大数据专题-汇总文档-周彬韬

作业1:数据仓库与OLAP调研报告

姓名：周彬韬

学号：2001210564

1. 前言

课程大作业打算做一个数据仓库的项目，因此调研围绕着数据仓库来展开。本调研报告主要分为七部分内容：

- 第一部分：前言，即目录与各章节简介。
- 第二部分：关于数据仓库的简介，包括其概念、特点、架构等简介。
- 第三部分：调研阅读Hive，论文《Apache Hive: From MapReduce toEnterprise-grade Big Data Warehousing》https://dl.acm.org/doi/pdf/10.1145/3299869.3314045，了解大数据下的数仓与Hive作为企业级数据仓库最近几年的优化。
- 第四部分：调研阅读Kafka与RabbitMQ，论文《Industry Paper: Kafka versus RabbitMQ》https://dl.acm.org/doi/pdf/10.1145/3093742.3093908，比较两种消息队列，其中kafka详细介绍，RabbitMQ简略介绍，有助于后续架构设计与项目技术选型。
- 第五部分：调研阅读SnowFlake，论文《Snowflake Elastic Data WareHouse》https://dl.acm.org/doi/pdf/10.1145/2882903.2903741 提出了一种新型云原生、OLAP、分布式数据仓库，以此窥见数仓的未来演变方向。
- 第六部分：项目思路、技术栈与预期目标。
- 第七部分：参考文献。

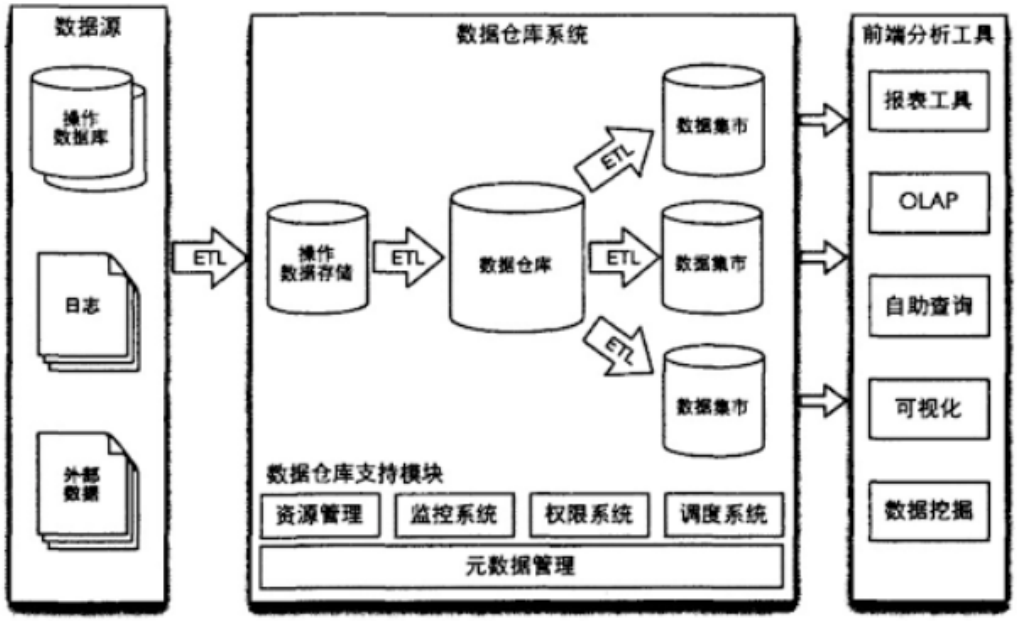
☐ 声明：部分材料源自本人博客 https://blog.csdn.net/weixin_39666736

---

2. 数据仓库简介

2.1 概念

数据仓库（data warehouse）是用于报告和数据分析的系统，被认为是商业智能（business intelligence-BI）的核心，**数据仓库是来自一个或多个不同源的集成数据的中央存储库**。数据仓库将当前和历史数据存储在一起，以便分析性报告和决策支持。



2.2 特点

- 面向主题：

数据仓库中的所有数据一般按照主题划分，从较高层次上对信息系统的数据进行归纳和整理，同一主题下包含多个数据源，主题是对业务数据的一种抽象。

- 集成

各个数据源的数据库往往是异构的、并且相互独立，反应的信息是局部的，所以需要对数据进行整合，而数据仓库中的数据是经过数据的抽取、清洗、切换、加载（Extract, Transform, Load - ETL）得到的，所以为了保证数据不存在二义性，**必须对数据进行编码统一和必要的汇总，以保证数据仓库内数据的一致性，这是数据仓库建设过程中，最关键、最复杂的一步**。数据仓库在经历数据集成阶段后，使数据仓库中的数据都遵守统一的编码规则，并且消除许多冗余数据。

- 不可更新

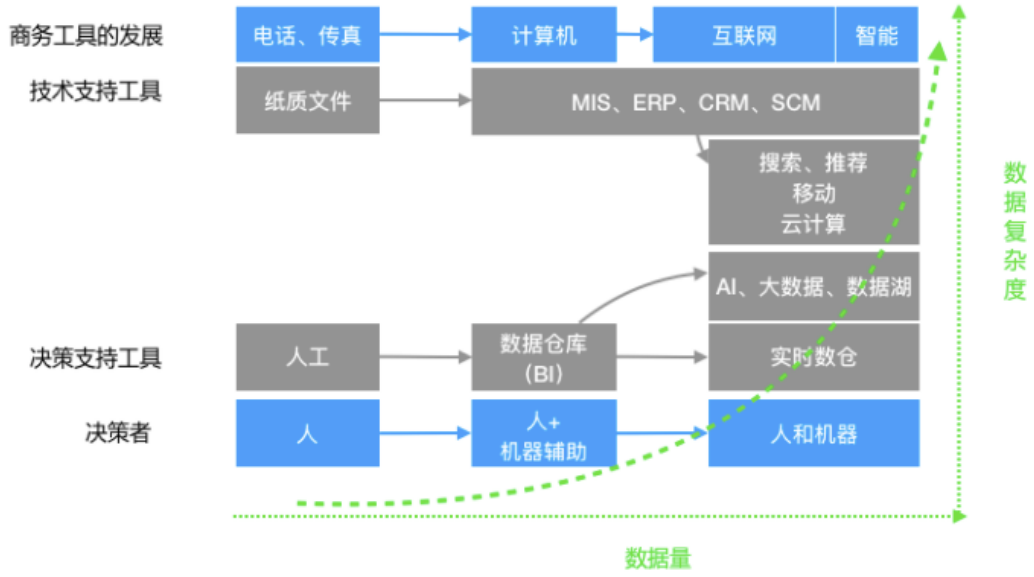
一旦某个数据进入到数据仓库后，一般情况下数据会被长期保留，超过规定的期限才会被删除，**数据仓库中的数据反映的都是一段历史时期的数据内容**，它的主要操作是查询、分析而一般不进行更新。

