- 第一个选择是即将要统一化的 Flink DataStream API，此前我们在做方案评估时也有调研 Flink DataSet API (deprecated)，DataStream API 可以做到统一，并且在流处理和批处理方面的支持都是比较完善的。但缺点是，毕竟是 Imperative API，可能没有较多的优化，后续应该会持续优化。可以看下 FLIP-131: Consolidate the user-facing Dataflow SDKs/APIs (and deprecate the DataSet API) [1] 和 FLIP-134: Batch execution for the DataStream API [2]。
- 第二个选择是 Spark Dataset，也是用户一个比较自然的选择。可以用 Dataset API 做 Streaming，区别于 Flink 的 Dataset、DataStream API 等物理 API，它是基于 Spark Dataframe SQL engine 做一些 type safety，优化程度相对好一些。可以看下文章 Databricks: Introducing Apache Spark Datasets [3] 和 Spark Structured Streaming Programming Guide: Unsupported-operations [4]。
- 第三个选择是 Beam On Spark，它目前主要还是用 RDD runner，目前支持带 optimization 的 runner 还是有一定难度的。之后会详细说下 Beam 在案例 B 中的一些 ongoing 的工作。可以看看 Beam Documentation - Using the Apache Spark Runner [5] 和 BEAM-8470 Create a new Spark runner based on Spark Structured streaming framework [6]。

# Learnings from Exploring Production Use Case Scenario

## Use Case A - Machine Learning Feature Engineering

- Flink DataStream (DataSet) API has very close user interface as Spark Dataset API
- Both Flink and Spark API are very different from Beam API. Beam on Spark users can depend on lightweight converters to reuse existing business logic for Flink / Spark
- The performance of Flink DataSet and Spark Dataset is very close for this use case

从用户的反馈上来说，Flink 的 DataStream (DataSet) API 和 Spark 的 Dataset API 在用户 interface 上是非常接近的。作为 Infra 工程师来说，想要帮用户解决问题，对 API 的熟悉程度就比较重要了。

但是 Beam 和 Flink、Spark 的 API 是非常不一样的，它是 Google 的一条生态系统，我们之前也帮助用户解决了一些问题，他们的 workflow 是在 Beam on Samza 上，他们用 p collections 或者 p transformation 写了一些业务逻辑，output、input 方法的 signature 都很不一样，我们开发了一些轻量级 converter 帮助用户复用已有的业务逻辑，能够更好的用在重新写的 Flink 或 Spark 作业里。

从 DAG 上来看，案例 A 是一个非常简单的业务流程，就是简单直接的对 Object 进行转换。Flink 和 Spark 在这个案例下，性能表现上是非常接近的。

# Learnings from Exploring Production Use Case Scenario

## Use Case A - Machine Learning Feature Engineering

- ❑ Flink users check Flink Dashboard UI for exceptions or query history server for past job metrics



### Available Requests

Below is a list of available requests, with a sample JSON response. All requests are of the sample form `http://hostname:8082/jobs`. Below we list only the path part of the URLs.

Values in angle brackets are variables, for example `http://hostname:port/jobs/<jobid>/exceptions` will have to be requested for example as `http://hostname:port/jobs/7584be6904e4e955c2a58a80c4631f5/exceptions`.

- `/conf`
- `/jobs/overview`
- `/jobs/<jobid>`
- `/jobs/<jobid>/vertices`
- `/jobs/<jobid>/conf`
- `/jobs/<jobid>/exceptions`
- `/jobs/<jobid>/accumulators`
- `/jobs/<jobid>/vertices/<vertexid>`
- `/jobs/<jobid>/vertices/<vertexid>/subtasks`
- `/jobs/<jobid>/vertices/<vertexid>/taskmanagers`
- `/jobs/<jobid>/vertices/<vertexid>/accumulators`
- `/jobs/<jobid>/vertices/<vertexid>/subtasks/accumulators`
- `/jobs/<jobid>/vertices/<vertexid>/subtasks/subtaskmanagers`
- `/jobs/<jobid>/vertices/<vertexid>/subtasks/subtaskmanagers/attempt/<attempt>/accumulators`
- `/jobs/<jobid>/plan`

