实现MySQL/ES数据增量同步的Binlog方案

目录

一、背景

在GBI系统中，问数之前需要接入MySQL数据源，目前存在2种数据源:

- **文件数据源**，文件数据会被导入到内置的MySQL数据源中，这部分数据是不会变更的，只需要一次全量同步到ES建立选表选列与专业词的索引即可;

- **用户数据源**，GBI系统直接接入用户的数据库，比如BAP与智能客服的业务场景，这种场景下，用户在首次接入时，可以执行全量同步到ES，但是随后用户数据源的数据发生了表的增删改操作后这些变更需要反馈到ES中。如果使用全量同步ES方案，虽然实现比较简单但是代价比较高，存在大量数据的重复处理与资源的浪费同时随着表数据的增加，全量同步的时延会越来越不可控。因此需要对数据源表的变更执行增量同步，只对变更的数据做处理。无须处理无关的数据，这种同步是近实时的并且同步需要的资源比较少。

目前，实现数据库增量变更获取的方案基本原理都是基于Log的方案，在MySQL数据库中这种方案就是常见的Binlog。MySQL是广泛使用的数据库，所以，目前主要考虑MySQL的增量同步方案。目标就是构建一个高性能、可扩展具有容错机制的MySQL到ES增量同步系统。

二、功能与非功能需求分析

通过分析GBI中数据到ES的同步需求得知

- 用户数据源可以在任意状态下接入GBI，此时其Binlog可能可能只有短期内的部分日志，需要支持在接入时snapshot数据库表并在snapshot后接入Binlog
- 由于同时存在文件数据源与用户数据源2种场景，需要支持同时支持全量同步与增量同步
- 可能会接入很多数据源或者是终止接入数据源。需要Binlog连接管理是动态的，系统具有新增接入、修改接入、删除接入等增删改查的操作
- Binlog可能是用户数据库全部的数据变更，需要考虑通过GBI的源数据信息来过滤出需要的数据变更事件。
- 数据变更事件需要带有表名、列名等信息，而原始的Binlog日志中不包含这些信息。
- 在云上部署或者容器部署时，一个pod也就是服务可能管理多个Binlog连接，当Pod异常时或者系统所容时，需要把当前进程管理的连接分配到其他服务中，并且从断开的postion位置继续处理，不会产生丢失数据或者重复处理(目前是幂等的)的情况，也就是需要系统具有容错性与可靠性
- 需要考虑schema变更情况如何正确的处理数据，如果删除列需要删除对应的数据，如果新增列需要新增对应的数据
- 增量变更需要实时的把变更数据同步到ES，需要实时性得到保证，大量的数据处理需要保证一定的性能。目前暂无特定强制性需求
- 服务未来可能会根据数据处理情况扩容，需要重新分配Binlog连接与数据的处理，保证系统的可扩展性
- 因为开发时间有限，实现方案需要简单易实现
- GBI在产品的初级阶段，最好需要尽可能少的资源或者较少的依赖实现增量同步，人员有限，尽可能减少运维支持或者人工干预。

三、MySQL Binlog

MySQL Binlog也就是MySQL的Binary Log，专门用来记录数据变更事件日志，MySQL中的复制基于这个日志。Binlog只记录可以产生变更的statements的事件。Binlog支持3种format的事件:

- Statement Format：记录SQL statesment
- Row Format：记录行的变更，这是MySQL8.0默认的format
- Mixed Format：混合Fomrat，默认是Statement Format，在特定的情况下，比如SQL执行需要依赖特定的环境，比如机器时间等，为了保持复制的一致性，会自动切换为Row Format记录

Binlog Event的详细内容如下:

- Pos：事件开始位置
- Event_type：事件类型
- Server_id：产生这个事件的 MySQL server_id
- End_log_position：下一个事件的开始位置
- Info：事件类型相关的具体信息，可能是文本或者二进制