- 随时间变化

数据仓库随着时间不断增加新的数据内容，同时也不断删除过期的旧数据，因此数据仓库的数据特征都包含时间项

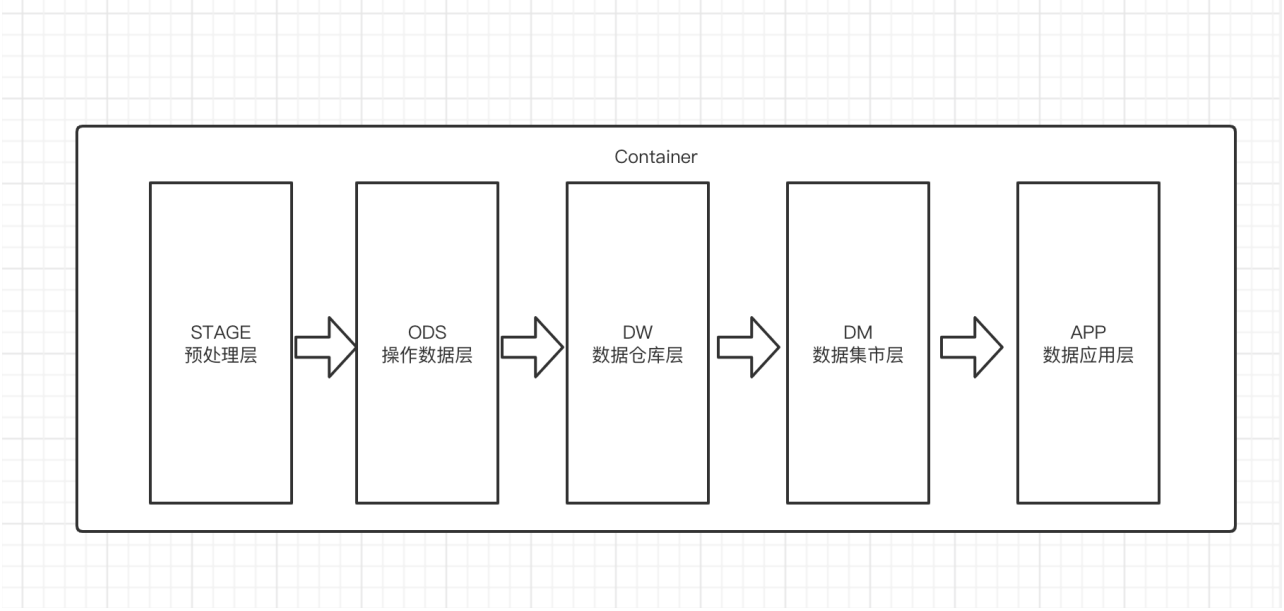
2.3 数据仓库的作用

- 统一的数据中心
- 为运营人员提供数据支持
- 为领导提供报表，以支持决策



2.4 数仓分层

- STAGE，预处理层，存储每天的增量数据
- ODS，操作数据层，做数据清洗，存储基础原始明细数据
- DW，数据仓库层，一般采用维度、事实表设计
- DM，数据集市层，宽表化设计，形成公共指标，在 DW层的基础之上根据不同的业务需求做轻度汇总所得
- APP，数据应用层，一般报表数据就存放在这里，面向最终展示



2.5 数仓架构演变

- 传统数仓架构

传统的数仓结构化、半结构化数据通过离线ETL定期加载到离线数仓中，之后通过计算引擎取得结果

- Lambda架构

随着业务的发展，人们对数据实时性提出了更高的要求，出现了Lambda架构，**它解决了大数据批量离线处理和实时数据处理的需求，其将对实时性要求高的部分拆分出来，增加一条实时链路**，将数据发送到消息队列中，实时流数据与离线批数据并行运行，最终由统一的数据服务层合并到前端。

缺点：Lambda架构，一个比较严重的问题是需要维护两套逻辑，一部分在批量引擎实现，一部分在流式引擎实现。

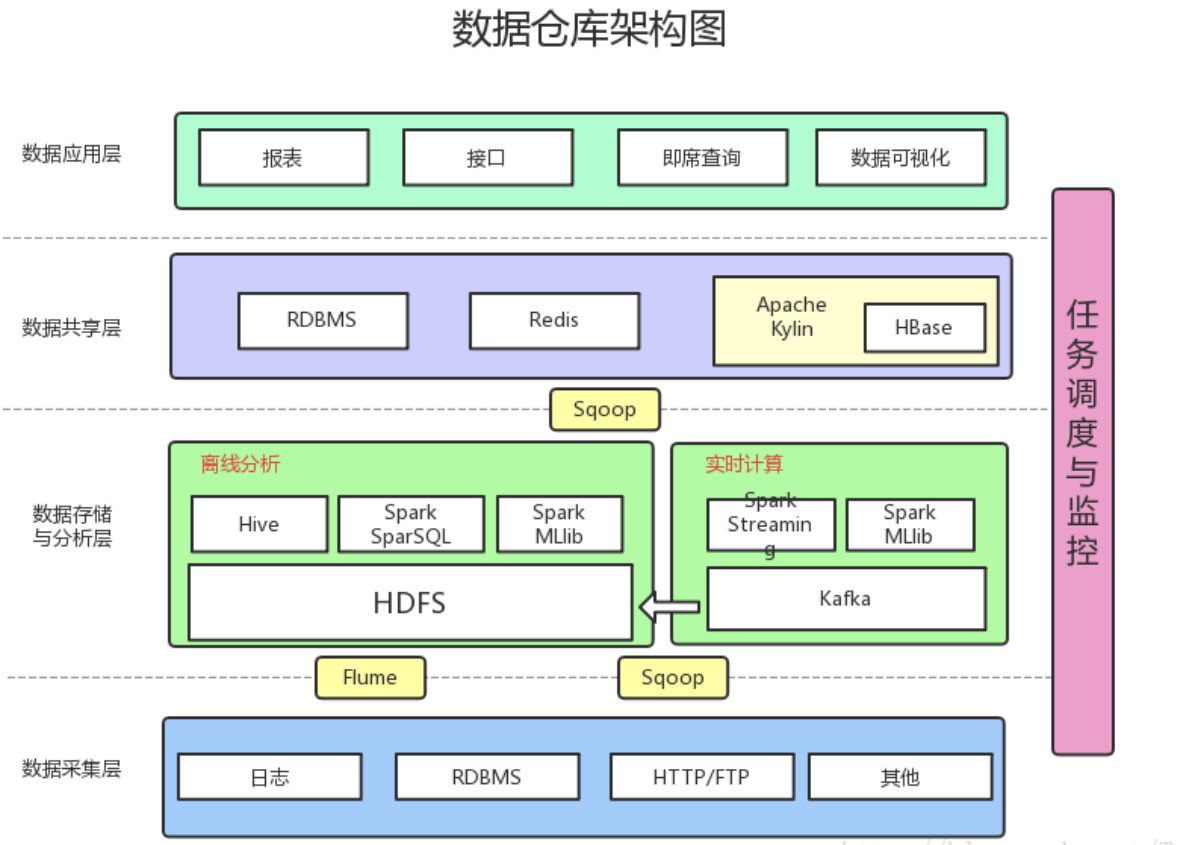
- Kappa架构

Kappa架构核心是用消息队列收集各种数据，当需要全量重新计算时，重新起一个流计算实例，**Kappa架构的优点是将实时和离线代码统一起来**，缺点是流式处理对于历史数据高吞吐量力不从心，并且数据量一大很难满足数据查询即时响应的要求，并且由于采集数据的格式不统一，每次需要开发不同的streaming程序

- IOTA架构

大数据IOTA架构思路是设定标准数据模型，通过边缘计算把所有计算过程分散在数据产生、计算、查询的过程中，满足即时查询的需要

当今主流公司用的数据仓库架构图



2.6 数仓未来发展趋势

- 实时

  未来企业对数据实时性要求越来越高，之后传统的离线数仓会消失，实时方式会成为主流

- 海量

  数仓装载的数据会越来越多，TB、PB级别

- 多模

  半结构化、非结构化数据更多将会出现在数仓领域，湖仓一体

- 治理

  数仓除了传统的数据存储、计算能力外，还需要提供完备的数据治理能力，包括元数据、数据血缘、数据质量等

- 智能

  数据使用上，AI将会赋能数据应用，赋能数仓

---

3. 数仓Hive

3.1 Hive简介

3.1.1 Hive是什么?

大数据主要解决海量数据的三类问题：**传输问题、存储问题、计算问题**，而Hive主要解决存储和计算问题。

Hive是基于Hadoop构建的一套数据仓库分析系统，它提供了丰富的SQL查询方式来分析存储在Hadoop分布式文件系统中的数据；

可以将结构化的数据文件映射为一张数据库表，并提供完整的SQL查询功能；

可以将SQL语句转换为MapReduce任务运行，通过自己的SQL查询分析需要的内容，这套SQL简称Hive SQL，使不熟悉mapreduce的用户可以很方便地利用SQL语言查询、汇总和分析数据。