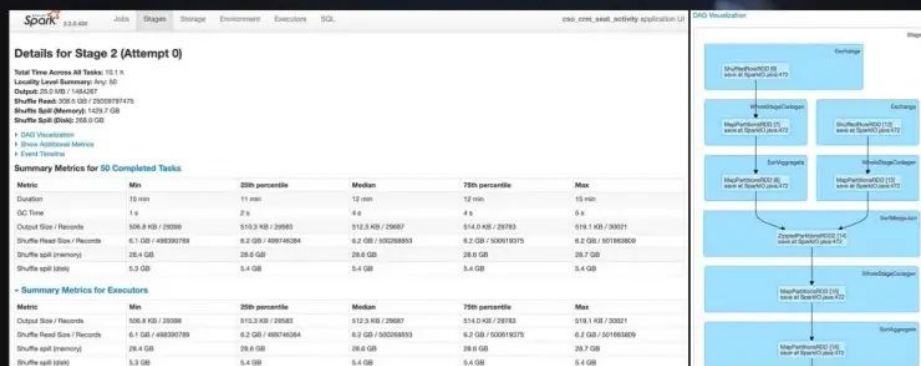
通常，我们会用 Flink Dashboard UI 看一些异常、业务流程等，相比 Spark 来说是一个比较明显的优势。Spark 去查询 Driver log，查询异常是比较麻烦的。但是 Flink 依旧有几个需要提升的地方：

- History Server - 支持更丰富的 Metrics 等

# Learnings from Exploring Production Use Case Scenario

## Use Case A - Machine Learning Feature Engineering

- ❑ Spark users rely on Spark History Server UI for both information. Flink history server may need more development (available metrics, job logs, UI, etc.)



Spark History Server UI 呈现的 metrics 比较丰富的，对用户做性能分析的帮助是比较大的。Flink 做批处理的地方是否也能让 Spark 用户能看到同等的 metrics 信息量，来降低用户的开发难度，提



高用户的开发效率。

- 更好的批处理运维工具

FLINK FORWARD #ASIA 2020

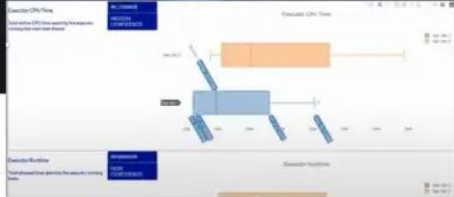
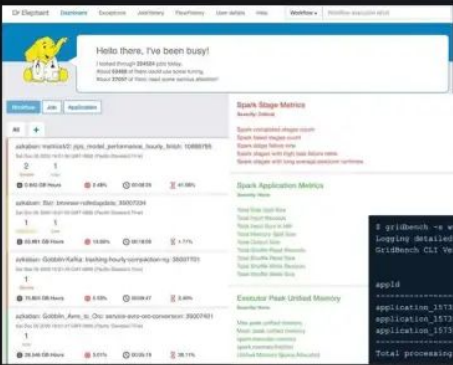
Learnings from Exploring Production Use Case Scenario

Use Case A - Machine Learning Feature Engineering

❑ Spark users rely on Dr. Elephant and GridBench to tune and understand their jobs

❑ LinkedIn Dr. Elephant GitHub Link: <https://github.com/linkedin/dr-elephant>

❑ LinkedIn GridBench Talk: [Project Optimum: Spark Performance at LinkedIn Scale](#)



```
s gridbench -e war resource -e application_1573579182113_1779735 application_1573579182113_1780682 application_1573579182113_17810322
logging detailed information to /data/ydegnai/.gridbench/gridbench-cli.log
GridBench CLI Version: 0.0.36

appid          maxHeapMemory (GB)  stored  upTime (hrs)  resource (GB-hrs)
-----
application_1573579182113_1779735  16.1328125  3  276  4467.20425
application_1573579182113_1780682  3.375  3  113  127.125
application_1573579182113_1781032  2.375  2  0  0.5
Total processing time: 12 seconds
```

分享一个 LinkedIn 从两三年前就在做的事情。LinkedIn 每天有 200000 的作业跑在集群上，需要更好的工具支持批处理用户运维自己的作业，我们提供了 Dr. Elephant 和 GridBench 来帮助用户调试和运维自己的作业。

Dr. Elephant 已开源，能帮助用户更好的调试作业，发现问题并提供建议。另外，从测试集群到生产集群之前，会根据 Dr. Elephant 生成的报告里评估结果的分数来决定是否允许投产。

