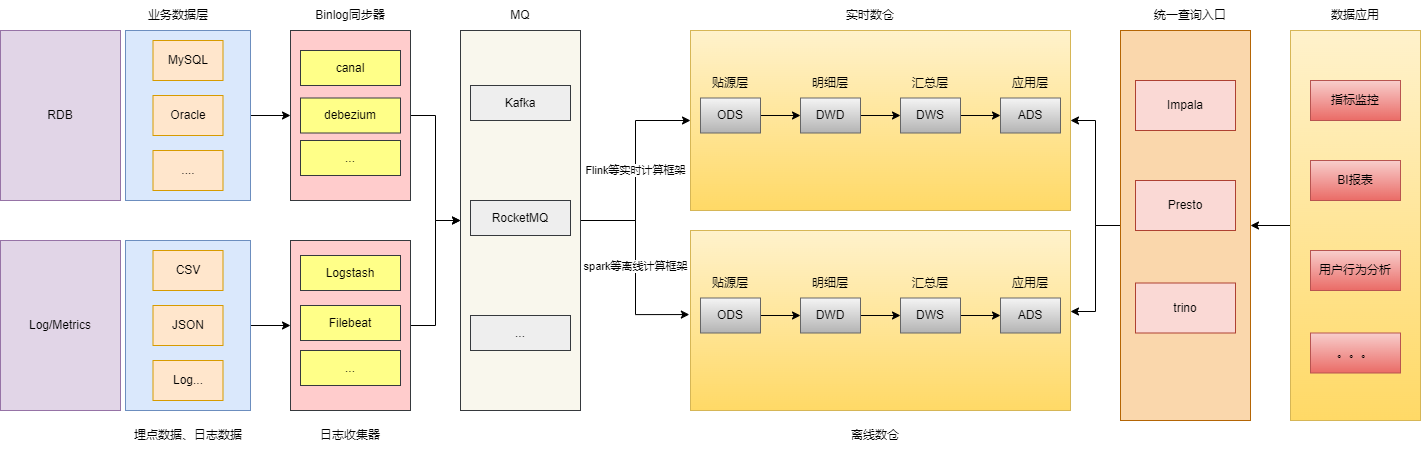
| 版本 | 作者 | 修改时间 |
| --- | --- | --- |
| V1.0 | 邵帅 | 2023-07-06 |

随着大数据应用的不断深入，企业不再满足离线数据加工计算的时效，实时数据需求已成为数据应用新常态。伴随着实时分析需求的不断膨胀，传统的数据架构面临的成本高、实时性无法保证、组件繁冗、运维难度高等问题日益凸显。为了适应业务快速迭代的特点，帮助企业提升数据生产和应用的时效性、进一步挖掘实时数据价值，实时数仓的构建至关重要。

本文将分享如何基于Apache Doris，快速构建一个极速易用的实时数仓。

# 实时架构

## Lambda架构



Lambda是比较经典的一款架构，以前实时的场景不是很多，以离线为主，当附加了实时场景后，由于离线和实时的时效性不同，导致技术生态是不一样的。而Lambda架构相当于附加了一条实时生产链路，在应用层面进行一个整合，双路生产，各自独立。在业务应用中，顺理成章成为了一种被采用的方式。

Lambda的架构分为三层：

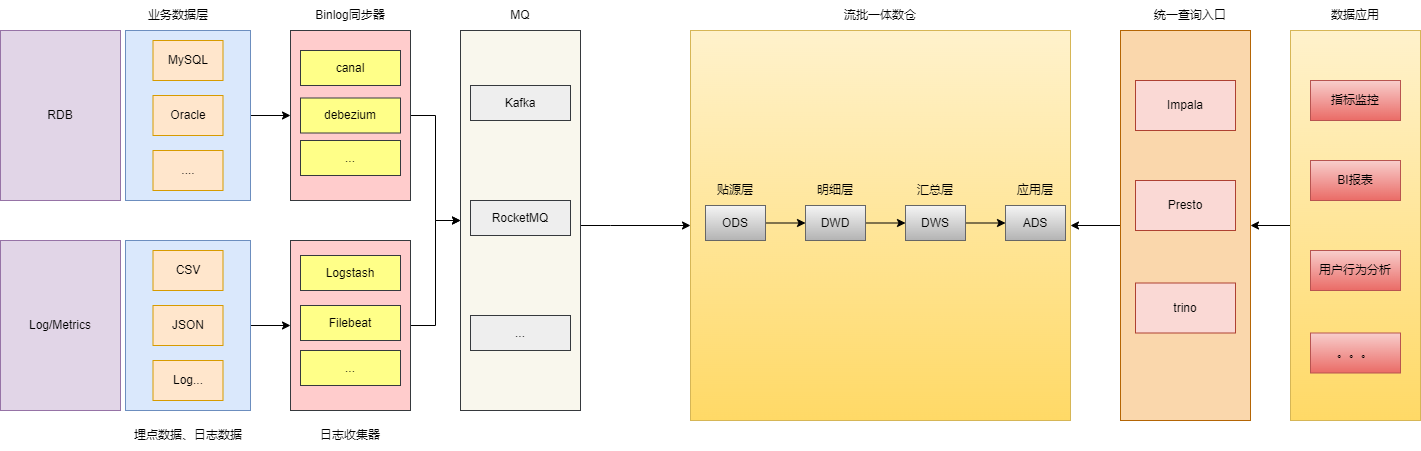
* Batch Layer：离线数仓和离线计算
* Speed Layer：实时数仓和实时计算
* Serving Layer：统一查询入口，屏蔽各个数据存储系统的差异