Hive本质是将HQL转化为 MapReduce程序。

3.1.2 优缺点

优点：

- Hive 封装了一层接口，并提供类 SQL 的查询功能，避免去写 MapReduce，减少了开发人员的学习成本
- 适合处理大数据
- 容错性：可以保障即使有节点出现问题，SQL 语句也可以完成执行

- 可拓展性强：可以自由扩展集群的规模，不需要重启服务而进行水平拓展

缺点：

- Hive 延迟较高，不适用于实时分析
- Hive 不支持事物，因为没有增删改，所以主要用来做 OLAP（联机分析处理），而不是 OLTP（联机事物处理）
- Hive 自动生成的 MapReduce 作业，通常情况下不够智能
- Hive 不支持记录级别的增删改操作，但是可以通过查询创建新表来将结果导入到文件中；（hive 2.3.2 版本支持记录级别的插入操作）

3.1.3 hive与传统数据库对比

	Hive	RDBS
查询语言	HQL	SQL
数据存储	HDFS	Local FS
执行	MapReduce	excutor
延迟	高	低
索引	位图索引	复杂索引

3.1.4 论文introduction

《Apache Hive: From MapReduce toEnterprise-grade Big Data Warehousing》简介：

以前提交给研究界的关于HIVE的工作主要集中在（i）其在HDFS和MapReduce基础上的初始架构和实现（ii）改进以解决原始系统中的多个性能缺陷，包括引入优化的列文件格式，物理优化，以减少MapReduce阶段执行数目，和一种矢量化执行顺序来提高运行效率。本文描述了在上一篇文章之后Hive引入的一些新特性，特别是，它注重改造工作，分为以下四个方面：

SQL和ACID支持（3.3）SQL遵从性是数据仓库的核心要求。SQL支持得到了扩展，包括相关子查询、完整性约束和扩展OLAP操作等。反过来，仓库用户需要支持在上插入、更新、删除和合并他们的个人记录需求。Hive通过使用构建在Hive元存储之上的事务管理器提供快照隔离，从而提供ACID保证

优化技术（3.4）查询优化特别适用于使用声明性查询语言（如SQL）的数据管理系统。Hive没有从头开始实现自己的优化器，而是选择与Calcite集成，并将其优化功能带到系统中。此外，Hive还包括数据仓库环境中常用的其他优化技术，如查询重新优化、查询结果缓存和物化视图重写

运行时延迟（3.5）为了覆盖更广泛的用例，包括交互式查询，提高延迟是至关重要的。Hive支持优化柱状数据存储和运算符矢量化。除了这些改进之外，Hive已经从MapReduce迁移到了Tez，Tez它比其他运行时更灵活地实现任意数据处理应用程序MapReduce。此外，Hive包括LLAP，这是一个额外的层，由提供数据缓存的长时间运行的执行器组成，提供运行时优化，并避免启动时的YARN分配开销。

联合功能（3.6）Hive最重要的功能之一是在过去几年出现的许多专用数据管理系统之上提供统一的SQL层。由于Calcite集成和存储处理程序的改进，Hivecan可以无缝地推动计算并从这些系统读取数据。反过来，该实现很容易扩展，以支持将来的其他系统

3.2 Hive架构和组件

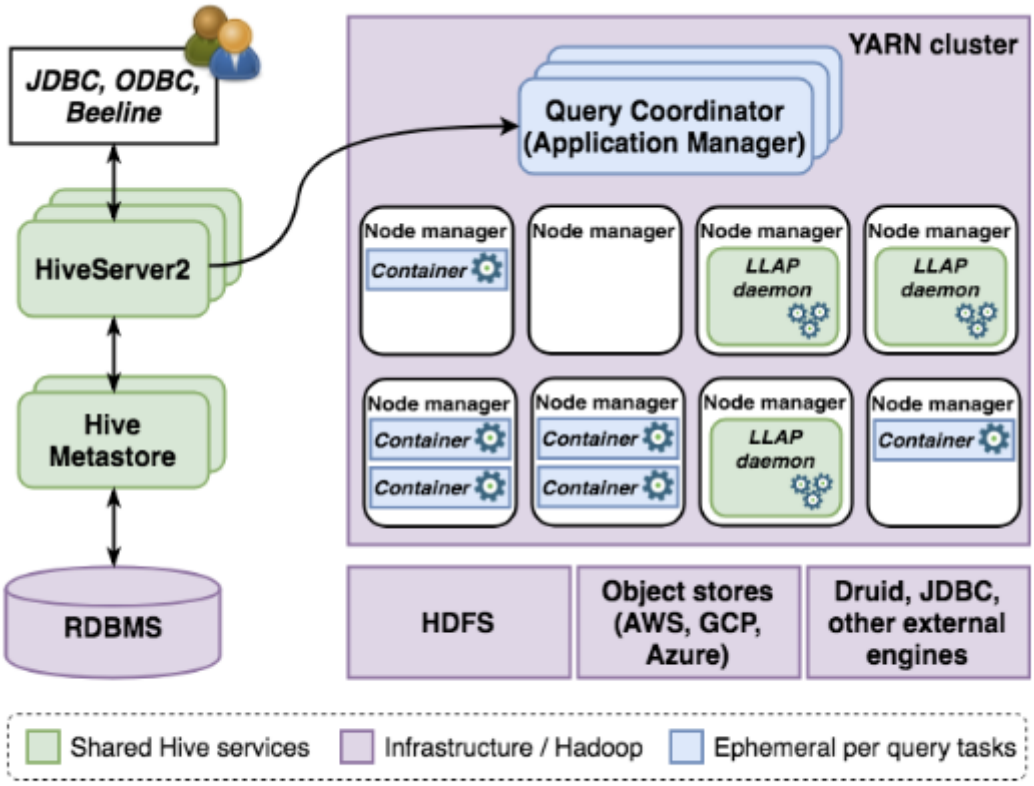


Figure 1: Apache Hive architecture.