GridBench 主要是做一些数据统计分析，包括 CPU 的方法热点分析等，帮助用户优化提升自己的作业。GridBench 后续也有计划开源，可以支持各种引擎框架，包括可以把 Flink 加进来，Flink job 可以用 GridBench 更好的做评估。GridBench Talk: Project Optimum: Spark Performance at LinkedIn Scale [7]。

用户不仅可以看到 GridBench 生成的报告，Dr. Elephant 生成的报告，也可以通过命令行看到 job 的一些最基本信息，应用 CPU 时间、资源消耗等，还可以对不同 Spark job 和 Flink job 之间进行对比分析。

以上就是 Flink 批处理需要提升的两块地方。

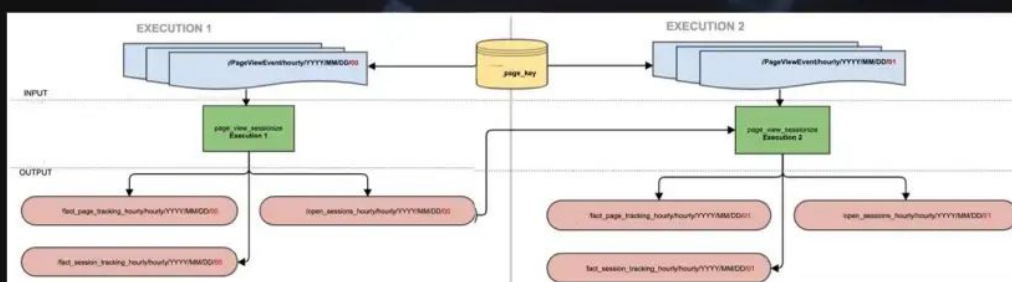
### 3.2 案例 B - 复杂的 ETL 数据流

FLINK FORWARD #ASIA 2020

## Learnings from Exploring Production Use Case Scenario

### Use Case B - Complex ETL Data Processing Workflow

- [Category]: Batch first → Streaming
- The core business logic is session window aggregation (per hourly basis) to compute aggregated page views. There are both small and large table join



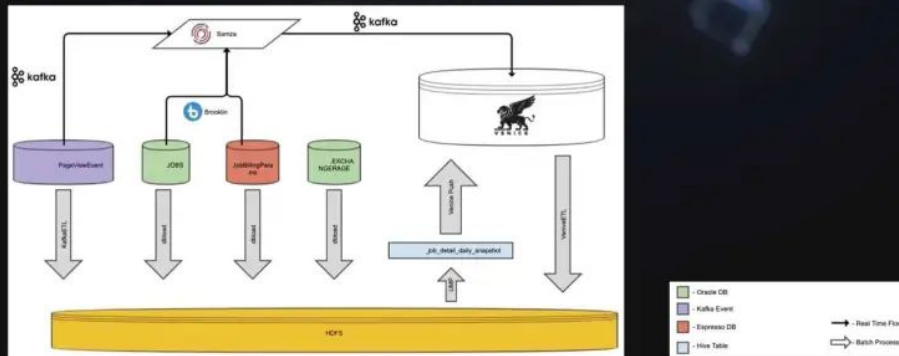
第二类方向，批处理 -> 流处理，归类为流批一体。

ETL 数据流的核心逻辑相对复杂一些，比如包括 session window 聚合窗口，每个小时计算一次页面的用户浏览量，分不同的作业，中间共享 metadata table 中的 page key，第一个作业处理 00 时间点，第二个作业处理 01 时间点，做一些 sessionize 的操作，最后输出结果，分 open session、close session，以此来增量处理每个小时的数据。

# Learnings from Exploring Production Use Case Scenario

## Use Case B - Complex ETL Data Processing Workflow

- ❑ The original workflow uses Spark SQL for offline incremental processing
- ❑ It cost users great effort building a separate Beam on Samza nearline workflow
- ❑ Many large, complicated UDFs written in both Hive and Spark
- ❑ Users are familiar with both Spark SQL and Spark DataFrame API



这个 workflow 原先是通过 Spark SQL 做的离线增量处理，是纯离线的增量处理。当用户想把作业移到线上做一些实时处理，需要重新搭建一个比如 Beam On Samza 的实时的 workflow，在搭建过程中我们和用户有非常紧密的联系和沟通，用户是遇到非常多的问题的，包括整个开发逻辑的复用，确保两条业务逻辑产生相同的结果，以及数据最终存储的地方等等，花了很长时间迁移，最终效果是不太好的。