Lambda架构的缺点：

1. 需要维护两套计算逻辑：一般来说Spark，MapReduce主要用于离线计算逻辑，Flink用于实时计算逻辑。
2. 需要维护两套分布式存储架构：离线数仓一般用HDFS，S3，Hive等构建。实时数仓会采用Clickhouse，Doris来构建。
3. 最后，我们无法保障实时流的数据和离线数据的一致性。此时，只能通过离线数据定时清洗，保证数据的一致性。

为了解决Lambda架构遗留的问题，又引入了Kappa架构。

## Kappa架构



Kappa从架构设计来讲，比较简单，生产统一，一套逻辑同时生产离线和实时。与Lambda架构的区别就是采用流批一体数仓，Kappa架构虽说目前很火，但也会产生一些问题：

* 消息中间件缓存的数据量和回溯数据有性能瓶颈。通常算法需要过去180天的数据，如果都存在消息中间件，无疑有非常大的压力。同时，一次性回溯订正180天级别的数据，对实时计算的资源消耗也非常大。
* 在实时数据处理时，遇到大量不同的实时流进行关联时，非常依赖实时计算系统的能力，很可能因为数据流先后顺序问题，导致数据丢失。
* Kappa在抛弃了离线数据处理模块的时候，同时抛弃了离线计算更加稳定可靠的特点。Lambda虽然保证了离线计算的稳定性，但双系统的维护成本高且两套代码带来后期运维困难。

## 痛点与需求

|  | 优点 | 缺点 |
| --- | --- | --- |
| Lambda | 1.架构简单<br />2.兼顾了离线批处理和实时流处理的优点<br />3.稳定，实时计算成本可控<br />4.离线数据易于修正 | 1.需要维护两套计算逻辑<br />2.需要维护两套分布式存储系统<br />3.很难保证离线，实时数据一致性 |
| Kappa | 1. 去掉了Lambda架构中Batch Layer模块，只需要维护实时模块<br />2.无需离线实时数据Merge | 1.依赖MQ的能力<br />2.实时数据处理时存在丢失数据的可能 |

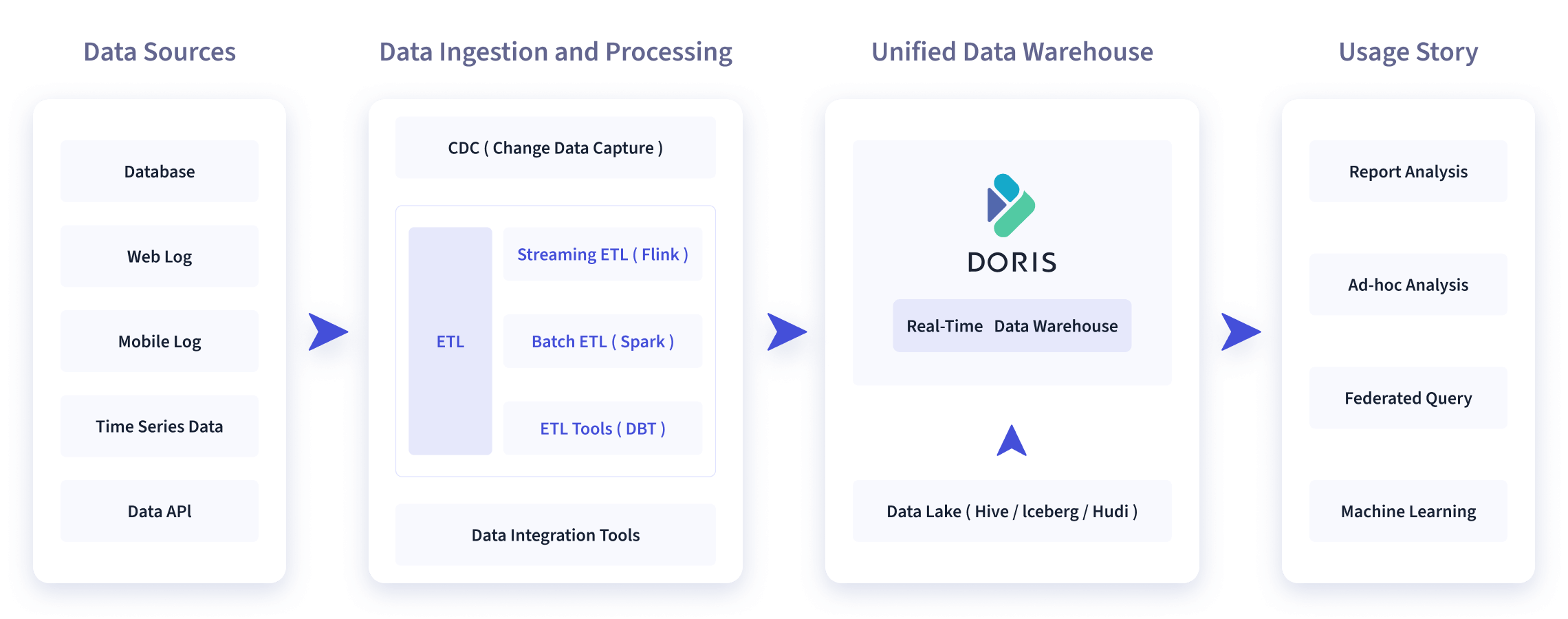
为了兼顾两种架构的优点，一个统一的实时架构呼之欲出，。“统一”是指数据结构的统一，我们希望半结构化和结构化数据能够统一存储在一起。同时也指的是在一个平台中能完成多种分析和计算场合（实时查询，交互式分析，批量处理等）。