- 用户接口

1.hive shell

2.JDBC （java访问hive）

3.WebUI（浏览器访问hive）

- 元数据 metastore

元数据包括：表名，表所属的数据库（Derby、MySQL），表的拥有者，列、分区字段、表的默认类型、表数据所在目录

- 驱动器 Drive

1. 解析器（SQL Parser）：将SQL字符串转换成抽象语法树AST，这一步一般都用第三方工具库完成，比如antlr；对AST进行语法分析，比如表是否存在、字段是否存在、SQL语义是否有误。

2. 编译器（Physical Plan）：将AST编译生成逻辑执行计划。

3. 优化器（Query Optimizer）：对逻辑执行计划进行优化。

4. 执行器（Execution）：把逻辑执行计划转换成可以运行的物理计划。对于Hive来说，就是MR/Spark。

数据在hive中可以使用任何支持的文件格式存储在任何与hadoop兼容的文件系统中，最常见的文件格式是orc和parquet，兼容的文件系统包括最常见的AWS、S3和Azure Blob，hive还可以将数据读写到其他stand-alone系统中，如Druid和Hbase。

Data catalog，Hive将所有的数据源元信息存储在Hive metastore中，简称HMS，它使用RDBMS来持久化信息

Exchangeable data processing runtime.Hive已经成为Hadoop上最流行的SQL引擎之一，它逐渐远离MapReduce，支持与YARN兼容的更灵活的处理运行时，使用Tez代替MapReduce，Tez将数据处理建模为代表应用程序逻辑的顶点和代表数据传输边的DAG，Tez与LLAP兼容

Query server，HiveServer2（简称HS2）允许用户在Hive中执行SQL查询。HS2支持本地和远程JDBC和ODBC连接

下图描述了SQL查询在HS2中成为可执行计划的各个阶段。一旦用户向HS2提交了一个查询，该查询就由驱动程序处理，驱动程序解析语句并从其AST生成一个Calcite逻辑计划。然后优化calcite方案。请注意，HS2访问HMS中有关数据源的信息以进行验证和优化。随后，该计划被转换为一个物理计划，可能引入额外的运算符来进行数据分区、排序，等HS2对物理计划执行额外的优化，如果计划中的所有运算符和表达式都得到支持，则可以从中生成矢量化计划。物理计划被传递给任务编译器，任务编译器将操作符树分解为可执行文件的DAG任务。配置单元为每个支持的处理运行时实现一个单独的任务编译器，即Tez、Spark和MapReduce。之后生成任务后，驱动程序将其提交给运行时应用程序管理器，负责执行。对于每个任务，首先初始化该任务中的物理操作符，然后它们以流水线方式处理输入数据。执行完成后，driverfetch将查询结果返回给用户

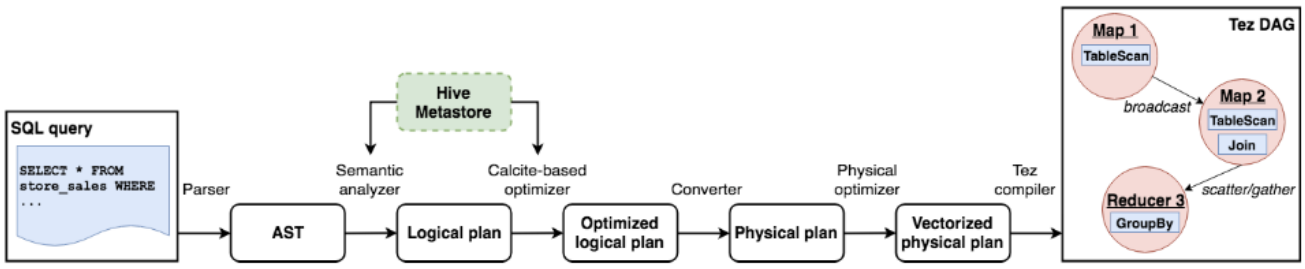


Figure 2: Query preparation stages in HiveServer2.

3.3 SQL 与 ACID支持

3.3.1 SQL 支持

标准SQL和ACID事务是企业数据仓库中的关键需求。在本节中，我们将介绍Hive的扩展SQL支持。此外，我们还描述了对Hive的改进，以便在Hadoop之上提供ACID保证

为了提供传统数据仓库的代替，Hive需要拓展以适应更多标准化SQL，Hive使用了一个嵌套的数据模型，支持所有主要的原子SQL数据类型以及非原子类型，如STRUCT、ARRAY和MAP，此外，每个新Hive Release重新租用都增加了对重要构造的支持，这些构造是SQL规范的一部分。例如，扩展了对相关子查询的支持，即引用外部查询中的列的子查询、高级olap操作（如分组集或窗口函数）、setoperations和完整性约束等。另一方面，Hive保留了对用户有价值的原始查询语言的多个特性一号基地最流行的特性之一是能够在表创建时使用apartifiedbycolumns子句指定物理存储布局。简而言之，子句将用户水平地划分一个表。然后，配置单元将每组分区值的数据存储在文件系统的不同目录中。为了说明这个想法，请考虑下面的表定义和中描述的相应物理布局

```
1 CREATE TABLE store_sales (
2   sold_date_sk INT, item_sk INT, customer_sk INT, store_sk INT,
3   quantity INT, list_price DECIMAL(7,2), sales_price DECIMAL(7,2)
4 ) PARTITIONED BY (sold_date_sk INT);
```

使用partitioned by子句的优点是，对于筛选那些值的查询，hive可以轻松跳过对完整分区的扫描

3.3.2 ACID实施

最初，Hive只支持从表中插入和删除完整分区。尽管缺少行级操作对于ETL工作负载来说是可以接受的，但是随着Hive逐渐发展到支持许多传统的数据仓库工作负载，对完整DML支持和ACID事务的要求也越来越高。因此，Hive现在支持执行INSERT、UPDATE、DELETE，MERGE语句。**它通过读时快照隔离和良好定义的语义（以防在HMS上构建的atransaction manager出现故障）提供ACID保证**。目前事务只能跨一条语句；我们计划在不久的将来支持多语句事务，使用Hive multi-insert语句在一个事务中写入多个表今后是可能的。

在支持Hive中的行级别操作方面需要克服的主要挑战是（i）系统中缺少事务管理器；（ii）底层文件系统中缺少文件更新支持。在下面，我们提供了更多关于Hive中ACID实施的详细信息，以及如何解决这些问题。

3.3.3 事务和锁管理

Hive将事务和锁定信息状态存储在HMS中，它使用全局Transaction标识符或XNID，即，元存储生成的单调递增值，用于系统中运行的每个事务，与之对应，每个XNID映射到一个或多WriteId， WriteId也是元数据生成且单增，但是是在表范围内，WriteId存储在事务写入的每个记录中；由AME事务写入同一表的所有记录共享SAMEWITREID。相反，共享SamewWriteId的文件使用FileId进行一次性标识，而文件中的每个记录都由rowdFieId进行标识。注意，WriteId、FileIDAndRowIDIdentifications的组合唯一地标识表中的每个记录。在单元中删除操作被建模为INSERTO，该操作将标记记录指向要删除的记录的唯一标识符

3.4 查询优化器

虽然在hive的初始版本中对优化的支持是有限的，但是很明显，它的执行内部的开发不足以保证高效的性能。因此，目前这个项目包含了关系数据库系统中通常使用的许多复杂技术。本节描述了帮助系统生成更好的计划并改进查询执行的最重要的优化特性，包括它与Apache Calcite的集成

3.4.1 基于规则和成本的优化器

最初，apache hive在解析输入SQL语句时执行多次重写以提高性能。此外，它还包含一个基于规则的优化器，可以对查询生成的物理计划应用简单的转换。例如，许多优化的目标都是尽量降低数据洗牌的成本，这是MapReduce引擎中的一个关键操作。还有其他操作优化来下推筛选器谓词、投影未使用的列和修剪分区。虽然这对于某些查询是有效的，但使用物理计划表示使得实现复杂的重写（如连接重排序、谓词简化和传播，或基于物化视图的重写）过于复杂

因此，引入了一种新的计划表示和优化方法，由ApacheCalcite[3,19]提供支持。calcitei是一个模块化的、可扩展的查询优化器，具有内置的元素，这些元素可以以不同的方式组合起来构建自己的优化逻辑。这些包括不同的重写规则、计划和成本模型

calcite提供了两种不同的计划引擎：（i）基于成本的计划器，它触发重写规则，以降低总体表达式成本；（ii）exhaustiveplanner，它彻底触发规则，直到生成不再被任何规则修改的表达式。转换规则对两个计划者都起不到明显的作用

