深度解读：Flink 1.11 SQL 流批一体的增强与完善

李劲松（之信） 蔡芳芳
发布于：2020 年 7 月 15 日 14:00

7 月 6 日，Apache Flink 1.11 正式发布。从 3 月初进行功能规划到 7 月初正式发版，1.11 用将近 4 个月的时间重点优化了 Flink 的易用性问题，提升用户的生产使用体验。

SQL 作为 Flink 中公认的核心模块之一，对推动 Flink 流批一体功能的完善至关重要。在 1.11 中，Flink SQL 也进行了大量的增强与完善，开发大功能 10 余项，不仅扩大了应用场景，还简化了流程，上手操作更简单。

其中，值得注意的改动包括：

- 默认 Planner 已经切到 Blink planner 上。
- 引入了对 CDC（Change Data Capture，变动数据捕获）的支持，用户仅用几句简单的 SQL 即可对接 Debezium 和 Canal 的数据源。
- 离线数仓实时化，用户可方便地使用 SQL 将流式数据从 Kafka 写入 Hive 等。

Flink SQL 演变

随着流计算的发展，挑战不再仅限于数据量和计算量，业务变得越来越复杂，开发者可能是资深的大数据从业者、初学 Java 的爱好者，或是不懂代码的数据分析者。如何提高开发者的效率，降低流计算的门槛，对推广实时计算非常重要。

SQL 是数据处理中使用最广泛的语言，它允许用户简明扼要地展示其业务逻辑。Flink 作为流批一体的计算引擎，致力于提供一套 SQL 支持全部应用场景，Flink SQL 的实现也完全遵循 ANSI SQL 标准。之前，用户可能需要编写上百行业务代码，使用 SQL 后，可能只需要几行 SQL 就可以轻松搞定。

Flink SQL 的发展大概经历了以下阶段：

- Flink 1.1.0：第一次引入 SQL 模块，并且提供 TableAPI，当然，这时候的功能还非常有限。

- Flink 1.3.0：在 Streaming SQL 上支持了 Retractions，显著提高了 Streaming SQL 的易用性，使得 Flink SQL 支持了复杂的 Unbounded 聚合连接。

- Flink 1.5.0：SQL Client 的引入，标志着 Flink SQL 开始提供纯 SQL 文本。

- Flink 1.9.0：抽象了 Table 的 Planner 接口，引入了单独的 Blink Table 模块。Blink Table 模块是阿里巴巴内部的 SQL 层版本，不仅在结构上有重大变更，在功能特性上也更加强大和完善。

- Flink 1.10.0：作为第一个 Blink 基本完成 merge 的版本，修复了大量遗留的问题，并给 DDL 带来了 Watermark 的语法，也给 Batch SQL 带来了完整的 TPC-DS 支持和高效的性能。

经过了多个版本的迭代支持，SQL 模块在 Flink 中变得越来越重要，Flink 的 SQL 用户也逐渐扩大。基于 SQL 模块的 Python 接口和机器学习接口也在快速发展。毫无疑问， SQL 模块作为最常用的 API 之一和生态的集成变得越来越重要。

SQL 1.11 重要变更

Flink SQL 在原有的基础上扩展了新场景的支持：

- Flink SQL 引入了对 CDC（Change Data Capture，变动数据捕获）的支持，它使 Flink 可以方便地通过像 Debezium 这类工具来翻译和消费数据库的变动日志。

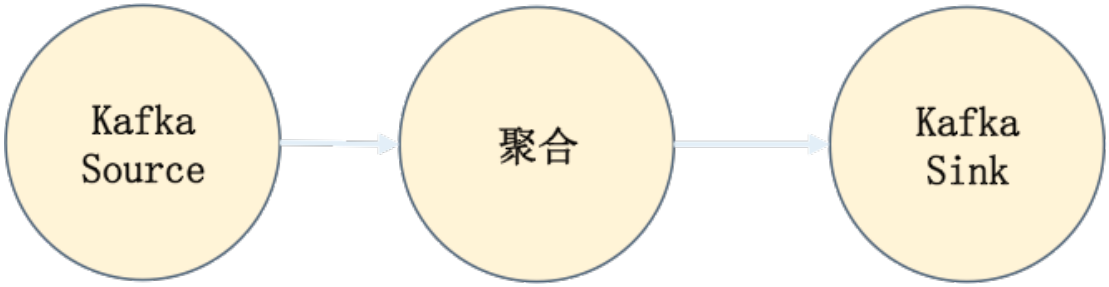
- Flink SQL 扩展了类 Filesystem connector 对实时化用户场景和格式的支持，从而可以支持将流式数据从 Kafka 写入 Hive 等场景。