# 基于Apache Doris与Apache Flink构建实时数仓

## Apache Doris介绍

首先先介绍一下Doris。Apache Doris 是一个基于 MPP 架构的**高性能**、实时的分析型数据库，以**极速易用**的特点被人们所熟知，仅需亚秒级响应时间即可返回海量数据下的查询结果，不仅可以支持高并发的点查询场景，也能支持高吞吐的复杂分析场景。

因此，Apache Doris 能够较好的满足报表分析、即席查询、统一数仓构建、数据湖联邦查询加速等使用场景，用户可以在此之上构建用户行为分析、AB 实验平台、日志检索分析、用户画像分析、订单分析等应用。

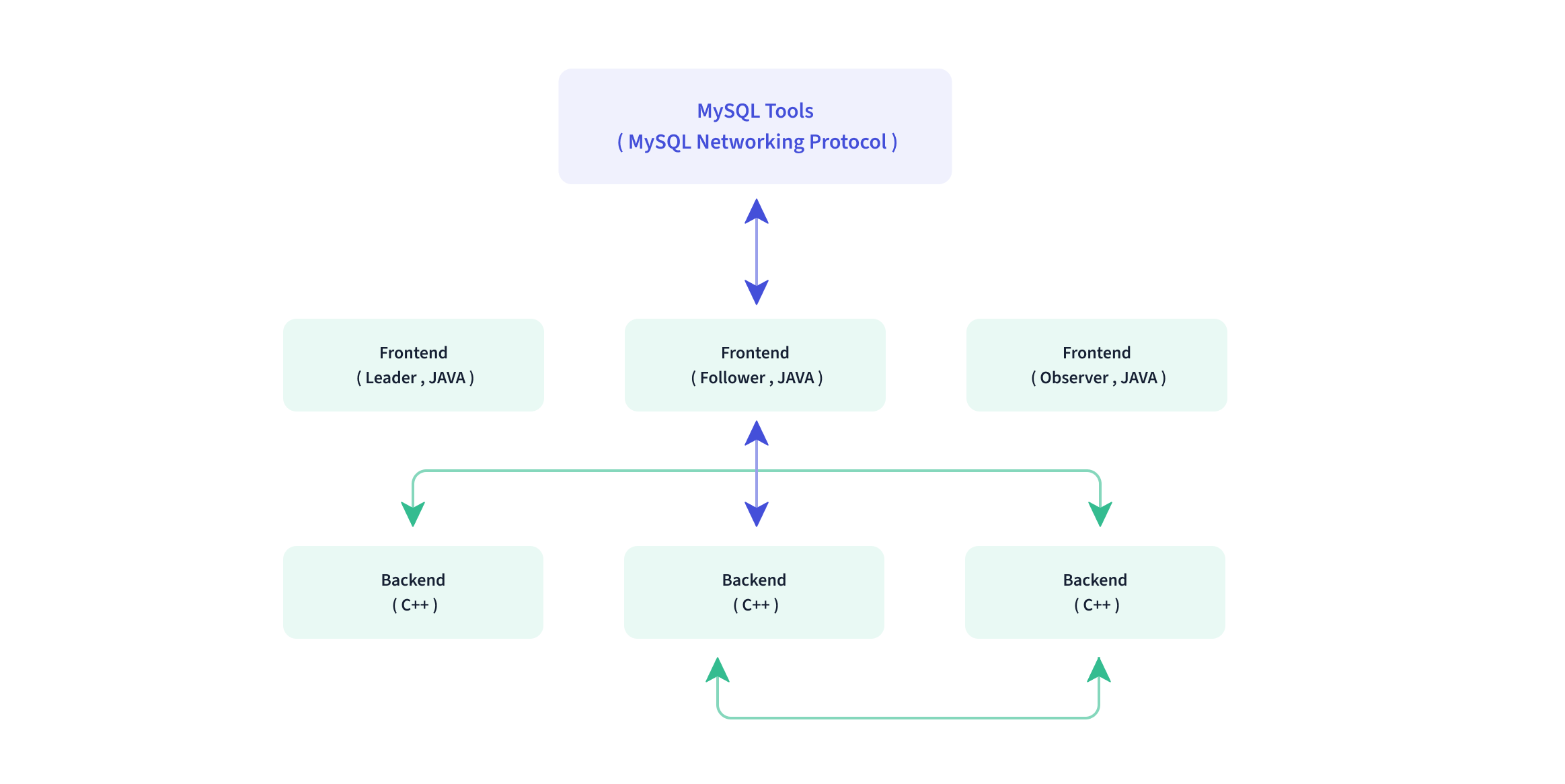


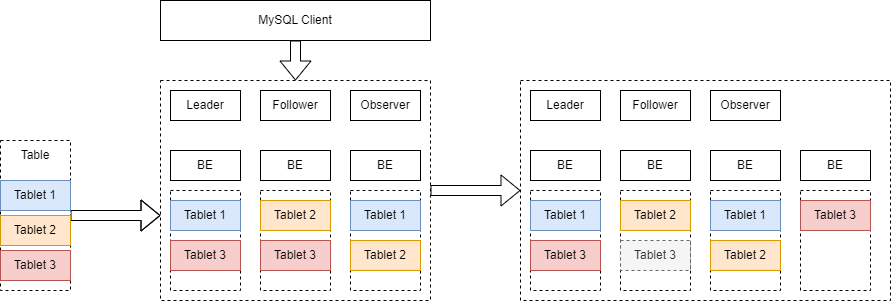
从上图中我们可以看到，数据源经过各种数据集成和加工后，通常会入库到实时数仓 Doris 之中。同时，部分数据会入到湖仓架构的 Hive 或 Iceberg 中，Doris会通过外表的方式联邦分析位于Hive、Iceberg、Hudi中的数据，在避免数据拷贝的前提下，查询性能大幅提升，然后，基于 Doris 构建一个统一的数仓，对外提供服务。