调研技术方案过程中默认使用基于ROW的Format。

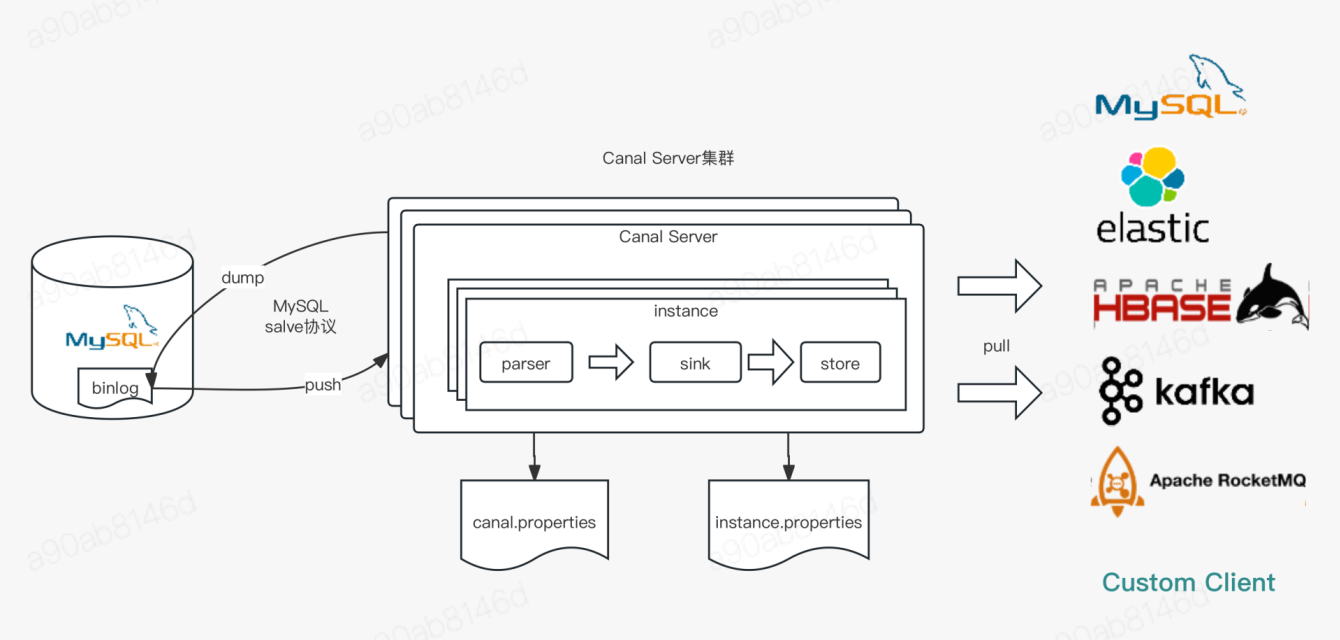
四、方案调研

4.1 Canal

阿里开源的用于增量订阅&消费的业务的开源工具，最初用来异地复制MySQL数据，其实现原理:

- Canal通过MySQL的replication机制，伪装成MySQL的Slave，向MySQL发送Dump命令
- MySQL收到dump请求，开始推送Binary logs给Canal
- Canal解析Binary log事件形成数据变更事件对象提供给下游的消费者，可以是MQ或者自定义的Client

其基本架构如下



Canal架构

一个集群包含多个Server，集群之间通过Zk实现高可用，一个Server包含多个Instance，一个Instance就是一个Binlog订阅。多次订阅只有一个保证在运行，通过Zk实现Failover。通过Canal-Admin的Rest API或者UI支持运行时Server/Instance变更。Instance目前只支持Memory一种，当发生故障后切换或者客户端来不及消费的情况，可能发生丢失事件的风险。

对于实现增量数据同步，Canal的优势有:

- 使用的是Binlog的数据变更获取方案，提供了足够的实时性
- 通过Canal-admin提供的Rest API与其本身组件的机制，可以实现Binlog接入的运行时管理
- 内置支持基于库表的事件过滤，并且事件中包含表的源数据信息
- 具有较高的容错性与扩展性

不满足的地方:

- 不支持第一次连接snapshot
- 不支持schema变更处理
- 需要部署运维较多的服务canal-admin，zk，canal-server集群等

4.2 mysql-binlog-connector-java

一个简单的MySQL Binlog事件连接与处理库，最开始来源于open-replicator项目，一个用Java写的高性能Binlog解析工具。接收所有的原始的Binlog事件，对于Binlog提供了反序列化的接口，可以自定义反序列化哪种Binlog的事件而忽略别的，提供了连接与事件处理时的一些回调。是很多Binlog方案的底层实现依赖

</> Java | 收起

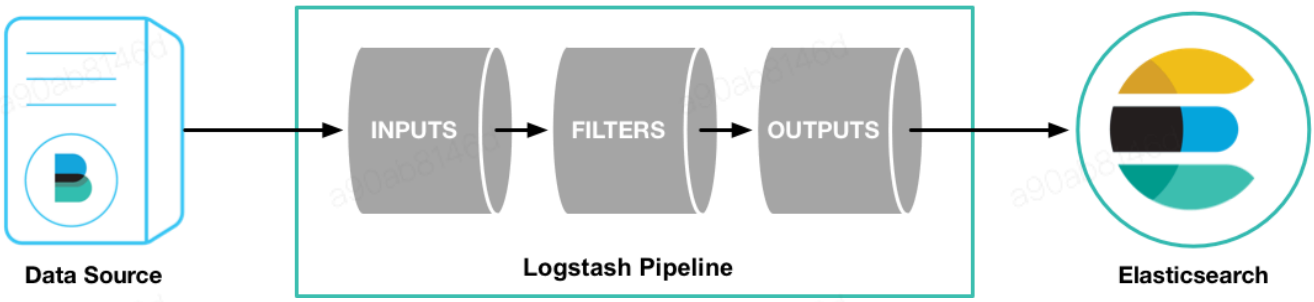
```java
BinaryLogClient client = new BinaryLogClient("localhost", 3306, "root", "123456");
EventDeserializer eventDeserializer = new EventDeserializer();
eventDeserializer.setCompatibilityMode(
    EventDeserializer.CompatibilityMode.DATE_AND_TIME_AS_LONG,
    EventDeserializer.CompatibilityMode.CHAR_AND_BINARY_AS_BYTE_ARRAY
);
client.setEventDeserializer(eventDeserializer);
client.registerEventListener(new EventListener() {

    @Override
    public void onEvent(Event event) {
        // 事件处理  WriteRowsEventData UpdateRowsEventData DeleteRowsEventData等事件
    }
});
client.connect();
```

因为它只是一个库，只用来接收Binlog事件，所有其他的工作都需要开发人员自行实现，对于超轻量级的场景可能比较有用。对于实现增量MySQL到ES，mysql-binlog-connector-java的优质只有极大的灵活性与自由度，所有处理由开发人员自行决定。需要较大的开发工作量。

4.3 Logstash

Logstash是ELK中的L。是一个数据流工具引擎，执行ETL操作，也可以说是一个声明式的插件化设计的数据管道。它可以摄入日志、文件、指标等数据，经过Logstash的处理或者转换，将数据输出到其他的应用、存储或变为其它的流式数据。Logstash包含3个主要部分每一个部分对应一类插件:

- input, 负责从数据源采集数据，官方提供了大量的插件，比如jdbc、udp、日志等
- filter，将数据修改为你指定的格式或内容，官方提供了一些预定义的过滤器
- output，将数据传输到目的地，官方提供了一些存储output或者一些MQ、文件的output



logstash数据处理示意图

Logstash常用于把MySQL或者服务的日志等同步到Elasticsearch。一种把MySQL数据增量同步到ES的方案如下:

```
input {
  jdbc {
    jdbc_driver_library => "/Users/zhangyongxiang/Downloads/mysql-connector-j-8.3.0/mysql-connector-j-8.3.0.jar"
    jdbc_driver_class => "com.mysql.cj.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://xxxx.xxx.xxx:3306/test?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=true&zeroDateTimeBehavior=convertToNull"
    jdbc_user => "root"
    jdbc_password => "xxxxx"
    jdbc_paging_enabled => true
    tracking_column => "unix_ts_in_secs"
    use_column_value => true
    tracking_column_type => "numeric"
    schedule => "*/1 * * * *"
    record_last_run => true
```

```
    last_run_metadata_path => "/Users/zhangyongxiang/Documents/elasticsearch-learn/logstash/project_last_run"
    statement => "SELECT *, UNIX_TIMESTAMP(created_time) AS unix_ts_in_secs FROM xxx WHERE (UNIX_TIMESTAMP(created_time)) > :sql_last_value AND created_time < NOW() ORDER BY created_time asc"
  }
}
filter {
    ...
}
output {
    elasticsearch {
        index => "view_project_idx"
        document_id => "%{[@metadata][_id]}"
        doc_as_upsert => true
        action => "update"
        hosts => "http://localhost:9200"
        manage_template => true
        template => "/opt/project-index-template.json"
        template_name => "project-index-template"
        template_overwrite => true
    }
}
```

Logstash更多的是一种手工执行的任务。

对于实现增量增量同步，Logstash的优势有:

- 支持第一次连接的snapshot
- 支持基于库表的事件过滤
- 实现比较简单

不满足的地方:

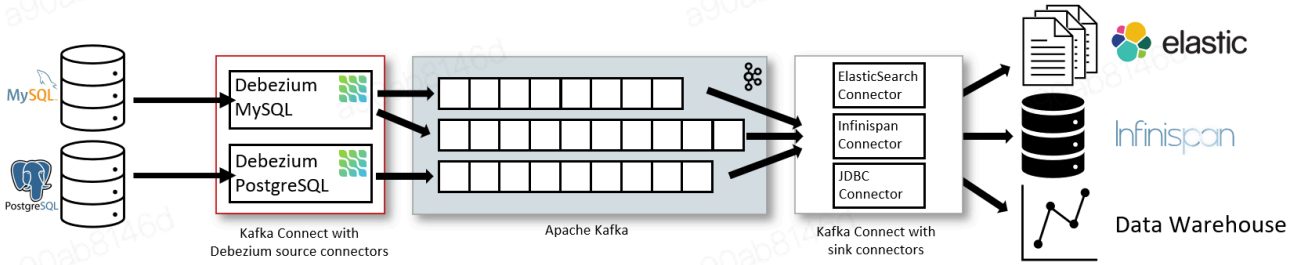
- 不支持数据源接入时的运行时管理
- 不是Binlog的方案，实时性无法得到保证，轮询式的访问可能会造成一定的浪费
- 数据不包含表的源数据信息
- 不具有容错型与扩展性
- 需要自定义filter plugin实现数据业务逻辑
- 对于表有特定的要求，count+select+特定的递增列实现的增量同步，无法处理数据变更与数据删除的场景，也无法处理Schema变更的场景

4.4 Debezium

Debezium是记录数据库变更的组件，在很多方案中都有使用。只记录行级别的变更并形成一个数据变更事件流。底层使用mysql-binlog-connector-java实现Binlog接入。其本身只是提供了兼容Kafka Connect协议的一些Source Connectors，这些Connectors大部分都是用与CDC的。有3种应用形式

4.4.1 Debezium Cluster

本身就是Kafka Connect架构，将Debezium Connectors与Kafka Connect一起部署，通过Kafka Connect的Rest API可以实现动态建立数据变更事件管道。通过Kafka/Kafka Connect实现系统的高可用、容错与扩展性。基本的原理就是通过Kafka Connect的Rest API与一个Connector插件建立一个数据管道，数据管道摄取数据库的所有变更事件，包括schema与行数据的变更与position等，分别发送到Kafka集群中对应的topic中。在发送到topic前可以对变更事件做数据过滤，路由等一些数据管道的处理，发送到kafka后的事件可以被别的应用或者Kafka Connect本身的Sink Connector处理。的其基本架构如下:



对于实现数据增量同步，Debezium Cluster优势有:

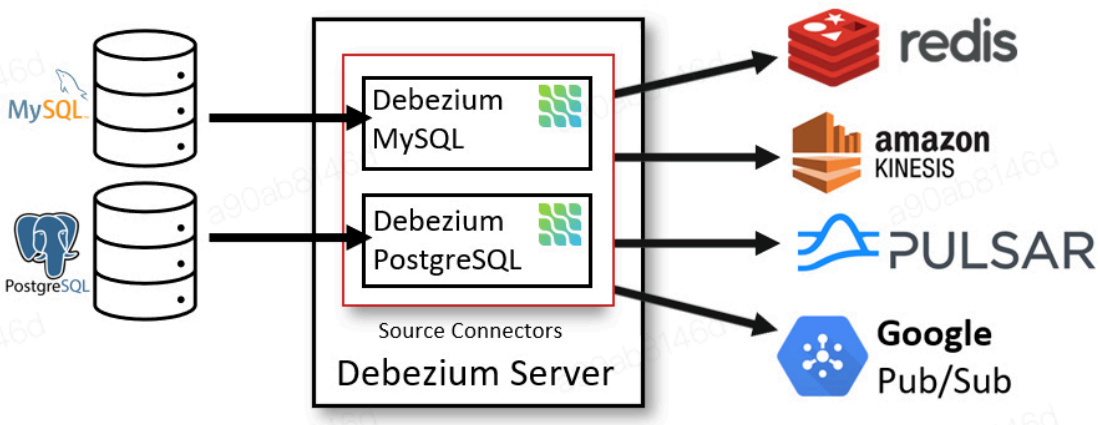
- 使用的是Binlog的数据变更获取方案，提供了足够的实时性
- 通过Kafka Connect提供的Rest API，可以实现Binlog接入的运行时管理
- 内置支持基于库表的事件过滤，并且事件中包含表的元数据信息
- 具有较高的容错性与极大的扩展性
- 支持第一次连接snapshot，支持schema变更处理