另外，用户的作业逻辑里同时用 Hive 和 Spark 写了非常多很大很复杂的 UDF，这块迁移也是非常大的工作量。用户对 Spark SQL 和 Spark DataFrame API 是比较熟悉的。

上图中的黑色实线是实时处理的过程，灰色箭头主要是批处理的过程，相当于是一个 Lambda 的结构。

# Learnings from Exploring Production Use Case Scenario

## Use Case B - Complex ETL Data Processing Workflow

Declarative API	Flink Table API / SQL	Spark DataFrame API / SQL	Beam Schema Aware API / SQL
<b>Current Status</b>	Unified, full support, optimized	Unified, partial support, optimized	Unified, early stage
<b>Related Work</b>	<a href="#">Alibaba Cloud Blog: What's All Involved with Blink Merging with Apache Flink</a>  <a href="#">FLINK-11439 INSERT INTO flink_sql SELECT * FROM blink_sql</a>	<a href="#">Databricks Blog: Deep Dive into Spark SQL's Catalyst Optimizer</a>  <a href="#">Databricks Blog: Project Tungsten: Bringing Apache Spark Closer to Bare Metal</a>	<a href="#">Beam Design Document - Schema Aware PCollections</a>  <a href="#">Beam User Guide - Beam SQL Overview</a>

针对案例 B，作业中包括很多 join 和 session window，他们之前也是用 Spark SQL 开发作业的。很明显我 们要从 Declartive API 入手，当前提供了 3 种方案：

- 第一个选择是 Flink Table API/SQL，流处理批处理都可以做，同样的SQL，功能支持很全面，流处理和批处理也都有优化。可以看下文章 [Alibaba Cloud Blog: What's All Involved with Blink Merging with Apache Flink?](#) [8] 和 [FLINK-11439 INSERT INTO flink\\_sql SELECT \\* FROM blink\\_sql](#) [9]。
- 第二个选择是 Spark DataFrame API/SQL，也是可以用相同的 interface 做批处理和流处理，但是 Spark 的流处理支持力度还是不够的。可以看下文章 [Databricks Blog: Deep Dive into Spark SQL's Catalyst Optimizer](#) [10] 和 [Databricks Blog: Project Tungsten: Bringing Apache Spark Closer to Bare Metal](#) [11]。
- 第三个选择是 Beam Schema Aware API/SQL，Beam 更多的是物理的 API，在 Schema Aware API/SQL 上目前都在开展比较早期的工作，暂不考虑。所以，之后的主要分析结果和经验都是从 Flink Table API/SQL 和 Spark DataFrame API/SQL 的之间的对比得出来的。可以看下文章 [Beam Design Document - Schema-Aware PCollections](#) [12] 和 [Beam User Guide - Beam SQL overview](#) [13]。



# Learnings from Exploring Production Use Case Scenario

## Use Case B - Complex ETL Data Processing Workflow

- ❑ Flink Table API / SQL has close interface to Spark DataFrame / SQL with few differences in key words, rules
- ❑ Both Flink and Spark have nice integration with Hive (e.g. for Hive UDF reuse)
- ❑ Flink performs better in pipeline (not batch) mode than Spark for this use case

从用户的角度来说，Flink Table API/SQL 和 Spark DataFrame API/SQL 是非常接近的，有一些比较小的差别，比如 keywords、rules、join 具体怎么写等等，也会给用户带来一定的困扰，会怀疑自己是不是用错了。

Flink 和 Spark 都很好的集成了 Hive，比如 Hive UDF 复用等，对案例B中的 UDF 迁移，减轻了一半的迁移压力。

Flink 在 pipeline 模式下的性能是明显优于 Spark 的，可想而知，要不要落盘对性能影响肯定是比较大的，如果需要大量落盘，每个 stage 都要把数据落到磁盘上，再重新读出来，肯定是要比不落盘的 pipeline 模式的处理性能要差的。pipeline 比较适合短小的处理，在 20 分钟 40 分钟还是有比较大的优势的，如果再长的 pipeline 的容错性肯定不能和 batch 模式相比。Spark 的 batch 性能还是要比 Flink 好一些的。这一块需要根据自己公司内部的案例进行评估。



# Learnings from Exploring Production Use Case Scenario

## Use Case B - Complex ETL Data Processing Workflow

- Flink has better built-in, window-related functionality support compared to Spark

```
// Session Event-time Window
.window(Session withGap 10.minutes on $"rowtime" as $"w")

// Session Processing-time Window (assuming a processing-time attribute "proctime")
.window(Session withGap 10.minutes on $"proctime" as $"w")
```