除此之外，Flink SQL 也从多个方面提高 SQL 的易用性，系统性的解决了之前的 bug、完善了用户 API。

CDC 支持

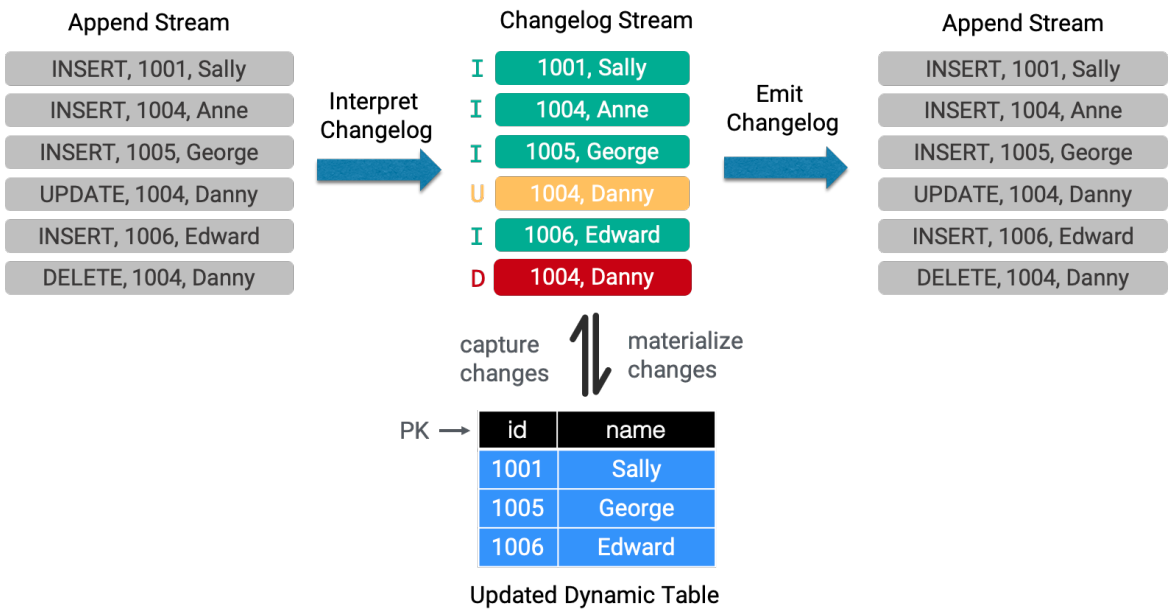
CDC 格式是数据库中一种常用的模式，业务上典型的应用是通过工具（比如 Debezium 或 Canal）将 CDC 数据通过特定的格式从数据库中导出到 Kafka 中。在以前，业务上需要定义特殊的逻辑来解析 CDC 数据，并把它转换成一般的 Insert-only 数据，后续的处理逻辑需要考虑到这种特殊性，这种 work-around 的方式无疑给业务上带来了不必要的复杂性。

如果 Flink SQL 引擎能原生支持 CDC 数据的输入，将 CDC 对接到 Flink SQL 的 Changelog Stream 概念上，将会大大降低用户业务的复杂度。



流计算的本质是就是不断更新、不断改变结果的计算。考虑一个简单的聚合 SQL，流计算中，每次计算产生的聚合值其实都是一个局部值，所以会产生 Changelog Stream。在以前想要把聚合的数据输出到 Kafka 中，如上图所示，几乎是不可能的，因为 Kafka 只能接收 Insert-only 的数据。

Flink 之前主要是因为 Source&Sink 接口的限制，导致不能支持 CDC 数据的输入。Flink SQL 1.11 经过了大量的接口重构，在新的 Source&Sink 接口上，支持了 CDC 数据的输入和输出，并且支持了 Debezium 与 Canal 格式（FLIP-105）。这一改动使动态 Table Source 不再只支持 append-only 的操作，而且可以导入外部的修改日志（插入事件）将它们翻译为对应的修改操作（插入、修改和删除）并将这些操作与操作的类型发送到后续的流中。



如上图所示，理论上，CDC 同步到 Kafka 的数据就是 Append 的一个流，只是在格式中含有 Changelog 的标识：

- 一种方式是把 Changlog 标识看做一个普通字段，这也是目前普遍的使用方式。
- 在 Flink 1.11 后，可以将它声明成 Changelog 的格式，Flink 内部机制支持 Interpret Changelog，可以原生识别出这个特殊的流，将其转换为 Flink 的 Changlog Stream，并按照 SQL 的语义处理；同理，Flink SQL 也具有输出 Change Stream 的能力（Flink 1.11 暂无内置实现），这就意味着，你可以将任意类型的 SQL 写入到 Kafka 中，只要有 Changelog 支持的 Format。

为了消费 CDC 数据，用户需要在使用 SQL DDL 创建表时指指定“format=debezium-json”或者“format=canal-json”：

```
CREATE TABLE my_table (
    ...
) WITH (
    'connector'='...', -- e.g. 'kafka'
    'format'='debezium-json'
);
```

Flink 1.11 的接口都已 Ready，但是在实现上：

- 只支持 Kafka 的 Debezium-json 和 Canal-json 读。
- 欢迎大家扩展实现自己的 Format 和 Connector。

Source & Sink 重构

Source & Sink 重构的一个重要目的是支持上节所说的 Changelog，但是除了 Changelog 以外，它也解决了诸多之前的遗留问题。

新 Source & Sink 使用标准姿势 （详见官方文档）：

```
CREATE TABLE kafka_table (
    ...
) WITH (
    'connector' = 'kafka-0.10',
    'topic' = 'test-topic',
    'scan.startup.mode' = 'earliest-offset',
    'properties.bootstrap.servers' = 'localhost:9092',
    'format' = 'json',
    'json.fail-on-missing-field' = 'false'
);
```