不满足的地方:

- 需要部署运维较多的服务，包括Zookeeper集群、Kafka集群、Kafka Connect集群
- 对特定的topic有要求，需要对一些topic的数据做永久存储
- 每个表对应一个topic，topic可能会过多，所有表对应一个topic，topic可能会不太够用
- 因为变更事件的顺序的原因，只能使用单分片，限制了性能

4.4.2 Debezium Server

一个封装好的Server，用于把变更事件信息直接发送到消息中间件，比如Kafka、Pulsar等。直接部署就可以使用，需要静态配置一个Connector，且不支持运行时变更。其基本架构如下:



Debezium Server

不满足使用场景，暂不考虑。

4.4.3 Debezium Engine

将Debezium Connector直接内嵌到Java应用，直接消费变更事件。与直接使用mysql-binlog-connector-java的方式类似，相比mysql-binlog-connector-java，Debezium Engine提供了更全面的事件信息，包含schema变化与列名。因为没有了Kafka Connect/Kafka，所以也就没有了分布式系统带有的扩展性与容错性，需要开发人员自行维护。其基本实现思路如下，添加依赖到Java应用中

```xml
        <dependency>
            <groupId>io.debezium</groupId>
            <artifactId>debezium-api</artifactId>
            <version>2.5.0.Final</version>
        </dependency>
        <dependency>
            <groupId>io.debezium</groupId>
            <artifactId>debezium-embedded</artifactId>
            <version>2.5.0.Final</version>
        </dependency>
        <dependency>
            <groupId>io.debezium</groupId>
            <artifactId>debezium-connector-mysql</artifactId>
            <version>2.5.0.Final</version>
        </dependency>
```

一个简单的Demo

Java | 收起

```
    final Properties props = new Properties();
        props.setProperty("name", "engine");
        props.setProperty("connector.class",
                "io.debezium.connector.mysql.MySqlConnector");
        props.setProperty("offset.storage",
"org.apache.kafka.connect.storage.FileOffsetBackingStore");
        props.setProperty("offset.storage.file.filename",
"/tmp/offsets.dat");
        props.setProperty("offset.flush.interval.ms", "60000");
        /* begin connector properties */
        props.setProperty("database.hostname", "xxxx.xxx.com");
        props.setProperty("database.port", "3306");
        props.setProperty("database.user", "root");
        props.setProperty("database.password", "123456");
        props.setProperty("database.server.id", "85744");
        props.setProperty("topic.prefix", "my-app-connector");
        props.setProperty("schema.history.internal",
                "io.debezium.storage.file.history.FileSchemaHistory");
        props.setProperty("schema.history.internal.file.filename",
                "/tmp/schemahistory.dat");
        // Create the engine with this configuration ...
        try (DebeziumEngine<ChangeEvent<String, String>> engine =
DebeziumEngine
                .create(Json.class).using(props).notifying(record -> {
                    try {
                        System.out.println(record);
                    } catch (Exception e) {

                    }
                }).build()) {
            ExecutorService executor =
Executors.newSingleThreadExecutor();
            executor.execute(engine);
        }
```

Debezium Engine会把offset/schema等信息存储到文件中，放到服务器执行时需要把它们放到持久性存储中间件中。

对于实现增量数据同步，Debezium Engine的优势有:

- 使用的是Binlog的数据变更获取方案，提供了足够的实时性，并且因为是直接处理变更事件，给予开发人员很大的自由决定如何处理数据
- 因为时编程的方式接入数据源Binlog，实现Binlog接入的运行时管理是非常方便的

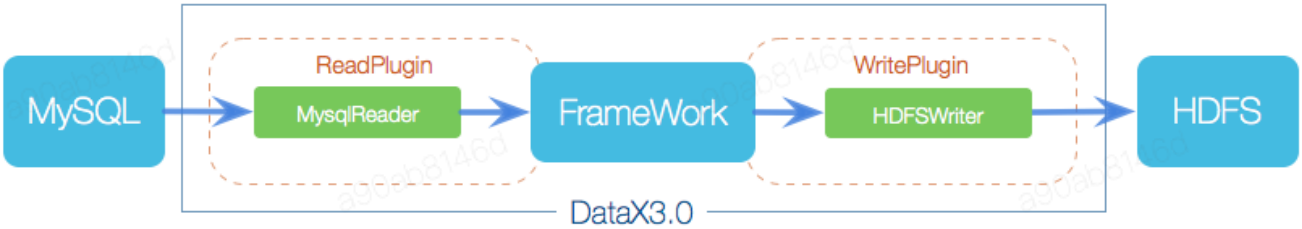
- 内置支持基于库表的事件过滤，并且事件中包含表的元数据信息
- 支持第一次连接snapshot，支持schema变更处理
- 单个服务，实现简单，部署运维比较简单，适用轻量化的实现方案