Hive实现了与其他查询优化器类似的多阶段优化[52]，其中每个优化阶段使用一个规划器和一组重写规则。这允许Hive通过指导不同查询计划的搜索来减少总体优化时间。一些calcite规定hive连接重排序、多个操作符重排序和消除、常量折叠和传播以及基于约束的转换。

3.4.2查询重新优化

hive支持在执行过程中抛出某些错误时进行查询重新优化。特别地，它实现了两个独立的再优化战略第一种策略，overlay，更改所有查询重新执行的某些配置参数。例如，用户可以选择强制查询重新执

行中的所有连接使用特定的算法，例如，使用排序的哈希分区-合并。这个当已知某些配置值以使查询执行更有效时，可能会很有用健壮的第二种策略是重新优化，它依赖于运行时的统计数据。在计划查询时，Optimizer根据从HMS检索到的统计信息估计计划中中间结果的大小。如果这些估计不准确，优化器可能会犯规划错误，例如，错误的连接算法选择或内存位置。这反过来可能导致性能差和执行错误。配置单元捕获计划中每个操作员的运行时统计信息。如果在查询执行期间检测到上述任何问题，则使用运行时统计信息重新优化查询并再次执行。

(a)

```
CREATE MATERIALIZED VIEW mat_view AS
SELECT d_year, d_moy, d_dom, SUM(ss_sales_price) AS sum_sales
FROM store_sales, date_dim
WHERE ss_sold_date_sk = d_date_sk AND d_year > 2017
GROUP BY d_year, d_moy, d_dom;
```

(b)

q_1:

```
SELECT SUM(ss_sales_price) AS sum_sales
FROM store_sales, date_dim
WHERE ss_sold_date_sk = d_date_sk AND
  d_year = 2018 AND d_moy IN (1,2,3);
```

q_1':

```
SELECT SUM(sum_sales)
FROM mat_view
WHERE d_year = 2018 AND d_moy IN (1,2,3);
```

(c)

q_2:

```
SELECT d_year, d_moy, SUM(ss_sales_price) AS sum_sales
FROM store_sales, date_dim
WHERE ss_sold_date_sk = d_date_sk AND d_year > 2016
GROUP BY d_year, d_moy;
```

q_2':

```
SELECT d_year, d_moy, SUM(sum_sales)
FROM (
  SELECT d_year, d_moy, SUM(sum_sales) AS sum_sales
  FROM mat_view
  GROUP BY d_year, d_moy
  UNION ALL
  SELECT d_year, d_moy, SUM(ss_sales_price) AS sum_sales
  FROM store_sales, date_dim
  WHERE ss_sold_date_sk = d_date_sk AND
    d_year > 2016 AND d_year <= 2017
  GROUP BY d_year, d_moy
) subq
GROUP BY d_year, d_moy;
```

3.5 查询执行器

查询执行内部的改进，例如从MapReduce到ApacheTez的转换，以及基于列的存储格式和矢量操作符的实现，将Hive中的查询延迟降低了几个数量级。然而，为了进一步改进执行运行时，Hive需要额外的增强来克服其初始体系结构固有的限制，该体系结构是为长时间运行的查询定制的。除此之外，（i）执行需要在启动时分配容器，这很快成为低延迟查询的一个关键瓶颈，（ii）即时（JIT）编译器优化没有尽可能有效，因为在查询执行后容器被简单地杀死，以及（iii）Hive无法利用查询内部和查询之间的数据共享和缓存可能性，导致不必要的IO开销

3.5.1 LLAP：live long and process

Live Long and Process（也称为LLAP）是一个可选层，它提供持久的多线程查询执行器和内存缓存中的多租户，以在配置单元中大规模提供更快的SQLprocessing。LLAP并没有取代Hive使用的现有执行运行时，比如Tez，而是增强了它。特别是，执行是由配置单元查询协调器在LLAP节点和常规容器上透明地调度和监视

LLAP的数据I/O、缓存和查询片段执行能力都封装在数据库中。守护程序被设置为在集群中的工作节点中连续运行，这有助于JIT优化，同时避免任何启动开销。纱线用于粗粒度的资源分配和调度。它为数据保留内存和CPU，并处理重启和重新定位。

I/O elevator。elevator daemon使用单独的线程卸载数据I/O和解压缩，我们称之为I/O elevator。数据分批读取并转换为内部游程编码（RLE）列格式，以便进行矢量化处理。一个列批在被读取后立即进入执行阶段，这允许在处理下一批时处理前一批2.改造从底层文件格式到内部数据格式的转换是使用特定于每种格式的插件完成的。目前，LLAP支持ORC、Parquet和文本文件的翻译格式I/O elevator可以将投影、sargable pred-icates2和Bloom过滤器向下推送到文件读取器（如果提供的话）。例如，ORC文件可以利用这些结构跳过读取整个列和行组。

3.5.2 关于workload管理的优化

workload manager控制对配置单元执行的每个查询的LLAP资源的访问。管理员可以创建资源计划，即自包含的资源共享配置，通过在LLAP上运行的并发查询来提高执行的可预测性和集群共享。这些因素在多租户环境中至关重要。虽然可以在系统中定义多个资源重组计划，但在给定的时间内，对于给定的部署，只有其中一个计划可以处于活动状态。资源计划由HMS中的配置单元持久化

为了提供一个示例，请考虑以下生产集群的resource plan定义

```
CREATE RESOURCE PLAN daytime;
CREATE POOL daytime.bi
  WITH alloc_fraction=0.8, query_parallelism=5;
CREATE POOL daytime.etl
  WITH alloc_fraction=0.2, query_parallelism=20;
CREATE RULE downgrade IN daytime
  WHEN total_runtime > 3000 THEN MOVE etl;
ADD RULE downgrade TO bi;
CREATE APPLICATION MAPPING visualization_app IN daytime TO bi;
ALTER PLAN daytime SET DEFAULT POOL = etl;
ALTER RESOURCE PLAN daytime ENABLE ACTIVATE;
```

Line1创建资源计划。第2-3行创建一个池，池中有80%的LLAP资源集群。那些资源可用于同时执行多达5个查询。类似地，第4-5行创建一个poolet，其中包含可用于并发执行多达20个查询的剩余资源。第6-8行创建一个规则，当查询运行超过3秒时，将查询从位移到位资源池。请注意，可以执行前面的操作，因为与容器相比，查询片段更容易抢占。Line9为一个名为Interactive\u bi的应用程序创建一个映射，即Interactive\u bi启动的所有查询最初都将从中获取资源他们在里面第10行为系统中其余的查询设置默认池toetl。最后，第11行启用并激活群集中的资源计划

3.6 联邦数仓系统

在过去的十年中，专业数据管理系统的数量不断增加，这些系统之所以流行，是因为它们在特定的用例中比传统的RDBMS具有更好的性价比

除了本机处理功能外，Hive还可以作为中介，因为它的设计目的是支持多个独立数据管理上的查询系统通过Hive统一访问这些系统的好处是多方面的。应用程序开发人员可以选择多种系统的混合，以实现所需的性能和功能，但他们只需要针对单个接口编写代码。因此，应用程序独立于不成熟的数据系统，这使得以后更改系统更加灵活

hive自身可以用来实施不同系统间的数据迁移和转化，减少对第三方工具的需求。此外，Hive作为一个中介，可以通过Ranger或Sentry在全球范围内实施访问控制和捕获审计跟踪，还可以通过Atlas帮助满足法规遵从性要求