### Doris的易用性

Doris**整体架构**如下图所示，Doris 架构非常简单

* 高度兼容MySQL协议
* 主从架构，不依赖其他组件
* **Frontend（FE）**，主要负责用户请求的接入、查询解析规划、元数据的管理、节点管理相关工作。
* **Backend（BE）**，主要负责数据存储、查询计划的执行。
* 任何节点，都可以线性扩展
  + 多个FE节点之间通过BerkeleyDB java Edition进行Leader选举
  + Leader负责元数据的写入，Follower节点会同步元数据信息，如果Leader宕机，其余Follower会参与选举Leader
  + Observer不参与选举，只负责从Leader中同步元数据信息
  + 部署多个BE节点，可以使数据根据分片规则和副本数分配到各个BE节点上，实现高可用
  + 如果新增或者删除BE节点，Doris会自动完成数据迁移和副本均衡

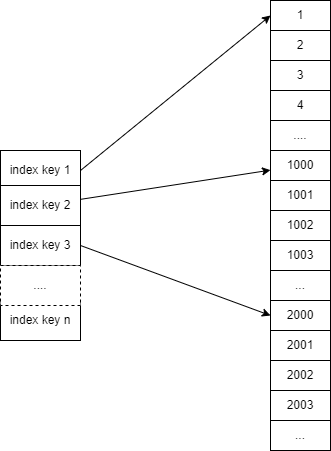
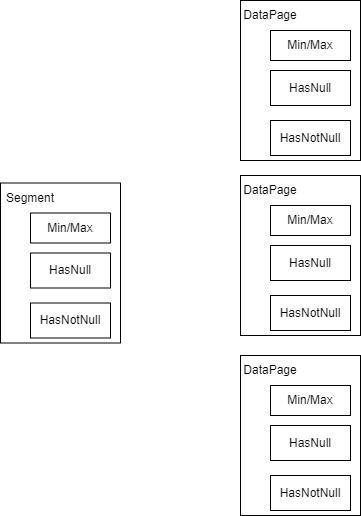
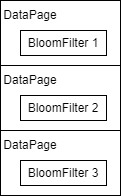
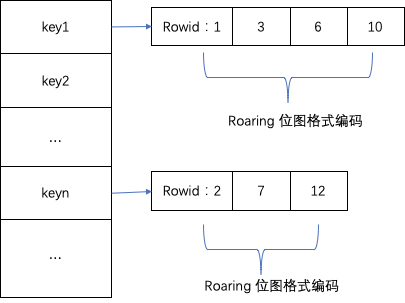




### Doris高性能

主要从三个方面来讲

1. 存储引擎
2. 查询引擎
3. 优化器

* Doris支持丰富的索引类型
  + 前缀稀疏索引：基于排序列查询的快速过滤
  + 
  + ZoneMap Index ：Segment 和每个列对应每个 Page 的统计信息
  + 
  + Bloom Filter ：对高基数列的等值过滤裁剪非常有效，Page级别
  + 
  + Invert Index ：基于BitMap，能够对任意字段实现快速检索
  + 
* Doris提供了三种**存储模型**
  + Aggregate Key 模型：相同 Key 的 Value 列合并，通过提前聚合大幅提升性能
  + Unique Key 模型：Key 唯一，相同 Key 的数据覆盖，实现行级别数据更新
  + Duplicate Key 模型：明细数据模型，满足事实表的明细存储
  + Doris 也支持强一致的物化视图，物化视图的更新和选择都在系统内自动进行，不需要用户手动选择，从而大幅减少了物化视图维护的代价。
* Doris在**查询引擎**方面，采用了**MPP**模型，节点间和节点内都并行执行，也支持多个大表的分布式 Shuffle Join。查询引擎**向量化**。
* 分区，分桶。SQL/Partition/Page Cache。
* **Doris 采用了 Adaptive Query Execution 技术，** 可以根据 Runtime Statistics 来动态调整执行计划，比如通过 Runtime Filter 技术能够在运行时生成 Filter 推到 Probe 侧，并且能够将 Filter 自动穿透到 Probe 侧最底层的 Scan 节点，从而大幅减少 Probe 的数据量，加速 Join 性能。Doris 的 Runtime Filter 支持 In/Min/Max/Bloom Filter。
* 在**优化器**方面 Doris 使用 CBO 和 RBO 结合的优化策略
  + RBO(基于规则)支持
    - 常量折叠
    - where event\_date > str\_to\_date("2023-04-06", "%Y-%m-%d %H:%i:%s")  
      ->  
      where event\_date > "2023-04-06 00:00:00"
    - 子查询改写
    - select \* from t1 where t1.col1 in (select col2 from t2)  
      ->  
      select a.\* from t1 a join t2 b on a.col1 = b.col2
    - 提取公因式
    - select \* from t where (a > 1 And b = 2) or (a > 1 and b = 3) or (a > 1 and b = 4)  
      ->  
      select \* from t where a > 1 and (b = 2 or b = 3 or b = 4)
    - 智能预过滤
    - select \* from t where (1 < a < 3 and b in ('a')) or (2 < a < 4 and b in ('b'))  
      ->  
      select \* from t where (1 < a < 4) and (b in ('a', 'b')) and ((1 < a < 3 and b in ('a')) or (2 < a < 4 and b in ('b')))
    - 谓词下推(先Filter再Join)
    - select \* from t1 a join t2 b on a.id = b.id where a.col1 > 100 and b.col2 < 300  
      ->  
      select \* from (select \* from t1 where t1.col1 > 100) a join (select \* from t2 where t2.col2 < 300) b on a.id = b.id
  + CBO(基于代价)
    - Join Reorder
    - Colocation Join
    - Bucket Join