不满足的地方：

- 需要开发人员自己处理容错于扩展性
- 没有了Kafka topic，需要自己开发适用于k8s部署环境的offset/schema的backstore

4.5 DataX

阿里云开源的离线数据同步管道，有点类似Logstash。目前是GBI使用的方案。其原理与Logstash类似。



Framework(channel)+plugin架构构建。将数据摄取和输出抽象成为Reader/Writer插件，类似Logstash的Input/output。

- Reader：Reader为数据摄取模块，负责采集数据源的数据，将数据发送给Framework，官方提供了大量的Reader，GBI使用是JDBC Reader，本质是执行SQL
- Writer： Writer为数据写入模块，负责不断向Framework取数据，并将数据写入到目的端。官方提供了大量的Writer，没有提供HTTP接口的Writer，GBI项目自行编写RestWriter组件。
- Framework：Framework用于连接reader和writer，作为两者的数据传输通道，并处理缓冲，流控，并发，数据转换等核心技术问题

DataX的更适用于全量数据同步，对于实现增量数据同步，DataX的优势有：

- 对于RDBMS的Reader来说，可以根据数据表的分区情况设置并发读取，提升吞吐量与性能
- 使用起来比较简单方便
- 其最新版本增加了任务容错与重试机制等，相比Logstash具有更高的稳定性
- 在数据转换上面支持Groovy函数等

不满足的地方：

- 与Logstash类似，其实时性较差，本身不能实现数据源接入的运行时管理，需要依托一个特定的任务管理平台，来管理任务的创建、执行，终止等，其本身没有提供这样的平台
- 增量同步方案对特定的表有要求，count+select+特定的递增列实现的增量同步，无法处理数据变更与数据删除的场景，也无法处理Schema变更的场景，与Logstash的特点类似

4.6 Pentaho Kettle

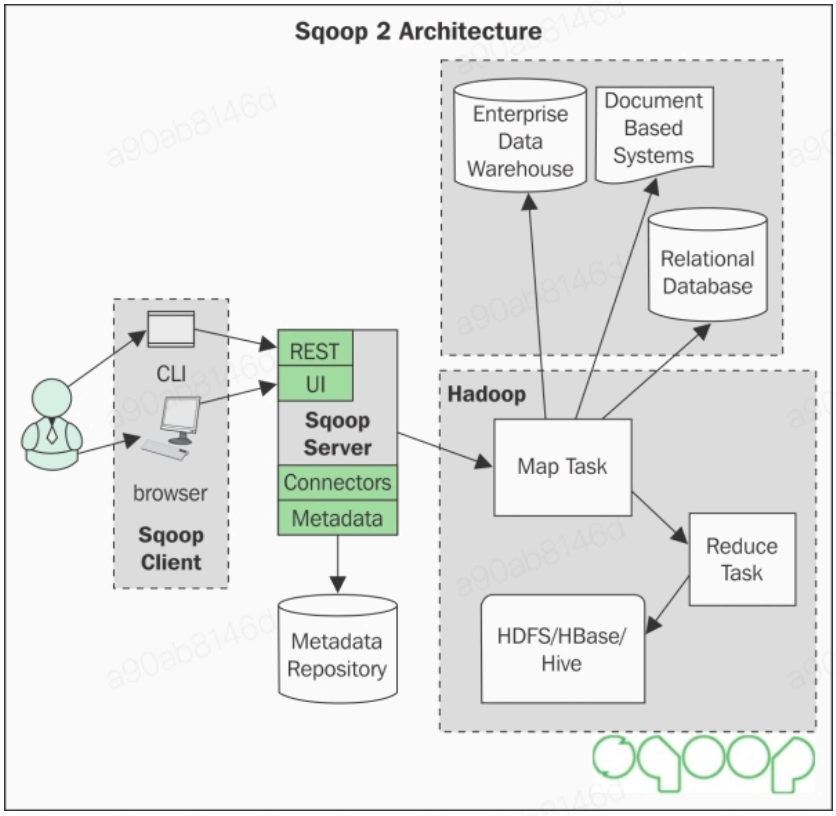
现在也叫做Pentaho Data Integration(PDI)，资料比较少，官网目前已找不到相关资料。Kettle提供了一低代码数据同步管道编辑器，在Kettle中，数据同步管道叫做一个Job或者作业。通过这

个编辑器可以指定数据源、执行的查询SQL，需要做的数据转换与数据的输出地址。生成的文件可以保存到一个Repository中，比如数据库中。可以通过Kettle本身直接运行数据同步作业，也可以将作业导出成一个*.kjb作业文件，通过Java应用指定*.kjb来运行。

- 其实现数据同步的原理是通过SQL语句的select，实现增量需要schema定义递增列，并创建多个作业来运行
- 生成kjb作业文件只能通过其低代码平台，暂不清楚生成作业文件的方式

4.7 Apache Sqoop

Sqoop也是一个数据同步引擎。Sqoop1只能用于将RDB数据迁移到Haddoop，Sqoop2支持结构化、半结构化、无结构的数据源直接传输数据，比如RDB、Casandre、Kafka等。其基本原理与架构如下:



Sqoop分为Client与Server，Server依托于Hadoop环境，所以必须有Haoop、YARN、HDFS等环境。Server是实际执行数据同步任务的地方，Client分为CLI或者Java客户端，用于向Server提交任务或者获取信息等，分为Interactive模式与Batch模式，都是通过Server的REST API与Server通信。可靠性与扩展性通过Hadoop来维护。通过Job接口创建数据同步管道并启动。基本的任务模型为，Link是input/output的统一抽象，Job类似一个执行数据转换与过滤的管道，指定了 data from which link to which link。