Group Window  
Aggregate  
Batch Streaming

```
// Sliding Event-time Window
.window(Slide over 10.minutes every 5.minutes on $"rowtime" as $"w")

// Sliding Processing-time window (assuming a processing-time attribute "proctime")
.window(Slide over 10.minutes every 5.minutes on $"proctime" as $"w")

// Sliding Row-count window (assuming a processing-time attribute "proctime")
.window(Slide over 10.rows every 5.rows on $"proctime" as $"w")
```

Flink 对 window 的支持明显比其他引擎要丰富的多，比如 session window，用户用起来非常方便。我们用户为了实现 session window，特意写了非常多的 UDF，包括做增量处理，把 session 全部 build 起来，把 record 拿出来做处理等等。现在直接用 session window operator，省了大量的开发消耗。同时 group 聚合等 window 操作也都是流批同时支持的。

### Session Window:

```
1 // Session Event-time Window
2 .window(Session withGap 10.minutes on $"rowtime" as $"w")
3
4 // Session Processing-time Window (assuming a processing-time attribute "proctime")
5 .window(Session withGap 10.minutes on $"proctime" as $"w")
```

### Slide Window:

```
1 // Sliding Event-time Window
2 .window(Slide over 10.minutes every 5.minutes on $"rowtime" as $"w")
3
4 // Sliding Processing-time Window (assuming a processing-time attribute "proctime")
```

```

5 .window(Slide over 10.minutes every 5.minutes on $"proctime" as $"w")
6
7 // Sliding Row-count Window (assuming a processing-time attribute "proct:
8 .window(Slide over 10.rows every 5.rows on $"proctime" as $"w")

```

FLINK FORWARD #ASIA 2020

## Learnings from Exploring Production Use Case Scenario

### Use Case B - Complex ETL Data Processing Workflow

- How to solve the UDF migration problem by using a standard portable API ?
- LinkedIn Transport GitHub Link: <https://github.com/linkedin/transport>
- LinkedIn Engineering Blog: <https://engineering.linkedin.com/blog/2018/11/using-translatable-portable-UDFs>

```

public class MapFromTwoArraysFunction extends StdUDF2<StdArray, StdArray, StdMap> {
    private StdType _mapType;

    @Override
    public List<String> getInputParameterSignatures() {
        return ImmutableList.of(
            "array(K)",
            "array(V)"
        );
    }

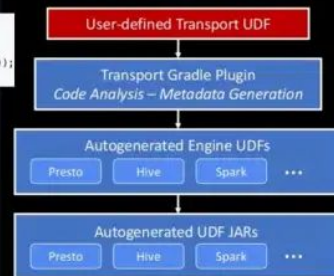
    @Override
    public String getOutputParameterSignature() {
        return "map(K,V)";
    }

    @Override
    public void init(StdFactory stdFactory) {
        super.init(stdFactory);
        _mapType = getStdFactory().createStdType(getOutputParameterSignature());
    }

    @Override
    public StdMap eval(StdArray a1, StdArray a2) {
        if (a1.size() != a2.size()) {
            return null;
        }
        StdMap map = getStdFactory().createMap(_mapType);
        for (int i = 0; i < a1.size(); i++) {
            map.put(a1.get(i), a2.get(i));
        }
        return map;
    }
}

```

### Auto-generated UDFs



UDF 是在引擎框架之间迁移时最大的障碍。如果 UDF 是用 Hive 写的，那是方便迁移的，因为不管是 Flink 还是 Spark 对 Hive UDF 的支持都是很好的，但如果 UDF 是用 Flink 或者 Spark 写的，迁移到任何一个引擎框架，都会遇到非常大的问题，比如迁移到 Presto 做 OLAP 近实时查询。

为了实现 UDF 的复用，我们 LinkedIn 在内部开发了一个 transport 项目，已经开源至 github [14] 上，可以看下 LinkedIn 发表的博客：Transport: Towards Logical Independence Using Translatable Portable UDFs [15]。

transport 给所有引擎框架提供一个面向用户的 User API，提供通用的函数开发接口，底下自动生成基于不同引擎框架的 UDF，比如 Presto、Hive、Spark、Flink 等。