### Doris数据导入导出

数据导入：

* Binlog Load: 官方文档上只写了MySQL导入的方式，需要使用Alibaba Canal组件配合使用。
* Broker Load: 是一个异步的导入方式，支持的数据源取决于Broker进程支持的数据源。
* Routine Load: 仅支持从 Kafka 进行例行导入。
* Spark Load:主要用于初次迁移，大数据量导入 Doris 的场景。
* Stream Load: 使用的是HTTP接口。
* MySQL Load: MySQL标准的LOAD DATA语法。
* S3 Load:导入方式与Broker Load基本相同。

数据导出：

* mysqldump
* select ... into outfile ...

### Doris数据湖分析

多源数据目录（Multi-Catalog）是 Doris 1.2.0 版本中推出的功能，旨在能够更方便对接外部数据目录，以增强Doris的数据湖分析和联邦数据查询能力。

原来版本的Doris只有Database -> Table的两层元数据层级，新版本多了一个Catalog层级。Catalog -> Database -> Table 的三层元数据层级。其中，Catalog 可以直接对应到外部数据目录。目前支持的外部数据目录包括：

1. Hive
2. Iceberg
3. Hudi
4. Elasticsearch
5. JDBC: 对接数据库访问的标准接口(JDBC)来访问各式数据库的数据。

该功能将作为之前外表连接方式（External Table）的补充和增强，帮助用户进行快速的多数据目录联邦查询。

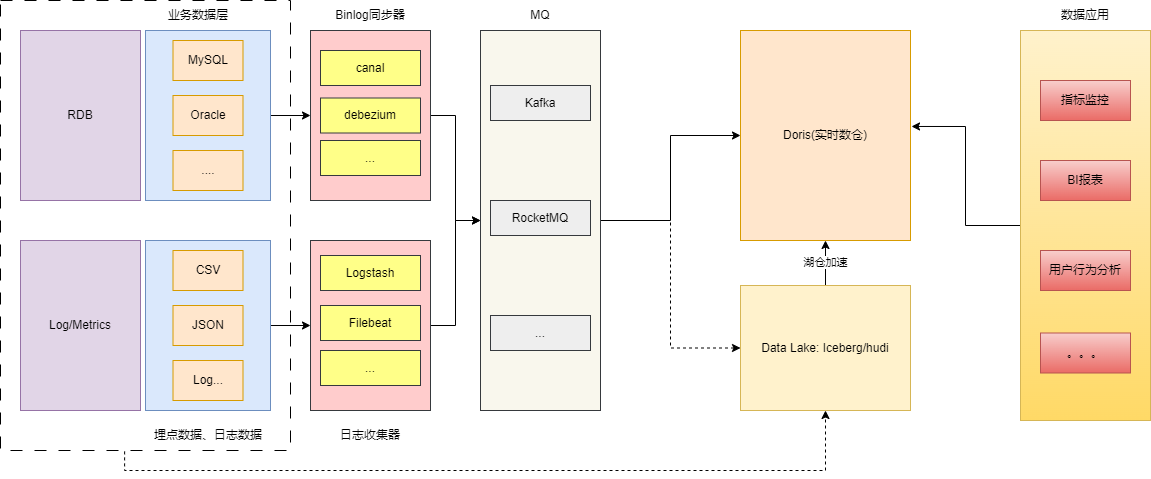
### Doris生态扩展

Doris提供了非常丰富的生态扩展，用户可以通过这些扩展，操作Doris，例如：

* Spark Doris Connector
  + 代码地址：https://github.com/apache/incubator-doris-spark-connector
  + 支持从Doris中读取数据
  + 支持Spark DataFrame批量/流式 写入Doris
  + 可以将Doris表映射为DataFrame或者RDD，推荐使用DataFrame。
  + 支持在Doris端完成数据过滤，减少数据传输量。
* Flink Doris Connector
  + 代码库地址：https://github.com/apache/doris-flink-connector
  + 可以支持通过 Flink 操作（读取、插入、修改、删除） Doris 中存储的数据。
  + 可以将 Doris 表映射为 DataStream 或者 Table。
  + 修改和删除只支持在 Unique Key 模型上
  + 目前的删除是支持 Flink CDC 的方式接入数据实现自动删除，如果是其他数据接入的方式删除需要自己实现。
* DataX doriswriter
  + 代码地址：https://github.com/apache/doris/tree/master/extension/DataX
  + 需要配合 DataX 服务一起使用。
* Seatunnel
  + Seatunnel Connector Flink Doris
  + Seatunnel Connector Spark Doris