为了以模块化和可扩展的方式与其他引擎交互，Hive包含了一个需要为每个引擎实现的Rage handlerinterface。存储处理程序包括：（i）输入格式，描述如何从外部引擎读取数据，包括如何分割功以提高并行性；（ii）输出格式，描述如何将数据写入外部引擎，（iii）aSerDe（seri alizer and deserializer），描述如何将数据从配置单元内部表示转换为外部引擎表示，反之亦然；（iv）asetore hook，定义作为针对HMS的事务的一部分调用的通知方法，例如，当创建外部系统备份的新表时，或当新行插入到这样的表中。ausable存储处理程序从外部系统读取数据的最小实现至少包含一个输入格式和反序列化程序。一次storagehandler接口已经实现，从Hive查询外部系统是直截了当的，storagehandler实现背后的所有复杂性对用户都是隐藏的。例如，ApacheDruid[4,58]是一个开放源代码数据存储库，设计用于对事件数据进行业务智能（OLAP）查询，广泛用于面向高级用户的分析应用程序；Hive提供了一个Druid存储处理程序，因此它可以利用其高效性执行交互式查询。要从配置单元开始查询德鲁伊，唯一需要做的就是从配置单元注册或创建德鲁伊数据源。首先，如果一个数据源已经存在于Druid中，我们可以用一个简单的语句将一个Hive externaltable映射到它：

```
1 CREATE EXTERNAL TABLE druid_table_1
2 STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
3 TBLPROPERTIES ('druid.datasource' = 'my_druid_source');
```

注意，我们不需要为数据源指定列名称或类型，因为它们是从Druid元数据自动调用的。反过来，我们可以使用以下简单语句在Druid中从Hive创建数据源：

```
CREATE EXTERNAL TABLE druid_table_2 (
  __time TIMESTAMP, dim1 VARCHAR(20), m1 FLOAT)
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler';
```

一旦Druid源作为externaltables在Hive中可用，我们就可以对它们执行任何允许的表操作

3.7 讨论与结论

ApacheHive的早期成功源于使用知名接口开发批处理操作并行性的能力。它使数据加载和管理变得简单，可以优雅地处理节点、软件和硬件故障，而无需昂贵的修复或恢复时间。在本文展示了社区如何将系统的实用性从ETL工具扩展到成熟的企业级数据仓库。我们描述了一个事务系统的添加，该系统非常适合星型模式数据库中所需的数据修改。我们展示了将查询延迟和并发性引入交互式操作领域所需的主要运行时改进。我们还描述了处理当前视图层次结构和大数据操作所必需的基于成本的优化技术。最后，我们展示了如何将Hive用作多个存储和数据系统的关系前端。所有这一切都发生在没有任何妥协的原始特征的系统，使它流行的。阿帕奇蜂巢架构和设计原则必须在今天的分析景观强大。我们相信，随着新的部署和存储环境的出现，它将继续蓬勃发展，正如今天集装箱化和云技术所显示的那样。

---

4. 消息队列 Kafka vs RabbitMQ

4.1 kafka基本概念

kafka是一个分布式的发布/订阅的消息队列，主要应用于大数据实时领域。

消息队列两大作用：

①**缓冲和削峰**：缓冲即 有助于解决生产者和消费者数据处理速度不一致的情况，削峰即能够使服务器定住突发的访问压力，不会因为突发的超负荷请求而崩溃

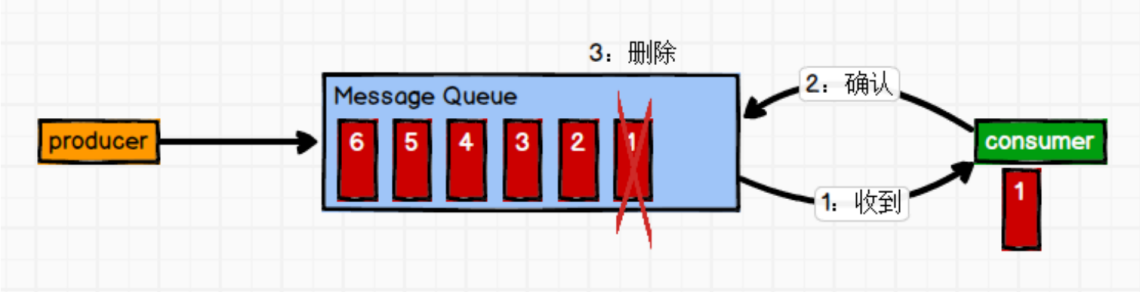
②**解耦**：允许你独立的拓展或者修改两边的处理过程，只要确保它们遵守相同的接口约束

消息队列的两种模式

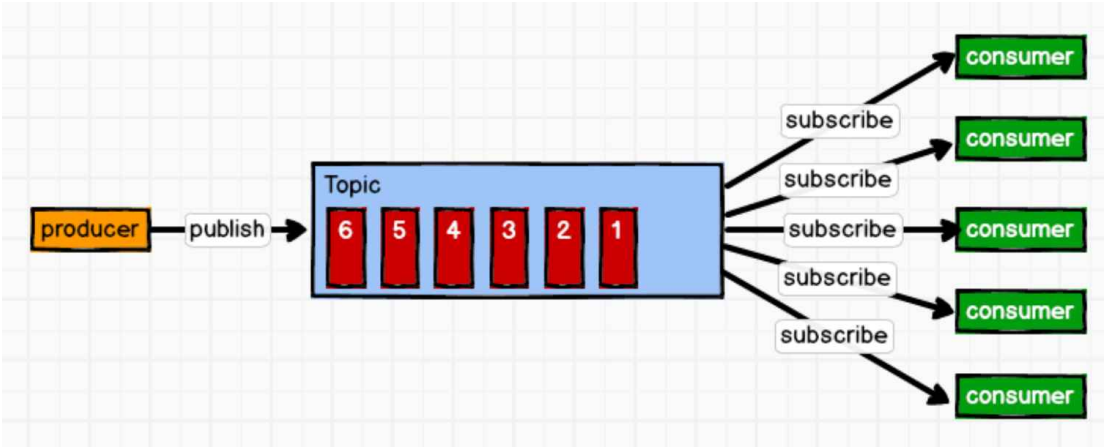
- **点对点模式**：

消息生产者生产消息发送到 Queue 中，然后消息消费者从 Queue 中取出并且消费消息。

消息被消费以后，queue 中不再有存储，所以消息消费者不可能消费到已经被消费的消息。 Queue 支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费