Sqoop2 官方提供了FTP/JDBC/Kafka/Kite/SFTP等几种Connector。Sqoop2 REST API提供可以对于Link/Job的增删改查与启动停止的API。

增量数据同步的优势:

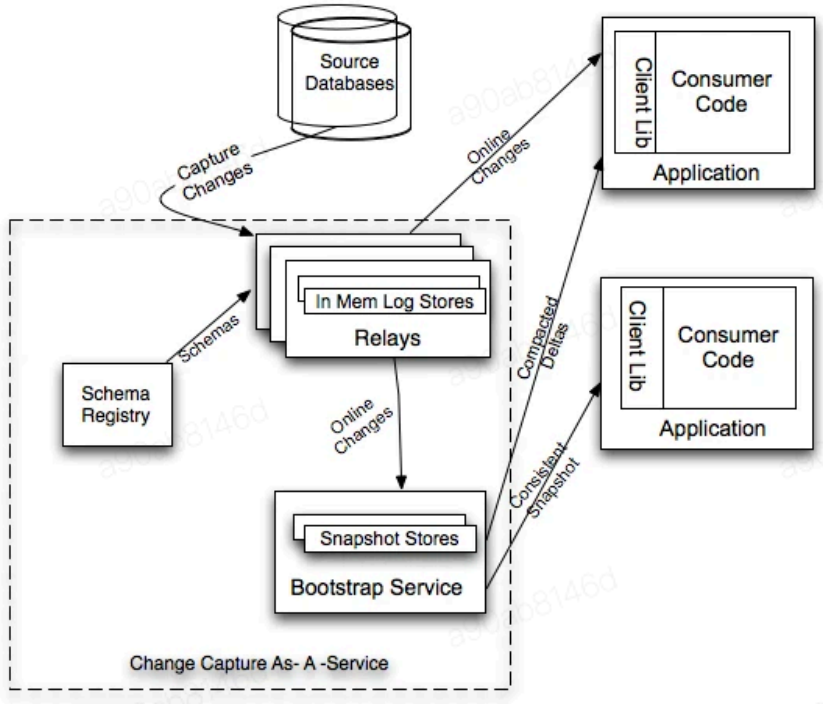
- 通过简单易用的API，Sqoop2本身支持运行时数据管道的动态管理
- 依托Hadoop提供了扩展性，可靠性暂不清楚

不满足的地方有:

- 不支持schema变更处理
- 其JDBC同步方案类似，是一种快照式的同步方案，使用SQL语句来完成数据摄取，在实现增量时候需要大量的Job，实时性无法得到保证
- 部署运维非常麻烦，需要大量的大数据相关的组件

4.8 Linkedin Databus

Databus是一个低延迟的数据变更获取系统，是Linkedin的数据处理pipeline的一部分。在MySQL的场景下是直接消费的Binlog，其基本架构如下:



Databus architecture

本身分为4个组件:

- Databus Relays, 从源数据库中的Databus源读取更改的行，并将它们序列化为内存缓冲区中的 Databus 数据更改事件。监听来自Databus Client的请求并传输新的Databus数据更改事件

- Databus Clients，连接Relay，消费MySQL的数据变更事件。如果是新的Client，通过Bootstrap service全量处理，然后消费Relay的新的变更事
- Databus Bootstrap Producers，一种特殊类型的Databus客户端，连接Relay，把所有的变更历史都存储到MySQL中，当有新的Databus Clients连接时，直接通过Bootstrap service全量处理历史数据
- Databus Bootstrap Servers，监听来自Databus Clients的请求，返回历史数据变更事件

Databus内部十分复杂，暂时也没有时间看懂了。

对于实现增量数据同步，DataBus方案的优势有:

- 使用的是Binlog的数据变更获取方案，提供了足够的实时性
- 支持基于库表的事件过滤，并且事件中包含表的源数据信息
- 具有较高的容错性与扩展性，因为组件都是集群部署的，其Client实现了负载均衡，在性能上会更高一些
- Databus通过记录所有变更事件到MySQL的方式使第一次连接的snapshot与普通的消费变更事件处理一致
- 支持schema变更处理

不满足的地方:

- 按照demo例子所示，Databus可能不能实现Binlog接入的运行时管理，其offset/schema管理是基于机器文件的，依赖特定的机器
- 需要部署运维较多的服务

4.9 技术方案对比

✅ 表示支持

❌ 表示不支持

– 表示需要外部控制逻辑辅助实现

★ 评分

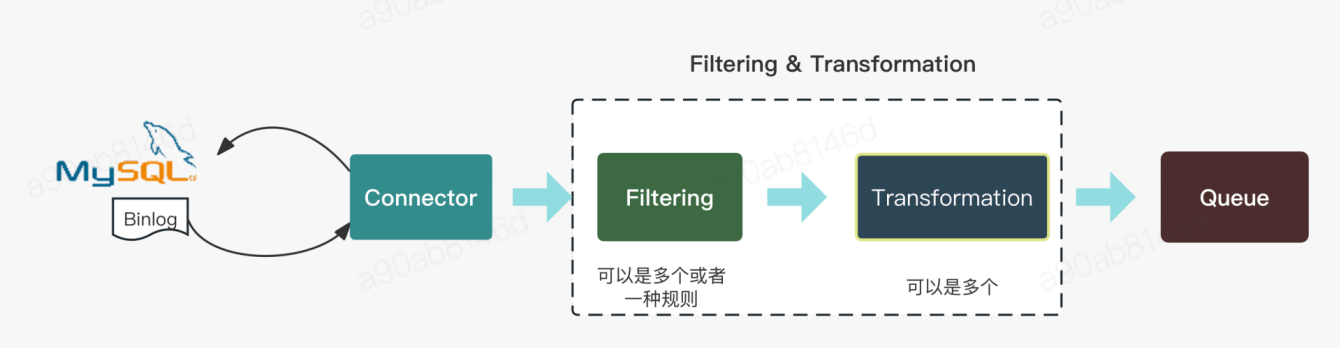
	Canal	binlog-java	Logstash	Debezium Cluster	Debezium Engine
scalability	★★★	–	–	★★★☆	–
fault tolerance	★★★	–	–	★★★	★
dynamic datasource	★★	★★★	★	★★	★★★
realtime	★★	★★★	★	★★	★★★