## 如何构建实时数仓

简单了解Doris的基本概念，接下来，看一下如何基于 Apache Doris 构建极速易用的实时数仓的架构？



因为 Doris 既可以承载数据仓库的服务，也可以承载 OLAP 等多种应用场景。因此，它的实时数仓架构变得非常简单。

1. 我们通过Flink CDC/Canal/Debeziunm将RDS的数据或者通过Logstash/Filebeat将日志类数据实时同步到Kafka中。
2. 通过Doris的Routine load 将 Kafka 等消息系统中的数据，实时同步到 Doris。当然，也可以使用Flink Doris Connector/Seatunnel Connector Flink Doris实时同步到Doris。
3. 在 Doris 内部，基于 Doris 不同的表模型、Rollup、以及物化视图的能力，构建实时数仓。
   * ODS 层通常会使用明细模型构建
   * DWD 层通过 SQL 调度任务，对 ODS 数据抽取并获取
   * DWS 和 ADS 层则可以通过 Rollup 和物化视图的技术手段构建。
4. Doris 还支持基于 Iceberg、Delta Lake 和 Hudi 的数据湖服务，提供一些联邦分析和湖仓加速的能力。

这样我们便完成了基于 Doris 构建一个实时数仓。在实时数仓之上，我们可以构建 BI 服务、Adhoc 查询、多维分析等应用。

### 如何保证一致性

Flink CDC 通过 Flink Checkpoint 机制结合 Doris 两阶段提交可以实现端到端的 Exactly Once 语义。具体过程分为四步：

* 事务开启(Flink Job 启动及 Doris 事务开启)：当 Flink 任务启动后， Doris 的 Sink 会发起 Precommit 请求，随后开启写⼊事务。
* 数据传输(Flink Job 的运⾏和数据传输)：在 Flink Job 运⾏过程中， Doris Sink 不断从上游算⼦获取数据，并通过 HTTP Chunked 的⽅式持续将数据传输到 Doris。
* 事务预提交：当 Flink 开始进⾏ Checkpoint 时，Flink 会发起 Checkpoint 请求，此时 Flink 各个算⼦会进⾏ Barrier 对⻬和快照保存，Doris Sink 发出停⽌ Stream Load 写⼊的请求，并发起⼀个事务提交请求到 Doris。这步完成后，这批数据已经完全写⼊ Doris BE 中，但在 BE 没有进⾏数据发布前对⽤户是不可⻅的。
* 事务提交：当 Flink 的 Checkpoint 完成之后，将通知各个算⼦，Doris 发起⼀次事务提交到 Doris BE ，BE 对此次写⼊的数据进⾏发布，最终完成数据流的写⼊。

当预提交成功，但 Flink Checkpoint 失败时，该怎么办？这时 Doris 并没有收到事务最终的提交请求，Doris 内部会对写入数据进行回滚（rollback），从而保证数据最终的一致性。

### 如何DDL 和 DML 同步

随着业务的发展，部分⽤户可能存在 RDS Schema 的变更需求。当 RDS 表结构变更时，⽤户期望 Flink CDC 不但能够将数据变化同步到 Doris，也希望将 RDS 表结构的变更同步到 Doris，⽤户则无需担⼼ RDS 表结构和 Doris 表结构不⼀致的问题。

Hive如果对数据表的Schema进行新增或者删除，都会触发数据同步，就会导致修改Schema的效率非常低。

Doris引入了Light Schema Change来解决这个问题，Light Schema Change 只修改了 FE 的元数据，并没有同步给 BE。但这样就有可能产生FE和BE的Schema不一致，我们对 BE 的写出流程进⾏了修改，具体包含三个⽅⾯。

* 数据写⼊：FE 会将 Schema 持久化到元数据中，当 FE 发起导⼊任务时，会把最新的 Schema 一起发给 Doris BE，BE 根据最新的 Schema 对数据进⾏写⼊，并与 RowSet 进⾏绑定。将该 Schema 持久化到 RowSet 的元数据中，实现了数据的各⾃解析，解决了写⼊过程中 Schema 不⼀致的问题。
* 数据读取：FE ⽣成查询计划时，会把最新的 Schema 附在其中⼀起发送给 BE，BE 拿到最新的 Schema 后对数据进⾏读取，解决读取过程中 Schema 发⽣不⼀致的问题。
* 数据 Compaction：当数据进⾏ Compaction 时，我们选取需要进⾏ Compaction 的 RowSet 中最新的 Schema 作为之后 RowSet 对应的 Schema，以此解决不同 Schema 上 RowSet 的合并问题。

经过对 Light Schema Change 写出流程的优化后， 单个 Schema Chang 从 **310 毫秒降低到了 7 毫秒**，整体性能有**近百倍的提升**，彻底的解决了海量数据的 Schema Change 变化难的问题。

有了 Light Schema Change 的保证， Flink CDC 能够同时⽀持 DML 和 DDL 的数据同步。那么是如何实现的呢？

* 开启 DDL 变更配置：在 Flink CDC 的 MySQL Source 侧开启同步 MySQL DDL 的变更配置，在 Doris 侧识别 DDL 的数据变更，并对其进⾏解析。
* 识别及校验：当 Doris Sink 发现 DDL 语句后，Doris Sink 会对表结构进⾏验证，验证其是否⽀持 Light Schema Change。
* 发起 Schema Change ：当表结构验证通过后，Doris Sink 发起 Schema Change 请求到 Doris，从⽽完成此次 Schema Change 的变化。