Flink 1.11 为了向前兼容性，依然保留了老 Source & Sink，使用 “connector.type” 的 Key，即可 Fallback 到老 Source & Sink 上。

Factory 发现机制

Flink 1.11 前，用户可能经常遇到一个异常，叫做 NoMatchingFactory 异常：

```
Caused by: org.apache.flink.table.api.NoMatchingTableFactoryException:
'org.apache.flink.table.factories.TableSourceFactory' in
the classpath.

Reason: No context matches.

The following properties are requested:
connector.properties.0.key=security.protocol
connector.properties.0.value=SSL
connector.properties.1.key=key.deserializer
```

指的是，定义了一个 DDL，在用的时候，DDL 属性找不到对应的 TableFactory 实现，可能的原因是：

- Classpath 下没有实现类，Flink SQL 是通过 Java SPI 的机制来发现 Factory；
- 参数写错了。

但是报的异常让人非常疑惑，根据异常的提示消息，很难找到到底哪里的代码错了，更难明确知道哪个 Key 写错了。

复制代码

```
public interface Factory {
    String factoryIdentifier();
    ……
}
```

所以在 Flink 1.11 中，社区重构了 TableFactory 接口，提出了一个新的 Factory 接口，它有一个方法，叫做 FactoryIdentifier。以后所有的 Factory 的 look up 都通过 identifier。这样的话就非常清晰明了，找不到是因为 Classpath 下没 Factory 的类，找得到那就可以定位到 Factory 的实现中，进行确定性的校验。

类型与数据结构

之前的 Source&Sink 接口支持用户自定义数据结构，即框架知道如何把自定义的数据结构转换为 Flink SQL 认识的内部数据结构，如：

复制代码

```
public interface TableSource<T> {
    TypeInformation<T> getReturnType();
    ...
}
```

用户可以自定义泛型 T，通过 getReturnType 来告诉框架怎么转换。

不过问题来了，当 getReturnType 和 DDL 中声明的类型不一致时怎么办？特别是两套类型系统的情况下，如：Runtime 的 TypeInformation，SQL 层的 DataType。由于精度等问题，可能导致经常出现类型不匹配的异常。

Flink 1.11 系统性地解决了这个问题。现在 Connector 开发者不能自定义数据结构，只能使用 Flink SQL 内部的数据结构：RowData。所以保证了默认 type 与 DDL 的对应，不用再返回类型让框架去确认了。

RowData 数据结构在 SQL 内部设计出来为了：

- 抽象类接口，在不同场景有适合的高性能实现。
- 包含 RowKind，契合流计算中的 CDC 数据格式。
- 遵循 SQL 规范，比如包含精度信息。
- 对应 SQL 类型的可枚举的数据结构。

SQL Data Types	Internal Data Structures
BOOLEAN	boolean
CHAR / VARCHAR / STRING	{@link StringData}
BINARY / VARBINARY / BYTES	byte[]
DECIMAL	{@link DecimalData}
TINYINT	byte
SMALLINT	short
INT	int
BIGINT	long
FLOAT	float
DOUBLE	double
DATE	int (number of days since epoch)
TIME	int (number of milliseconds of the day)
TIMESTAMP	{@link TimestampData}
TIMESTAMP WITH LOCAL TIME ZONE	{@link TimestampData}
INTERVAL YEAR TO MONTH	int (number of months)
INTERVAL DAY TO MONTH	long (number of milliseconds)
ROW / structured types	{@link RowData}
ARRAY	{@link ArrayData}
MAP / MULTISET	{@link MapData}
RAW	{@link RawValueData}