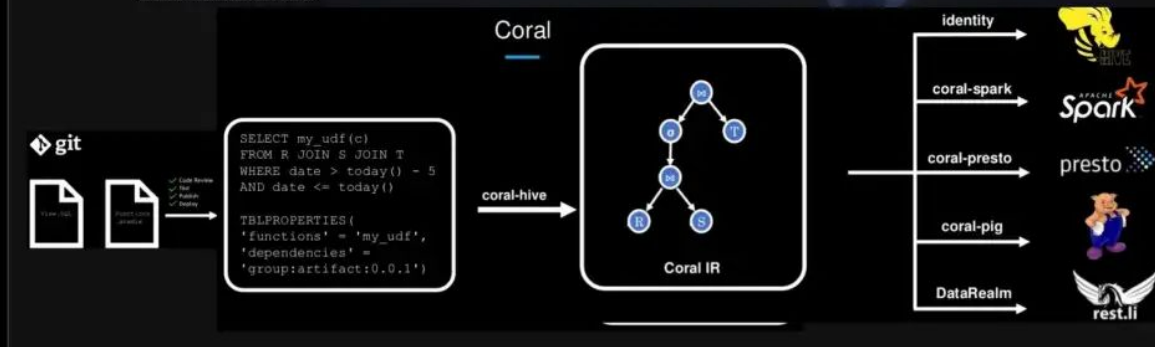
用一个共通的 UDF API 打通所有的引擎框架，能让用户复用自己的业务逻辑。用户可以很容易的上手使用，比如如下用户开发一个 MapFromTwoArraysFunction:

```
1 public class MapFromTwoArraysFunction extends StdUDF2<StdArray,StdArray,
2
3     private StdType _mapType;
4
5     @Override
6     public List<String> getInputParameterSignatures(){
7         return ImmutableList.of(
8             "array[K]",
9             "array[V]"
10        );
11    }
12
13    @Override
14    public String getOutputParameterSignature(){
15        return "map(K,V)";
16    }
17 }
18 @Override
19 public void init(StdFactory stdFactory){
20     super.init(stdFactory);
21 }
22 @Override
23 public StdMap eval(StdArray a1, StdArray a2){
24     if(a1.size() != a2.size()) {
25         return null;
26     }
27     StdMap map = getStdFactory().createMap(_mapType);
28     for(int i = 0; i < a1.size; i++) {
29         map.put(a1.get(i), a2.get(i));
30     }
31     return map;
32 }
```

# Learnings from Exploring Production Use Case Scenario

## Use Case B - Complex ETL Data Processing Workflow

- How to solve the SQL migration problem with a federated layer across multiple engines?
- LinkedIn Coral GitHub Link: <https://github.com/linkedin/coral>
- LinkedIn Coral Tech Talk @Facebook: [Coral & Transport UDFs: Building Blocks of a Postmodern Data Warehouse](#)



处理用户的 SQL 迁移问题，用户之前是用 Spark SQL 开发的作业，之后想使用流批一体，改成 Flink SQL。目前的引擎框架还是比较多的，LinkedIn 开发出一个 coral 的解决方案，已在 github [16] 上开源，在 facebook 上也做了一些 talk，包括和 transport UDF 一起给用户提供一个隔离层使用户可以更好的做到跨引擎的迁移，复用自己的业务逻辑。

看下 coral 的执行流程，首先作业脚本中定义熟悉的 ASCII SQL 和 table 的属性等，之后会生成一个 Coral IR 树状结构，最后翻译成各个引擎的 physical plan。

# Learnings from Exploring Production Use Case Scenario

## Use Case B - Complex ETL Data Processing Workflow

- How about using Flink for batch (not Spark for batch) at a very large scale?

□ **SHUFFLE?!**



在案例 B 分析中，流批统一，在集群业务量特别大的情况下，用户对批处理的性能、稳定性、成功率等是非常重视的。其中 Shuffle Service，对批处理性能影响比较大。

#### 四、Shuffle Service 在 Spark 和 Flink 上的对比

FLINK FORWARD #ASIA 2020

### Comparison on Spark and Flink Shuffle Service

Category	Spark Shuffle Service	Flink Shuffle Service
In-memory Shuffle	Fast, but not scalable	In-memory shuffle (from streaming)
Hash-based Shuffle	Better at fault-tolerance, still not scalable	Hash-based shuffle (early work)
Sort-based Shuffle	More scalable for large shuffles	FLIP-148: Introduce Sort-Merge Based Blocking Shuffle to Flink (recently merged)
External Shuffle Service	Shuffle performance and dependency isolation	FLINK-11805 A Common External Shuffle Service Framework (open)
Disaggregate Shuffle	Address the need of compute / storage separation in cloud	FLINK-10653 Introduce Pluggable Shuffle Service Architecture (recently merged)