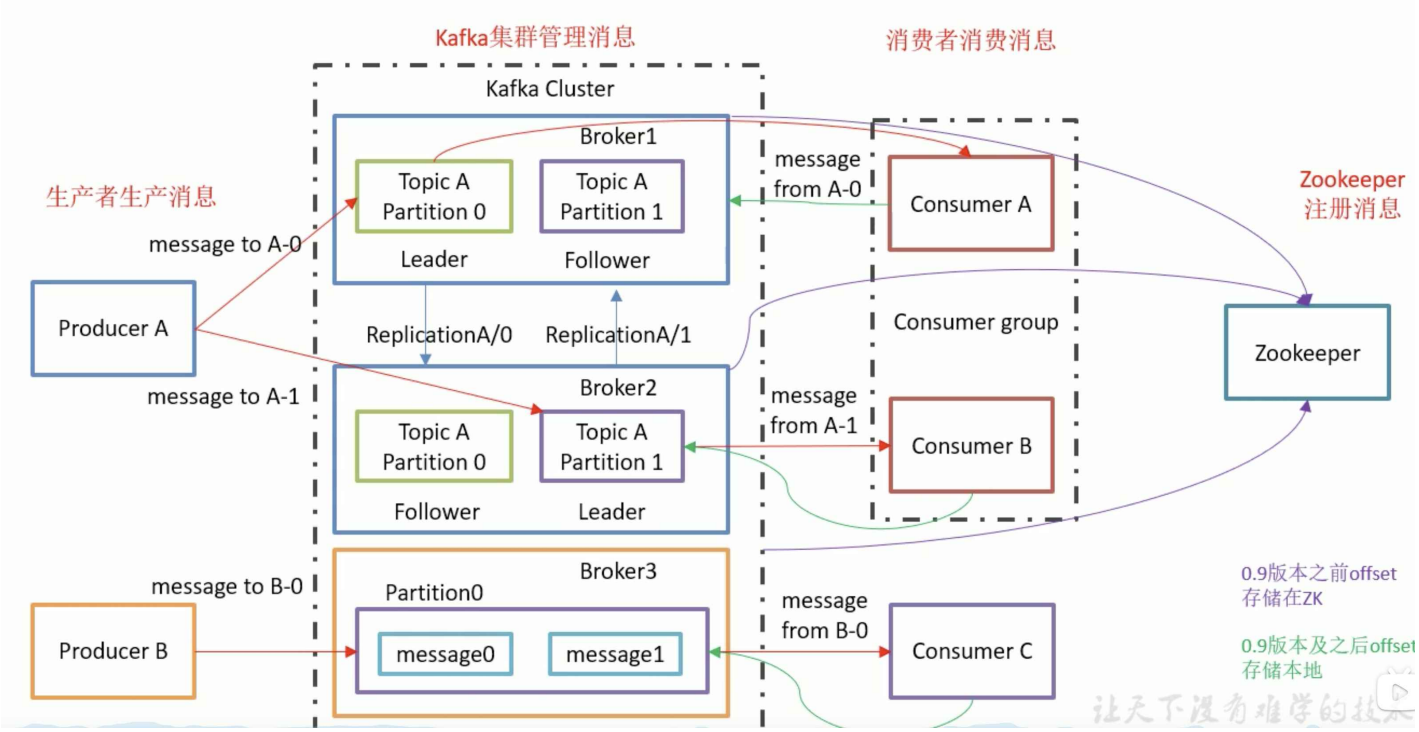


- **发布/订阅模式**：

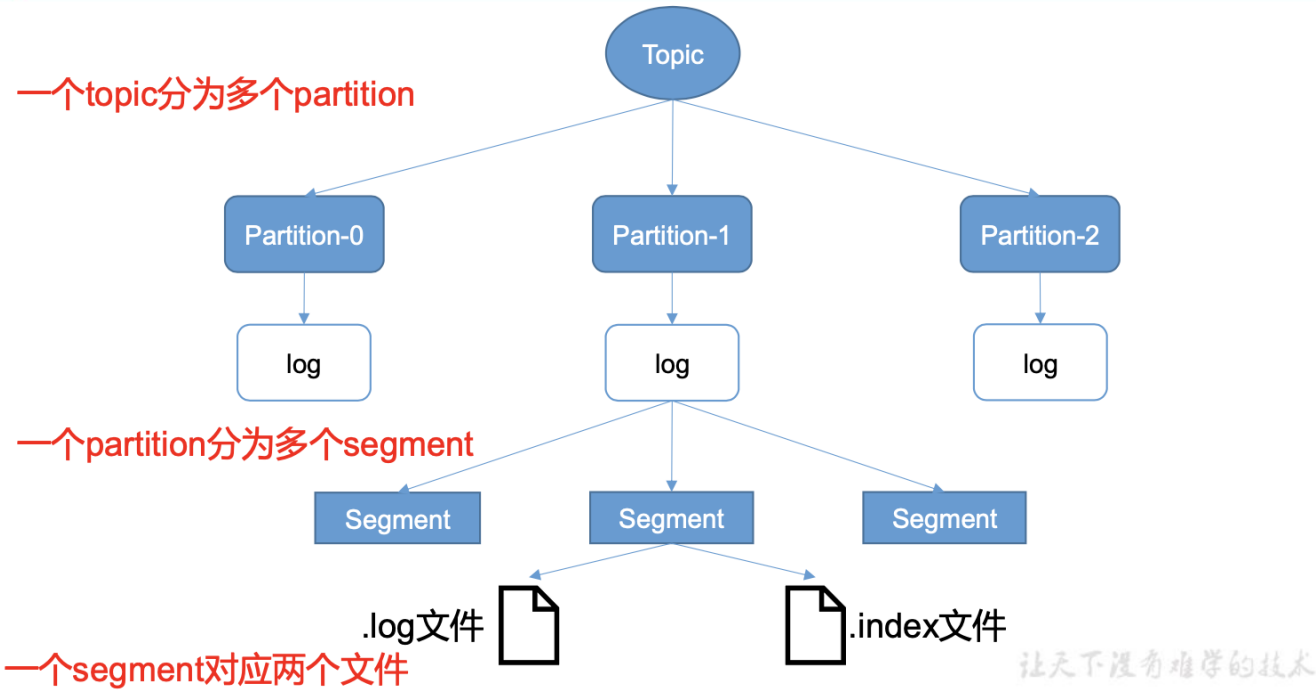


在发布订阅模式有两种消费消息的方式，1：消息队列“**推送**”消息，这样所有的消费者会在同一时间接收到消息，并且可能上一次来的消息未消费完，下一批消息就推送过来了；2：消费者“**拉取**”数据，**消费者自己维护一个长轮询**，不断主动询问消息队列中是否有数据，**消费者可以根据自己消费的速度拉取数据**，其中kafka采取的是第二种消费模式。