Upsert 与 Primary Key

流计算的一个典型场景是把聚合的数据写入到 Upsert Sink 中，比如 JDBC、HBase，当遇到复杂的 SQL 时，时常会出现：

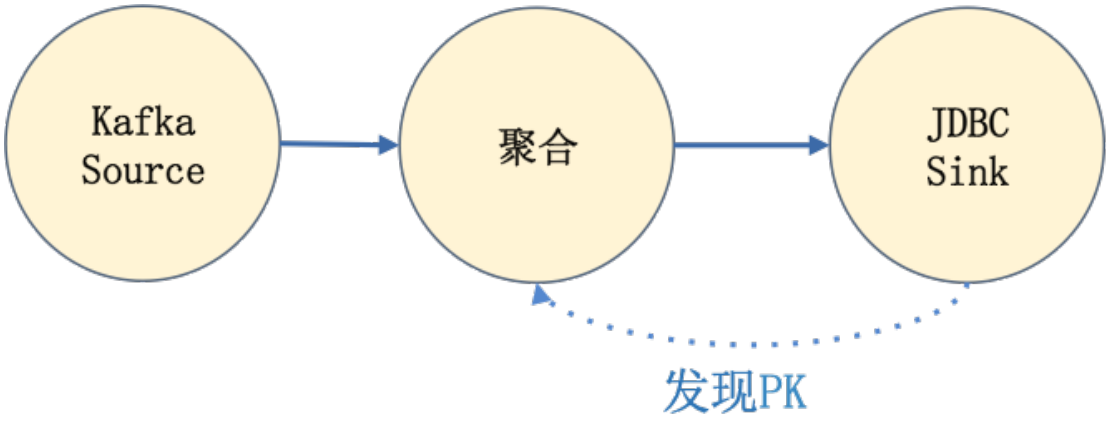
```
运行报错如下：
org.apache.flink.table.api.TableException: UpsertStreamTableSink requires that Table has a full primary keys if it is updated.

at org.apache.flink.table.planner.plan.nodes.physical.stream.StreamExecSink.translateToPlanInternal(StreamExecSink.scala:113)
at org.apache.flink.table.planner.plan.nodes.physical.stream.StreamExecSink.translateToPlanInternal(StreamExecSink.scala:48)
at org.apache.flink.table.planner.plan.nodes.exec.ExecNode$class.translateToPlan(ExecNode.scala:58)
at org.apache.flink.table.planner.plan.nodes.physical.stream.StreamExecSink.translateToPlan(StreamExecSink.scala:48)
at org.apache.flink.table.planner.delegation.StreamPlanner$$anonfun$translateToPlan$1.apply(StreamPlanner.scala:60)
at org.apache.flink.table.planner.delegation.StreamPlanner$$anonfun$translateToPlan$1.apply(StreamPlanner.scala:59)
```

UpsertStreamTableSink 需要上游的 Query 有完整的 Primary Key 信息，不然就直接抛异常。这个现象涉及到 Flink 的 UpsertStreamTableSink 机制。顾名思义，它是一个更新的 Sink，需要按 key 来更新，所以必须要有 key 信息。

如何发现 Primary Key？一个方法是让优化器从 Query 中推断，如下图发现 Primary Key 的例子。

这种情况下在简单 Query 当中很好，也满足语义，也非常自然。但是如果是一个复杂的 Query，比如聚合又 Join 再聚合，那就只有报错了。不能期待优化器有多智能，很多情况它都不能推断出 PK，而且，可能业务的 SQL 本身就不能推断出 PK，所以导致了这样的异常。



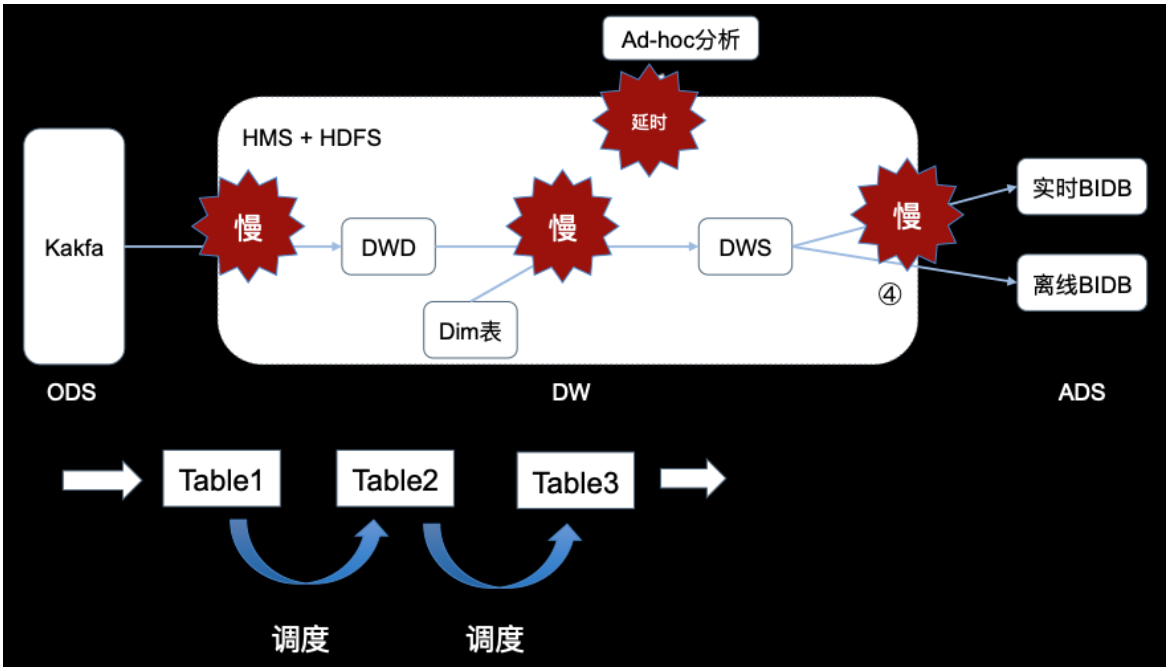
怎么解决问题？Flink 1.11 彻底的抛弃了这个机制，不再从 Query 来推断 PK 了，而是完全依赖 Create table 语法。比如 create 一个 jdbc_table，需要在定义中显式地写好 Primary Key（后面 not enforced 的意思是不强校验，因为 connector 也许没有具备 PK 的强校验的能力）。当指定了 PK，就相当于就告诉框架这个 jdbc Sink 会按照对应的 key 来进行更新。如此，就跟 query 完全没有关系了，这样的设计可以定义得非常清晰，如何更新完全按照设置的定义来。