解决了数据同步过程中源数据⼀致性的保证、全量数据和增量数据的同步以及 DDL 数据的变更后，一个完整的数据同步⽅案就基本形成了。

### 如何选择数据模型和数仓分层

一般来说，数仓分层主要分为四层：ODS，DWD，DWS，ADS

* ODS为原始数据层：存放未经过处理的原始数据至数据仓库，结构上与源系统保持一致，是数据仓库的数据准备区。对于Doris而言，ODS层通常会采用Duplicate Key模型。
* DWD为数据明细层：该层一般保持和ODS层一样的数据粒度，并且提供一定的数据质量保证。该层可能也会做一部分的聚合操作，提高数据可用性。从ODS 到 DWD 层数据的抽取，一般会采用微批调度调度的方法，需要借助外部的调度工具例如 DolphinScheduler 或 Airflow 等来对 ETL SQL 进行调度。
* DWS为数据服务层：又称数据集市或宽表。按照业务划分，如流量、订单、用户等，生成字段比较多的宽表，用于提供后续的业务查询，OLAP分析，数据分发等。这一层，我们可以选择Aggregate Key模型或者在Base表上，创建不同的Rollup或物化视图对Base表进行聚合计算
* ADS为数据应用层：存放数据产品个性化的统计指标数据。根据DM层与ODS层加工生成。这一层，我们可以选择Aggregate Key模型或者在Base表上，创建不同的Rollup或物化视图对Base表进行聚合计算

# Demo Code

## Routine Load

1. 创建一张user表

* CREATE TABLE IF NOT EXISTS user  
  (  
   id bigint null,  
   username varchar(256) null,  
   password varchar(256) null,  
   createTime datetime not null  
  )  
  DUPLICATE KEY(`id`, `username`)  
  DISTRIBUTED BY HASH(`id`) BUCKETS 1;

1. 创建Routine Load

* CREATE ROUTINE LOAD shawn\_test.user\_routine\_load\_test ON user  
  COLUMNS(id, username, password, createTime)  
  PROPERTIES  
  (  
   "desired\_concurrent\_number"="2",  
   "max\_batch\_interval" = "20",  
   "max\_batch\_rows" = "200000",  
   "max\_batch\_size" = "209715200",  
   "strict\_mode" = "false",  
   "format" = "json"  
  )  
  FROM KAFKA  
  (  
   "kafka\_broker\_list" = "slave02:9092",  
   "kafka\_topic" = "user",  
   "property.kafka\_default\_offsets" = "OFFSET\_BEGINNING"  
   );

1. kafka造数

* public class DataFaker {  
   public static void main(String[] args) throws Exception {  
   Map<String, Object> map = new HashMap<>();  
   map.put(BOOTSTRAP\_SERVERS\_CONFIG, "slave02:9092");  
   map.put(KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName());  
   map.put(VALUE\_SERIALIZER\_CLASS\_CONFIG, JsonSerializer.class.getName());  
   KafkaProducer<String, User> kafkaProducer = new KafkaProducer<>(map);  
   AtomicLong atomic = new AtomicLong();  
   for (int i = 0; i < 100; i++) {  
   long id = atomic.incrementAndGet();  
   ProducerRecord<String, User> record =  
   new ProducerRecord<>(  
   "user",  
   new User(  
   id,  
   "user" + id,  
   "user" + id,  
   DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss").format(LocalDateTime.now())));  
   kafkaProducer.send(  
   record,  
   (metadata, e) -> {  
   if (e == null) {  
   System.out.printf(  
   "topic: %s, partition %s, offset: %s\n",  
   metadata.topic(), metadata.partition(), metadata.offset());  
   } else {  
   e.printStackTrace();  
   }  
   });  
   TimeUnit.SECONDS.sleep(3);  
   }  
   kafkaProducer.close();  
   }  
  }

1. 查看user表里的数据

## Stream Load

1. 添加pom.xml

* <dependency>
    
   <groupId>org.apache.httpcomponents</groupId>
    
   <artifactId>httpclient</artifactId>
    
   <version>4.5.3</version>
    
  </dependency>

1. 参考代码

* public class KafkaToDorisStreamLoad implements KafkaToDorisLoader {
    
    
   private static final String FE\_HOST = "192.168.54.150";
    
   private static final String FE\_HTTP\_PORT = "8030";
    
   private static final String USER = "root";
    
   private static final String DB = "shawn\_test";
    
   private static final String TABLE = "user\_stream\_load";
    
   private static final String URL = "http://%s:%s/api/%s/%s/\_stream\_load";
    
    
   // Build http client builder
    
   private static final HttpClientBuilder httpClientBuilder =
    
   HttpClients.custom()
    
   .setRedirectStrategy(
    
   new DefaultRedirectStrategy() {
    
   @Override
    
   protected boolean isRedirectable(String method) {
    
   // If the connection target is FE, you need to deal with 307 redirect。
    
   return true;
    
   }
    
   });
    
    
   @Override
    
   public void load(String json) throws IOException {
    
   try (CloseableHttpClient client = httpClientBuilder.build()) {
    
   HttpPut put = new HttpPut(String.format(URL, FE\_HOST, FE\_HTTP\_PORT, DB, TABLE));
    
   put.removeHeaders(HttpHeaders.CONTENT\_LENGTH);
    
   put.removeHeaders(HttpHeaders.TRANSFER\_ENCODING);
    
   put.setHeader(HttpHeaders.EXPECT, "100-continue");
    
   put.setHeader(HttpHeaders.AUTHORIZATION, basicAuthHeader(USER, ""));
    
    
   // You can set stream load related properties in the Header, here we set label and
    
   // column\_separator.
    
   put.setHeader("label", UUID.randomUUID().toString());
    
   put.setHeader("column\_separator", ",");
    
   put.setHeader("format", "json");
    
   put.setHeader("columns", "id,username,password,createTime");
    
    
   // Set up the import file. Here you can also use StringEntity to transfer arbitrary data.
    