In-memory Shuffle，Spark 和 Flink 都支持，比较快，但不支持可扩展。

Hash-based Shuffle，Spark 和 Flink 都支持，相比 In-memory Shuffle，容错性支持的更好一些，但同样不支持可扩展。

Sort-based Shuffle，对大的 Shuffle 支持可扩展，从磁盘读上来一点一点 Sort match 好再读回去，在 FLIP-148: Introduce Sort-Based Blocking Shuffle to Flink [17] 中也已经支持。

External Shuffle Service，在集群非常繁忙，比如在做动态资源调度时，外挂服务就会非常重要，对 Shuffle 的性能和资源依赖有更好的隔离，隔离之后就可以更好的去调度资源。FLINK-11805 A Common External Shuffle Service Framework [18] 目前处于 reopen 状态。



Disaggregate Shuffle，大数据领域都倡导 Cloud Native 云原生，计算存储分离在 Shuffle Service 的设计上也是要考虑的。FLINK-10653 Introduce Pluggable Shuffle Service Architecture [19] 引入了可插拔的 Shuffle Service 架构。

FLINK FORWARD #ASIA 2020

## Comparison on Spark and Flink Shuffle Service

### Advancement: Spark Push-based Shuffle Service

- Significantly improve disk I/O efficiency:  
Switch from small random reads to large sequential reads when accessing shuffle data in shuffle files
- LinkedIn Engineering Blog: <https://engineering.linkedin.com/blog/2020/introducing-magnet>

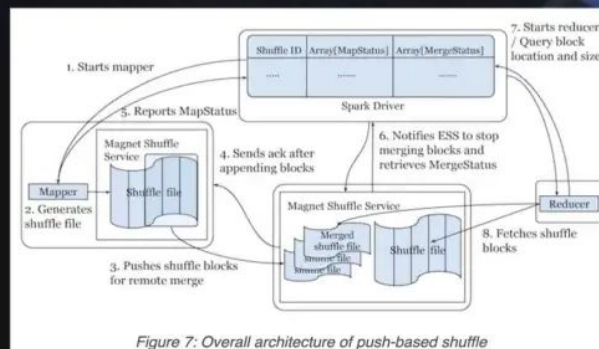


Figure 7: Overall architecture of push-based shuffle

Spark 对 Shuffle Service 做了一个比较大的提升，这个工作也是由 LinkedIn 主导的 magnet 项目，形成了一篇名称为 introducing-magnet 的论文 (Magnet: A scalable and performant shuffle architecture for Apache Spark) [20]，收录到了 LinkedIn blog 2020 里。magnet 很明显的提升了磁盘读写的效率，从比较小的 random range，到比较大的顺序读，也会做一些 merging，而不是随意的随机读取 shuffle data，避免 random IO 的一些问题。

# Comparison on Spark and Flink Shuffle Service

## Advancement: Spark Push-based Shuffle Service

- Mitigate shuffle reliability/scalability issues:
  - 1) Adopt a best-effort approach for pushing blocks
  - 2) Generate a second replica of shuffle intermediate data
- Handle stragglers during block push process via early termination techniques

	Total shuffle fetch wait time (min)	Total executor task runtime (min)	End-to-end job runtime (min)
Vanilla Spark shuffle	20636	50771	42
Magnet shuffle	445 (-98%)	29928 (-41%)	31 (-26%)

通过 Magnet Shuffle Service 缓解了 Shuffle 稳定性和可扩展性方面的问题。在此之前，我们发现了很多 Shuffle 方面的问题，比如 Job failure 等等非常高。如果想用 Flink 做批处理，帮助到以前用 Spark 做批处理的用户，在 Shuffle 上确实要花更大功夫。

- 在 Shuffle 可用性上，会采用 best-effort 方式去推 shuffle blocks，忽略一些大的 block，保证最终的一致性和准确性；
- 为 shuffle 临时数据生成一个副本，确保准确性。