复制代码

```
CREATE TABLE jdbc_table (
    id BIGINT,
    ...
    PRIMARY KEY (id) NOT ENFORCED
)
```

Hive 流批一体

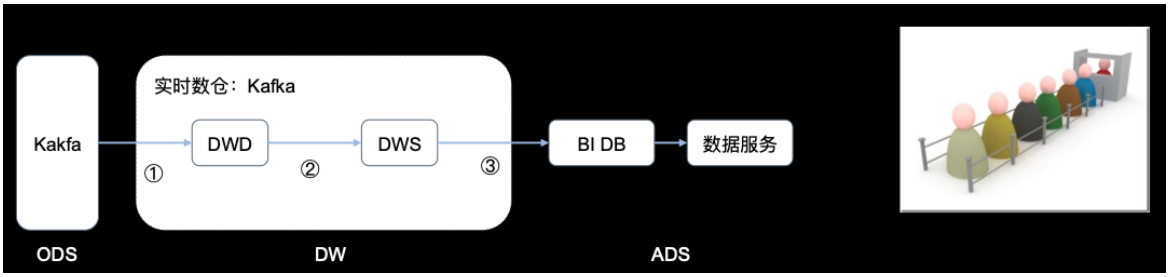
首先看传统的 Hive 数仓。一个典型的 Hive 数仓如下图所示。一般来说，ETL 使用调度工具来调度作业，比如作业每天调度一次或者每小时调度一次。这里的调度，其实也是一个叠加的延迟。调度产生 table1，再产生 table2，再调度产生 table3，计算延时需要叠加起来。



问题是慢，延迟大，并且 Ad-hoc 分析延迟也比较大，因为前面的数据入库，或者前面的调度的 ETL 会有很大的延迟。Ad-hoc 分析再快返回，看到的也是历史数据。

所以现在流行构建实时数仓，从 Kafka 读计算写入 Kafka，最后再输出到 BI DB，BI DB 提供实时的数据服务，可以实时查询。Kafka 的 ETL 为实时作业，它的延时甚至可能达到毫秒级。实时数仓依赖 Queue，它的所有数据存储都是基于 Queue 或者实时数据库，这样实时性很好，延时低。但是：

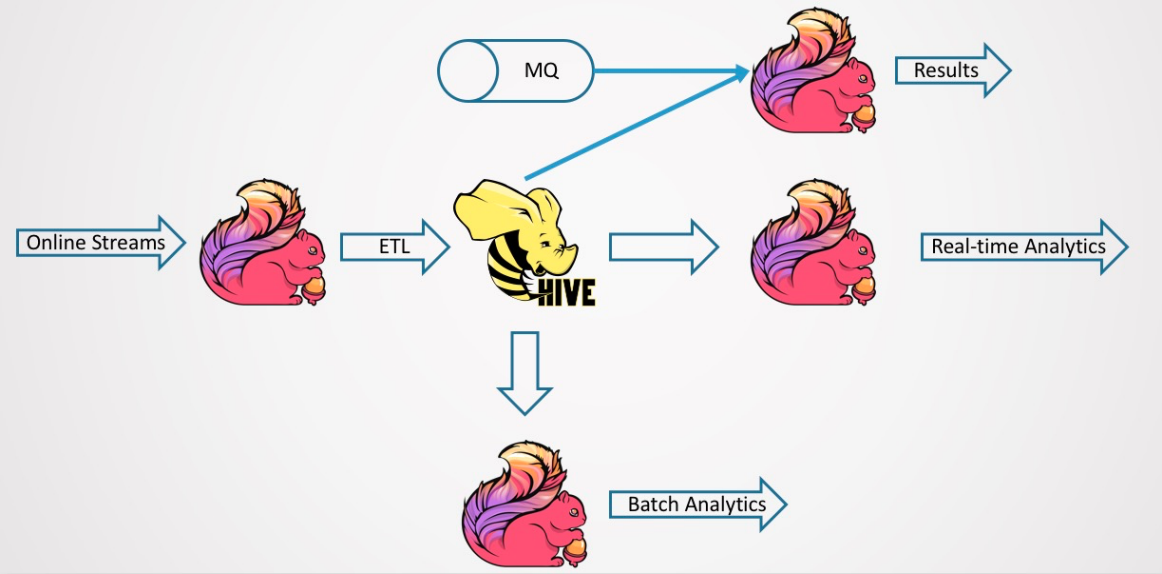
- 第一，基于 Queue，一般来说就是行存加 Queue，存储效率其实不高。
- 第二，基于预计算，最终会落到 BI DB，已经是聚合好的数据了，没有历史数据。而且 Kafka 存的一般来说都是 15 天以内的数据，没有历史数据，意味着无法进行 Ad-hoc 分析。所有的分析全是预定义好的，必须要起对应的实时作业，且写到 DB 中，这样才可用。对比来说，Hive 数仓的好处在于它可以进行 Ad-hoc 分析，想要什么结果，就可以随时得到什么结果。



能否结合离线数仓和实时数仓两者的优势，然后构建一个 Lambda 的架构?