maintainability	★★	★★★	★★	★	★★★
resource demand	★★	★★★	★★	★	★★★
simplicity	★★	★	★★	★★	★★★
first time snapshot	❌	–	✅	✅	✅
schema change	❌	–	❌	✅	✅
transfromation	✅	–	✅	✅	✅
table metadata	✅	❌	❌	✅	✅

通过以上的对比得出结论用于指导增量数据同步系统的设计

- 基于Binlog的增量方案能够提供最好的实时性
- 基于Binlog的增量方案能够处理数据删除的事件，基于SQL的增量方案需要全量处理
- 基于Binlog的增量方案的扩展性与容错型需要分布式系统协调工具实现，比如Zookpper
- 扩展性与容错型越高的方案其所需要的资源与架构的复杂性越高，需要根据特定系统的需求对此做一定的权衡
- 一个完善的增量数据同步系统需要支持第一次连接的snapshot、schema变更追踪、数据变更事件携带表列元数据信息

一个完善的Binlog增量同步架构的模式



Binlog数据同步架构标准模式

五、方案设计

方案设计分为数据引擎方案与开源方案

1. 数据引擎方案是根据GBI目前的现状并参考Canal/Debezium Cluster的设计的简化方案，使用Debezium Engine底层库来解析Binlog。
2. 开源方案是把所有增量数据同步相关的问题委托给开源的中间件，GBI只做简单记录，中间件使用Canal。

实现目标:

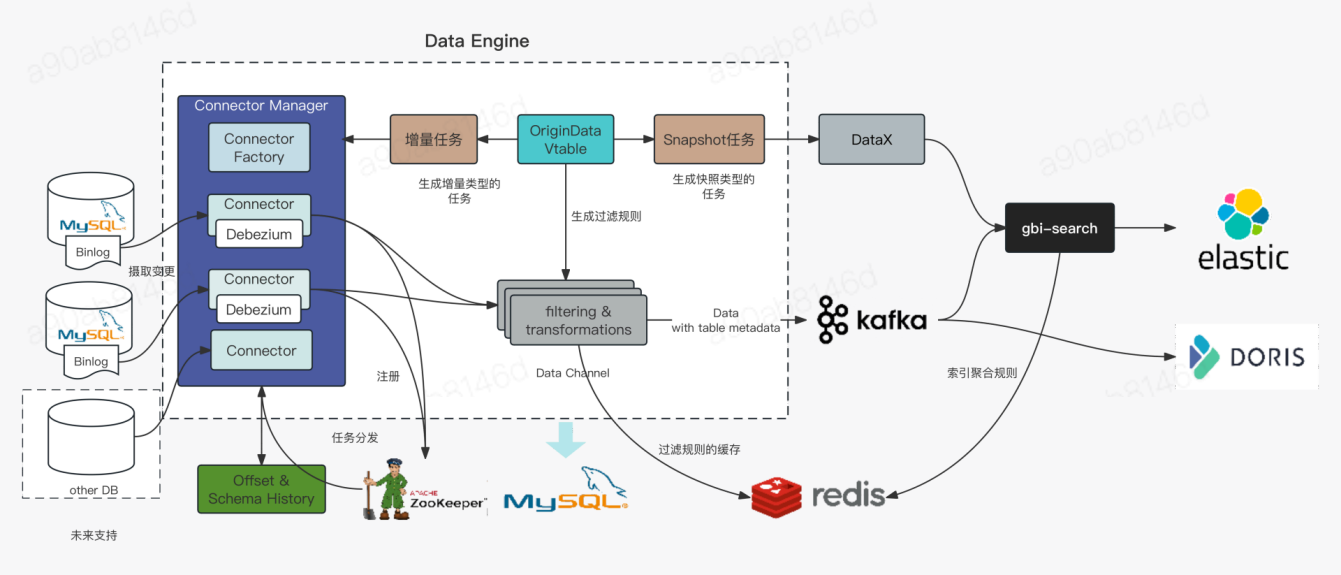
- 实现数据源Binlog动态接入，获取数据新增、修改、删除变更事件并将修改同步到ES与备份的Doris库
- 支持扩展性与容错，故障恢复，支持一定程度的实时性
- 支持第一次连接的snapshot、schema变更追踪、数据变更事件携带表列元数据信息
- 支持全量同步与增量同步同时存在，当数据库因为数据敏感等原因不提供Binlog相关资源，降级为人工触发的全量同步
- 在满足GBI需求的同时满足占用资源少，部署简单

5.1 数据引擎方案设计

数据引擎相关概念

- **Connector**: 类似Debezium的Connector或者Canal中Instance中的EventParser，用于摄取数据库的数据变更事件，这是一个抽象的概念，包括MySQL的数据变更事件获取或者其他类型的数据库的变更事件获取
- **Connector Factory:** 基于数据库中注册的增量任务信息生成Connector实例
- **Connector Manager**: 负责管理当前进程的所有的Connector，并负责于分布式协调中间件交互来执行任务分发与故障切换，目前分布式协调中间件暂定为Zookpper(还需要调研下)，或者通过Redis/MySQL来实现某种分布式协调，这里还需要调研下
- **Snapshot**：获取当前数据库表的全量内容的快照
- **Data Channel**: 增量数据变更事件管道，包含一个一个进程所有的Connectors作为输入，包含一个或者多个数据的过滤/转换器，包含一个或者多个Kafka Topics作为输出
- **Filtering**：过滤规则
- **Transformation:** 数据变更事件的内容转换