如果 push 过程特别慢，会有提前终止技术。

Magnet Shuffle 相比 Vanilla Shuffle，读取 Shuffle data 的等待时间缩减了几乎 100%，task 执行时间缩短了几近 50%，端到端的任务时长也缩短了几近 30%。

	Total shuffle fetch wait time (min)	Total executor task runtime (min)	End-to-end job runtime (min)
Vanilla Spark shuffle	20636	50771	42
Magnet shuffle	445 (-98%)	29928 (-41%)	31 (-26%)

## 五、总结

## Key Take-aways

- ❑ Flink has strong advantages in doing both streaming and batch in an increasingly optimized, unified way
- ❑ Flink may spend more time on improving batch user experience (e.g. history server, metrics, tuning, user community, operation flow)
- ❑ Flink may invest more into its shuffle service and mechanism for very large-scale batch workflows
- ❑ Support and federate multiple frameworks and engines is not an easy problem for all of us

- LinkedIn 非常认可和开心看到 Flink 在流处理和批处理上的明显优势，做的更加统一，也在持续优化中。
- Flink 批处理能力有待提升，如 history server, metrics, 调试。用户在开发的时候，需要从用户社区看一些解决方案，整个生态要搭建起来，用户才能方便的用起来。
- Flink 需要对 shuffle service 和大集群离线 workflow 投入更多的精力，确保 workflow 的成功率，如果规模大起来之后，如何提供更好的用户支持和对集群进行健康监控等。
- 随着越来越多的框架引擎出现，最好能给到用户一个更加统一的 interface，这一块挑战是比较大的，包括开发和运维方面，根据 LinkedIn 的经验，还是看到了很多问题的，并不是通过一个单一的解决方案，就能囊括所有的用户使用场景，哪怕是一些 function 或者 expression，也很难完全覆盖到。像 coral、transport UDF。

原视频：

<https://www.bilibili.com/video/BV13a4y1H7XY?p=12>

### 参考链接

[1] <https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=158866741>

[2] <https://cwiki.apache.org/confluence/display/FLINK/FLIP-134%3A+Batch+execution+for+the+DataStream+API>

- [3] <https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html>
- [4] <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#unsupported-operations>
- [5] <https://beam.apache.org/documentation/runners/spark/>
- [6] <https://issues.apache.org/jira/browse/BEAM-8470>
- [7] <https://www.youtube.com/watch?v=D47CSeGpBd0>
- [8] [https://www.alibabacloud.com/blog/whats-all-involved-with-blink-merging-with-apache-flink\\_595401](https://www.alibabacloud.com/blog/whats-all-involved-with-blink-merging-with-apache-flink_595401)
- [9] <https://issues.apache.org/jira/browse/FLINK-11439>
- [10] <https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>
- [11] <https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>
- [12] <https://docs.google.com/document/d/1tnG2DPHZYbsomvihlpXruUmQ12pHGK0QlvXS1FOTgRc/edit#heading=h.puuotbien1gf>
- [13] <https://beam.apache.org/documentation/dsls/sql/overview/>
- [14] <https://github.com/linkedin/transport>
- [15] <https://engineering.linkedin.com/blog/2018/11/using-translatable-portable-UDFs>
- [16] <https://github.com/linkedin/coral>
- [17] <https://cwiki.apache.org/confluence/display/FLINK/FLIP-148%3A+Introduce+Sort-Based+Blocking+Shuffle+to+Flink>
- [18] <https://issues.apache.org/jira/browse/FLINK-11805>
- [19] <https://issues.apache.org/jira/browse/FLINK-10653>
- [20] <https://engineering.linkedin.com/blog/2020/introducing-magnet>



**报名现已开放**

Flink Forward Asia 2021 重磅启动！FFA 2021 将于 12 月 4-5 日在北京·国家会议中心举办，预计将有 3000+ 开发者参与，探讨交流 Flink 最新动态。报名通道已开启，扫描下图二维码，或点击文末「[阅读原文](#)」即可报名 FFA 2021～



更多 Flink 相关技术问题，可扫码加入社区钉钉交流群～



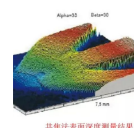
戳我，报名 FFA 2021 大会！

[阅读原文](#)

喜欢此内容的人还喜欢

从平面到3D-视觉检测的重要方向

机器视觉课堂



惊呆了！这样可以将Numpy加速700倍！

小白学视觉





## 使用深度学习和OpenCV的早期火灾检测系统

小白学视觉