核心问题在于成本过高。无论是维护成本、计算成本还是存储成本等都很高。并且两边的数据还要保持一致性，离线数仓写完 Hive 数仓、SQL，然后实时数仓也要写完相应 SQL，将造成大量的重复开发。还可能存在团队上分为离线团队和实时团队，两个团队之间的沟通、迁移、对数据等将带来大量人力成本。如今，实时分析会越来越多，不断的发生迁移，导致重复开发的成本也越来越高。少部分重要的作业尚可接受，如果是大量的作业，维护成本其实是非常大的。

如何既享受 Ad-hoc 的好处，又能实现实时化的优势？一种思路是将 Hive 的离线数仓进行实时化，就算不能毫秒级的实时，准实时也好。所以，Flink 1.11 在 Hive 流批一体上做了一些探索和尝试，如下图所示。它能实时地按 streaming 的方式来导出数据，写到 BI DB 中，并且这套系统也可以用分析计算框架来进行 Ad-hoc 的分析。这个图当中，最重要的就是 Flink Streaming 的导入。



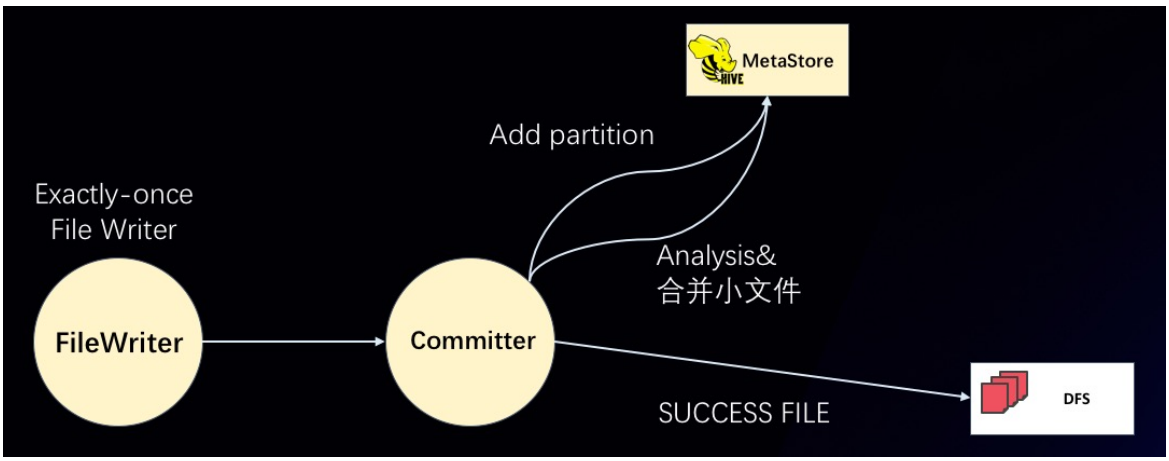
Streaming Sink

早期 Flink 版本在 DataStreaming 层，已经有一个强大的 StreamingFileSink 将流数据写到文件系统。它是一个准实时的、Exactly-once 的系统，能实现一条数据不多，一条数据不少的 Sink。

具体原理是基于两阶段提交：

- 第一阶段：SnapshotPerTask，关闭需要 Commit 的文件，或者记录正在写的文件的 Offset。
- 第二阶段：NotifyCheckpointComplete，Rename 需要 Commit 的文件。注意，Rename 是一个原子且幂等的操作，所以只要保证 Rename 的 At-least-once，即可保证数据的 Exactly-once。

这样一个 File system 的 Writer 看似比较完美了。但是在 Hive 数仓中，数据的可见性是依赖 Hive Metastore 的，那在这个流程中，谁来通知 Hive Metastore 呢?



SQL 层在 StreamingFileSink，扩展了 Partition 的 Committer。

相当于不仅要进行 File 的 Commit，还要进行 Partition 的 Commit。如图所示，FileWriter 对应之前的 StreamingFileSink，它提供的是 Exactly-once 的 FileWriter。而后面再接了一个节点 PartitionCommitter。支持的 Commit Policy 有：

- 内置支持 Add partition 到 Hive metastore；
- 支持写 SuccessFile 到文件系统当中；
- 并且也可以自定义 Committer，比如可以 analysis partition、合并 partition 里面的小文件。

Committer 挂在 Writer 后， 由 Commit Trigger 决定什么时机来 commit：

- 默认的 commit 时机是，有文件就立即 commit。因为所有 commit 都是可重入的，所以这一点是可允许的。
- 另外，也支持通过 partition 时间和 Watermark 来共同决定的。比如小时分区，如果现在时间到 11 点，10 点的分区就可以 commit 了。Watermark 保证了作业当前的准确性。