基本逻辑架构如下:



5.1.1 任务管理

主要是负责Snapshot任务与增量任务的增删改查、启动、停止与资源回收以及状态维护，在这里支持任务的降级。

在GBI目前的OriginData与Vtable的基础上，Snapshot任务复用之前系统存在的gbi_job，增量任务使用新的表来记录，主要包含的信息如下:

1	**gbi_cdc_job**	
2	cdc_job_id	用来标识cdc任务
3	vatble	cdc任务的配置信息，主要是通过vtable获取要摄取哪些MySQL的Binlog
4	connection	MySQL Binlog的连接信息
5	status	任务的状态信息

可能还需要一些用于故障恢复与任务分发的字段记录，暂时没想好。

5.1.2 Connector Manager

Connector Manager用于管理当前进程所有存活的Connector，内部使用Connector Factory来生成Connector，或者防止重复生成同一个数据源的Connector。要维护一个记录当前所有的Connector的连接信息，建立一个gbi_cdc_connector表

1	**gbi_cdc_connector**	
2	cdc_connector_id	用来标识connector
3	host	MySQL的地址
4	port	端口好
5	username	具有replication权限的用户

6	password	具有replication权限的用户的密码
7	parameter	连接参数

除此以外，Connector Manager可能还负责维护与分布式协调中间件的交互等（TODO）

5.1.3 Filtering & Transformation

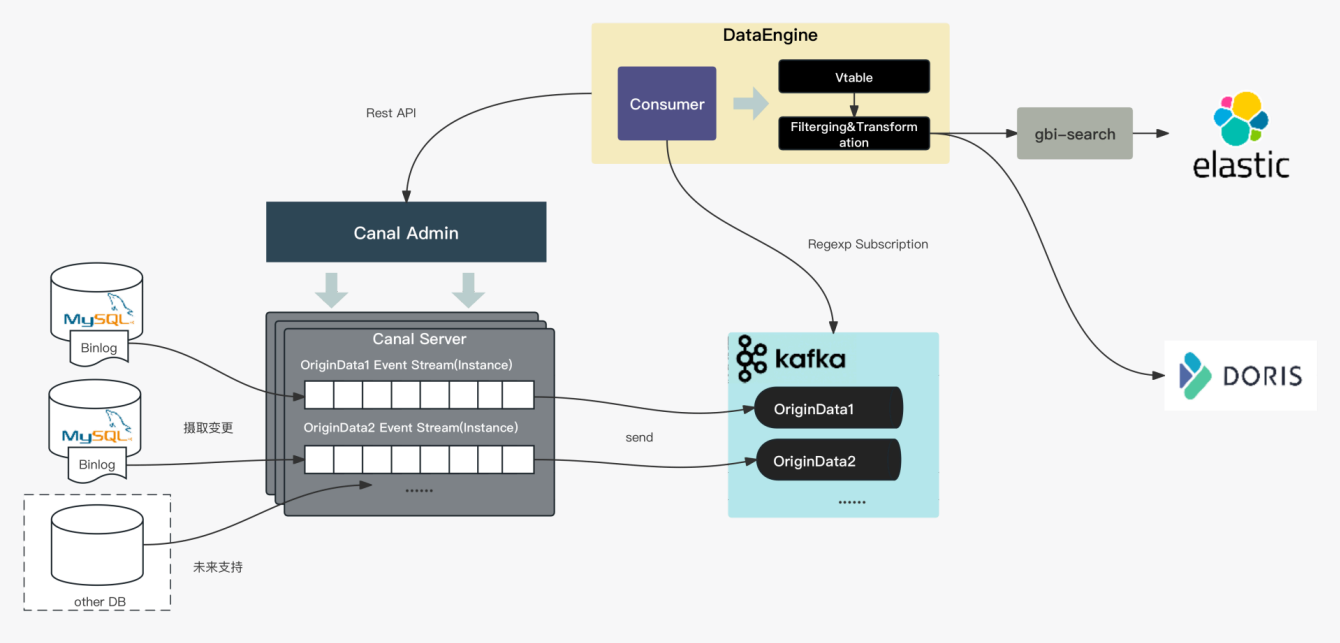
当前每个Data Engine进程维护一个线程池，每个线程完成filtering与transformations。

- filtering预期有3个，基于Vtable定义存在1.根据数据库名过滤，2.根据表名过滤，3.根据所选择的列过滤
- transformations预期有1个，就是把变更事件转换成一个容易理解的Json格式，并补充上一些必要的信息

5.2 开源方案设计

开源方案是把所有增量数据同步相关的问题委托给开源的中间件，GBI只做简单记录，中间件使用Canal

几本思想就是将GBI中的OriginData与Canal中的一个Instance与Kafka的一个Topic关联起来，通过Canal Admin的Rest API来操作任务。每个Canal的Instance摄取的数据都直接推送到Kafka的对应的Topic中，Data Engine消费者批量订阅这些Topic，获得数据后经过Vtable生成的规则过滤与转换将数据推送到Search与Doris。还保留之前的DataX的全量数据同步方案，这里忽略。基本的逻辑架构如下:



增量数据的开源方案

六、方案评估记录

2021-03-28

参与人	评估结果