   StringEntity entity = new StringEntity(json);
    
   put.setEntity(entity);
    
    
   try (CloseableHttpResponse response = client.execute(put)) {
    
   String loadResult = "";
    
   if (response.getEntity() != null) {
    
   loadResult = EntityUtils.toString(response.getEntity());
    
   }
    
    
   final int statusCode = response.getStatusLine().getStatusCode();
    
   if (statusCode != 200) {
    
   throw new IOException(
    
   String.format(
    
   "Stream load failed. status: %s load result: %s", statusCode, loadResult));
    
   }
    
   System.out.println("Get load result: " + loadResult);
    
   }
    
   }
    
   }
    
    
   /\*\*
    
   \* Construct authentication information, the authentication method used by doris here is Basic
    
   \* Auth
    
   \*
    
   \* @param username
    
   \* @param password
    
   \* @return
    
   \*/
    
   private String basicAuthHeader(String username, String password) {
    
   final String tobeEncode = username + ":" + password;
    
   byte[] encoded = Base64.encodeBase64(tobeEncode.getBytes(StandardCharsets.UTF\_8));
    
   return "Basic " + new String(encoded);
    
   }
    
    
   public static void main(String[] args) throws Exception {
    
   KafkaToDorisLoader loader = new KafkaToDorisStreamLoad();
    
   AtomicLong atomicLong = new AtomicLong();
    
   JsonMapper mapper = JsonMapper.builder().findAndAddModules().build();
    
   List<User> users = new ArrayList<>();
    
   for (int i = 0; i < 100; i++) {
    
   long id = atomicLong.incrementAndGet();
    
   User user =
    
   new User(
    
   id,
    
   "user" + id,
    
   "user" + id,
    
   DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss").format(LocalDateTime.now()));
    
   users.add(user);
    
   }
    
   String s = mapper.writeValueAsString(users);
    
   loader.load(s);
    
   }
    
  }

## Flink

https://doris.apache.org/zh-CN/docs/ecosystem/flink-doris-connector/

https://doris.apache.org/zh-CN/docs/ecosystem/seatunnel/flink-sink/

## Spark

https://doris.apache.org/zh-CN/docs/ecosystem/spark-doris-connector/

https://doris.apache.org/zh-CN/docs/ecosystem/seatunnel/spark-sink/

## JDBC

https://www.cnblogs.com/lshan/p/14792632.html

# 机场数据中台接入可行性分析

## 现状

目前机场的数据，一条记录大概1-2kb，每天如果生产10w条数据，一年3650w条数据，总大小7.3GB左右。

## 讨论

就目前而言，数据量并不大，并发量也不大，平均一秒产生一条数据，这与Doris的设计理念不是那么符合。Doris是基于 一个MPP 架构的高性能、实时的分析型数据库，专门用来海量数据的分布式计算的数据库，除了存储，还需要做计算。

如果单纯使用JDBC插入，生产上不建议使用insert into table (col1, col2, col3, ...) values (?,?,?, ...)语法，执行一条就是开启一个事务，性能很差。

Stream Load是一个同步操作，Routine Load的底层也是通过Stream Load的方式进行的，如果需要保证Exactly once需要用户手动开启两阶段事务提交，另外官方建议单次导入的数据量最好在1G-10G之间。

Routine Load也会存在一些问题，数据导入到kafka，Doris通过Routine Load的方式将kafka里的消息导入到Doris里去，性能方面受限于Stream Load，主要问题是Routine Load的Task如果出现了故障，不会进行故障转移，会造成数据丢失。

## 其他解决方案

### MySQL优化

1. 建立必要的索引
2. 查询语句优化，比如谓词下推，少用Join，避免索引失效等
3. 分区，分表，分库，主备

### 架构优化

1. 优先引入Hadoop，将ODS层的数据不持久化到MySQL，直接批量写入到HDFS里。
2. 使用flink做数仓分层，ODS->DW-ADS，全部使用flink做数据计算任务。例如上面说的少用Join，可以先使用flink SQL将join出来的结果sink到ADS，这样数据服务可以直接通过单表查询，性能也会提升很多。
3. ADS可以写入到MySQL一份，也可以写入到HDFS里一份。
4. ADS层一般会有大量的过去的数据，是静态，可以通过Datax/flink等手段，将静态数据做一次计算。比如报表，需要显示2021年、2022年、2023年的统计信息，可以把2021和2022的统计信息存储起来，这样只要计算2023年的就行了，避免很多重复计算。
5. Flink可以运行在YARN上，通过YARN的ResourceManager做资源的分配。

如果还有更好的解决方案，欢迎大家一起来讨论学习！

# 参考文献

[Apache Doris官方网站](https://doris.apache.org/zh-CN/docs/dev/summary/basic-summary/)

[如何基于 Apache Doris 与 Apache Flink 快速构建极速易用的实时数仓](https://juejin.cn/post/7212500870763020346#heading-5)