Streaming Source

Hive 数仓中存在大量的 ETL 任务，这些任务往往是通过调度工具来周期性的运行，这样做主要有两个问题：

- 实时性不强，往往调度最小也是小时级。
- 流程复杂，组件多，容易出现问题。

针对这些离线的 ETL 作业，Flink 1.11 为此开发了实时化的 Hive 流读，支持：

- Partition 表，监控 Partition 的生成，增量读取新的 Partition。
- 非 Partition 表，监控文件夹内新文件的生成，增量读取新的文件。

甚至可以使用 10 分钟级别的分区策略，使用 Flink 的 Hive streaming source 和 Hive streaming sink，可以大大提高 Hive 数仓的实时性到准实时分钟级，在实时化的同时，也支持针对 Table 全量的 Ad-hoc 查询，提高灵活性。

```
SELECT * FROM hive_table /*+ OPTIONS('streaming-source.enable'='true', 'streaming-sou
```

另外除了 Scan 的读取方式，Flink 1.11 也支持了 Temporal Join 的方式，也就是以前常说的 Streaming Dim Join。

```
SELECT
  o.amout, o.currency, r.rate, o.amount * r.rate
FROM
  Orders AS o
  JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r
  ON r.currency = o.currency
```

目前支持的方式是 Cache All，并且是不感知分区的，比较适合小表的情况。

Hive Dialect

Flink SQL 遵循的是 ANSI-SQL 的标准，而 Hive SQL 有它自己的 HQL 语法，它们之间的语法、语义都有些许不同。

如何让 Hive 用户迁移到 Flink 生态中，同时避免用户太大的学习成本？为此， Flink SQL 1.11 提供了 Hive Dialect，可以使得用户在 Flink 生态中使用 HQL 语言来计算。目前只支持 DDL，后续版本会逐步攻坚 Qeuries。

Filesystem Connector

Hive Integration 提供了一个重量级的集成，功能丰富，但是环境比较复杂。如果只是想要一个轻量级的 Filesystem 读写呢？

Flink table 在长久以来只支持一个 CSV 的 filesystem table，并且还不支持 Partition，行为上在某些方面也有些不符合大数据计算的直觉。

Flink 1.11 重构了整个 Filesystem connector 的实现：

- 结合 Partition，现在，Filesystem connector 支持 SQL 中 Partition 的所有语义，支持 Partition 的 DDL，支持 Partition Pruning，支持静态 / 动态 Partition 的插入，支持 overwrite 的插入。
- 支持各种 Formats：
  - CSV
  - JSON

- Aparch AVRO
- Apache Parquet
- Apache ORC

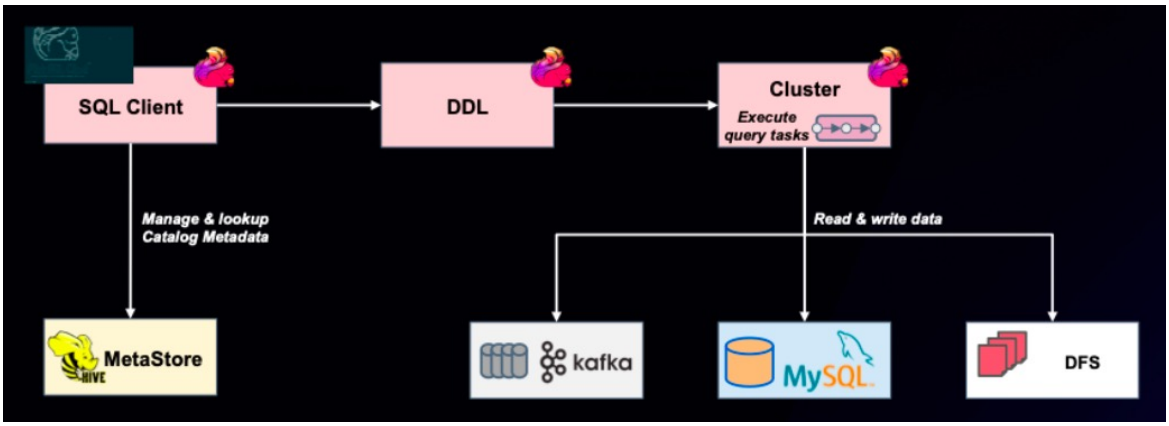
- 支持 Batch 的读写。
- 支持 Streaming sink，也支持 Partition commit，支持写 Success 文件。

用几句简单的 SQL，不用搭建 Hive 集成环境即可：

- 启动一个流作业写入 Filesystem 中，然后在 Hive 端即可查询到 Filesystem 上的数据，相比之前 Datastream 的作业，简单 SQL 即可搞定离线数据的入库。
- 通过 Filesystem Connector 来查询 Hive 数仓中的数据，功能没有 Hive 集成那么全，但是定义简单。

Table 易用性

DDL Hints 和 Like



在 Flink 1.10 以后，Hive MetaStore 逐渐成为 Flink streaming SQL 中 table 相关的 Meta 信息的存储。比如，可以通过 Hive Catalog 保存 Kafka tables。这样可以在启动的时候直接使用 tables。

通过 DDL 这种方式，把 SQL 提交到 cluster，就可以写入 Kafka，或者写入 MySQL、 DFS。使用 Hive Catalog 后，是不是说只用写一次 DDL，之后的流计算作业都是直接使用 kafka 的 table 呢?

不完全是，因为还是有一些缺陷。比如，一个典型的 Kafka table 有一些 execution 相关的参数。因为 kafka 一般来说都是存 15 天以内的数据，需要指定每次消费的时间偏移，时间偏移是在不断变化的。每次提交作业，使用 kafka table 的参数是不一样的。而这些参数又存储在 Catalog 里面，这种情况下只能创建另外一张表，所以字段和参数要重写一遍，非常繁琐。

- Flink 1.11，社区就开发了 Table Hints，它在 1.11 中目前只专注一个功能，即 Dynamic Table Options。用起来很简单，在 SQL 中 select from 时，在 table 后面写 table hints 的方式来指定其动态 options ，在不同的使用场景，指定不同的动态参数。

- Flink 1.11，引入了 Like 语法。LIKE 是标准的 SQL 定义。相当于 Clone 一张表出来复用它的 schema。LIKE 支持多种 constraints。可以选择继承，也可以选择完全覆盖。

Table Hints:

```
复制代码
SELECT id, name FROM kafka_table1 /*+ OPTIONS('scan.startup.mode'='earliest-offset')
```

LIKE:

```
复制代码
CREATE TABLE kafka_table2 WITH ( 'scan.startup.mode'='earliest-offset') LIKE kafka_ta
```

这两个手段在对接 Hive Catalog 的基础上，是非常好的补充，能够尽可能的避免在每次作业的时候都写一大堆 Schema。

内置 Connectors

Flink SQL 1.11 引入了新的三个内置 Connectors，主要是为了大家更方便的进行调试、测试，以及进行压测和线上的观察：

- Datagen Source：一个无中生有产生数据的 Source，可以定义生成的策略，比如 Sequence，比如 Random 的生成。方便线下进行功能性的测试，也可以拿来性能测试。
- Print Sink：直接在 Task 节点 Runtime 的打印出数据，比如线上作业某个 Sink 少数据了，不知道是上游发来数据有问题，还是 Sink 逻辑有问题，这时可以额外接一个 Print Sink，排查上游数据到底有没有问题。
- Blackhole Sink：默默把数据给吃掉，方便功能性的调试。

这三个 Connectors 的目的是为了在调试、测试中排除 Connectors 的影响，一般来说，Connectors 在流计算中是不可控的存在，很多问题把 Connectors 糅杂在一起，变得比较复杂难以排查。

SQL-API

TableEnvironment

TableEnvironment 作为 SQL 层的编程入口，无疑是非常重要的，之前的 API 主要是：

- Table sqlQuery：从一段 Select 的 query 中返回 Table 接口，把用户的 SQL 翻译成 Flink 的 Table。
- void sqlUpdate：本质上是执行一段 DDL/DML。但是行为上，当是 DDL 时，直接执行；当是 DML 时，默默 Cache 到 TableEnvironment，等到后续的 execute 调用，才会真正的执行。
- execute：真正的执行，提交作业到集群。

TableEnvironment 默默的 Cache 执行计划，而且多个 API 感觉上会很混乱，所以，1.11 社区重构了编程接口，目的是想要提供一个干净、并且不易出 bug 的清晰接口。

- 单 SQL 执行：TableResult executeSql(String sql)
- 多 SQL 执行：StatementSet createStatementSet()
- TableResult：支持 collect、print、getJobClient

现在 executeSql 就是一个大一统的接口，不管是什么 SQL，是 Query 还是 DDL 还是 DML，直接丢给它都可以很方便地使用起来。

并且，和 Datastream 也有了很清晰的界限：

- 调用过 toDataStream：一定要使用 StreamExecutionEnvironment.execute
- 没调用过 toDataStream：一定要使用 TableEnvironment.executeSql

SQL-Client

SQL-Client 在 1.11 对齐了很多 Flink 内部本来就支持的 DDL，除此之外值得注意的是，社区还开发了 Tableau 的结果展示模式，展示更自然一些，直接在命令行展示结果，而不是切换页面：

```
+-----+----------------------+----------------------+
| +/- |                 name |                  cnt |
+-----+----------------------+----------------------+
|   + |                  Bob |                    1 |
|   + |                Alice |                    1 |
|   + |                Greg  |                    1 |
|   - |                  Bob |                    1 |
|   + |                  Bob |                    2 |
+-----+----------------------+----------------------+
Received a total of 5 rows
```

总结和展望

上述解读主要侧重在用户接口方面，社区已经有比较丰富的文档，大家可以去官网查看这些功能的详细文档，以便更深入的了解和使用。

Flink SQL 1.11 在 CDC 方面开了个头，内部机制和 API 上打下了夯实的基础，未来会内置更多的 CDC 支持，直接对接数据库 Binlog，支持更多的 Flink SQL 语法。后续版本也会从底层提供更多的流批一体支持，给 SQL 层带来更多的流批一体的可能性。