4.2 kafka架构

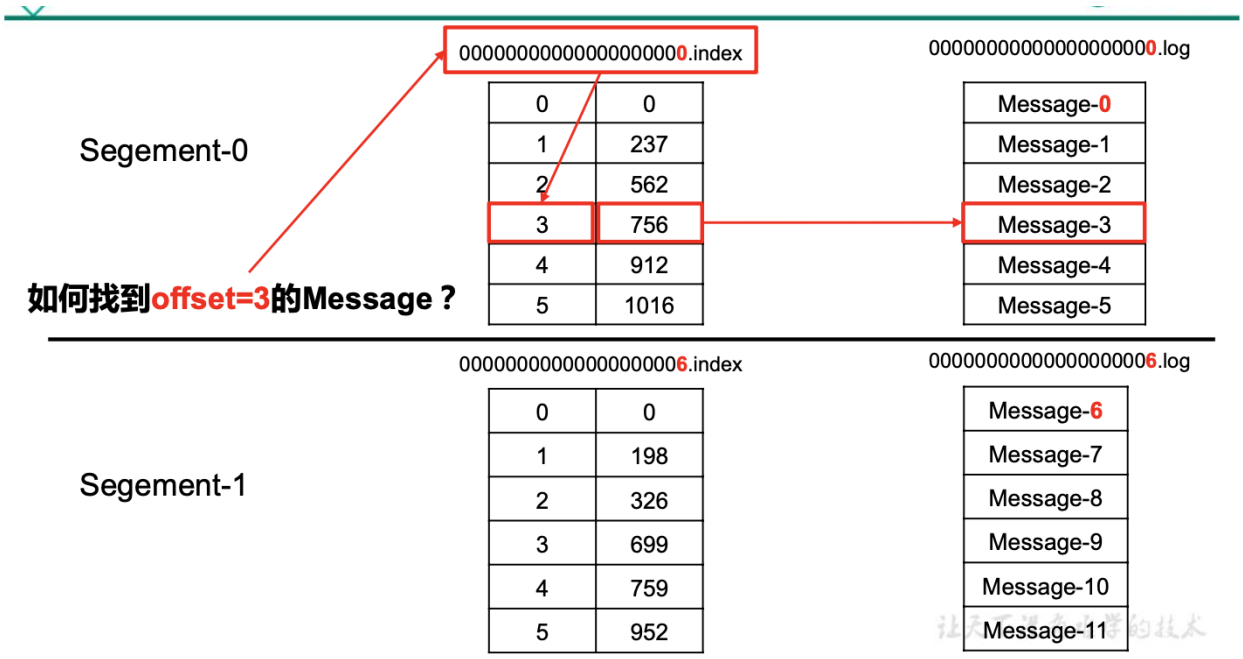


- producer：消息生产者，向kafka broker发消息的客户端
- consumer：消息消费者，向kafka broker拉取消息的客户端
- Consumer group（CG）：消息消费者组，订阅了某一主题的CG，对于某个分区，消息只能被消费者组中的一个消费者消费
- broker：一个kafka服务器就是一个broker，一个集群由多个broker组成，
- topic：生产者和消费者面向的都是主题topic
- partition：同一topic会被划分为多个partition，分摊到不同的broker上面，每个partition是一个队列，目的①可以并发写、读 ，目的②将写、读压力分摊，负载均衡
- replica：副本，为保证集群中某个节点发送故障时，该节点上的partition不丢失数据
- Leader：每个分区的“主”，生产者生产数据的对象，消费者消费数据的对象，都是leader
- follower：每个分区的“从”，实时从leader中同步数据，保持和leader的同步，leader发送故障时，某个follower会成为leader



每个partition对应一个log文件，每个log（partition）会分为多个segment，每个segment对应两个文件，“.log文件”和“.index文件”，分别存储数据内容和索引

查找时，首先二分查找O(logn)，找到对应的segment文件，再通过O(1)查找.index文件，找到对应的位置，最后一次通过O(1)读取到.log文件里面的数据（O(logn) —> O(1) —> O(1)）



4.3 kafka生产者

4.3.1 分区策略

- 指明 partition 的情况下，直接将指明的值直接作为 partiton 值;

- 没有指明 partition 值但有 key 的情况下，将 key 的 hash 值与 topic 的 partition 数进行取余得到 partition 值
- 既没有 partition 值又没有 key 值的情况下，第一次调用时随机生成一个整数(后 面每次调用在这个整数上自增)，将这个值与 topic 可用的 partition 总数取余得到 partition 值，也就是常说的 round-robin 算法

4.3.2 数据可靠性

为保证 producer 发送的数据，能可靠的发送到指定的 topic，topic 的每个 partition 收到 producer 发送的数据后，都需要向 producer 发送 ack(acknowledgement 确认收到)，如果 producer 收到 ack，就会进行下一轮的发送，否则重新发送数据

副本数据同步策略

当多少副本同步完成后，发送ack?

方案	优点	缺点
半数以上完成同步，就发送 ack	延迟低	选举新的 leader 时，容忍 n 台节点的故障，需要 2n+1 个副本
全部完成同步，才发送 ack	选举新的 leader 时，容忍 n 台节点的故障，需要 n+1 个副本	延迟高

Kafka 选择了第二种方案，原因如下:

1.同样为了容忍 n 台节点的故障，第一种方案需要 2n+1 个副本，而第二种方案只需要 n+1 个副本，而 Kafka 的每个分区都有大量的数据，**第一种方案会造成大量数据的冗余**。 2.虽然第二种方案的网络延迟会比较高，但网络延迟对 Kafka 的影响较小

ISR

采用第二种方案之后，设想以下情景:leader 收到数据，所有 follower 都开始同步数据， 但有一个 follower，因为某种故障，迟迟不能与 leader 进行同步，那 leader 就要一直等下去， 直到它完成同步，才能发送 ack。这个问题怎么解决呢?

leader维护了一个动态的（in-sync replica set）ISR，意为和leader保持同步的follower集合，当ISR中follower数据完成同步后，leader就会给follower发送ack，如果follower长期没和leader同步数

据，该follower就会被踢出ISR

ack参数配置：

①ack=0，不等待broker的ack，该操作提供了最低的延迟，当broker故障时会丢失数据

②ack=1，partition的leader落盘成功后返回ack，如果在follower同步成功之前leader故障，那么将会丢失数据

③ack=-1，partition的leader和follower全部落盘成功之后才返回ack，如果在follower同步完成之后，leader发送ack之前leader挂掉了，那么会造成数据重复

故障处理细节

HW：（high water）高水位，指的是ISR中最小的LEO，也是消费者可见的最大offset

LEO：（log end offse）每个副本的offset

4.4 kafka消费者

4.4.1 消费方式

consumer使用一个长轮询，一直“拉取”broker数据，pull模式优化：传入一个参数timeout，如果当前没有数据可以消费，consumer会等待timeout再去拉取数据

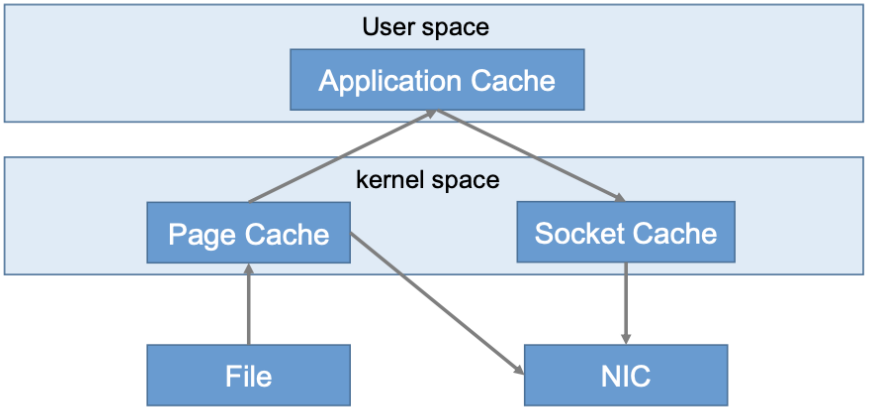
4.4.2 分区分配策略

有两种分区分配策略，一种是range，一种是roundrobin

4.4.3 kafka高效读写

- 零拷贝

拷贝不经过用户程序空间，在操作系统级别完成



- 顺序写磁盘

磁盘顺序写，大大减少磁头寻址时间

4.5 最适合kafka的场景

4.5.1 Pub/Sub

Kafka可以很好地匹配具有以下属性的pub/sub用例：（i）如果路由逻辑很简单，那么kafka“topic”概念可以处理需求，（ii）如果每个topic的吞吐量超出RabbitMQ可以处理的范围（例如event firehose）

4.5.2 scalable ingestion system

很多许多领先的大型数据处理平台都支持将数据加载到系统中后进行高吞吐量处理。然而，在许多情况下，**将数据加载到这些平台是主要的瓶颈**。kafka为此类场景提供了一个可扩展的解决方案，它已经集成到许多此类平台中，包括Apache Spark和Apache Flink等

4.5.3 data-layer infrastructure

由于其持久性和高效的多播性，kafka可以作为连接企业内各种批处理和流式服务及应用程序的底层数据基础设施

4.6 RabbitMQ简介

4.6.1 特点

RabbitMQ 是一个由 Erlang 语言开发的 AMQP 的开源实现。

AMQP： Advanced Message Queue，高级消息队列协议。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。

RabbitMQ 最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。具体特点包括：

1. 可靠性（Reliability）

RabbitMQ 使用一些机制来保证可靠性，如持久化、传输确认、发布确认。

2. 灵活的路由（Flexible Routing）

在消息进入队列之前，通过 Exchange 来路由消息的。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功能，可以将多个 Exchange 绑定在一起，也通过插件机制实现自己的 Exchange 。

3. 消息集群（Clustering）

多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。

4. 高可用（Highly Available Queues）

队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。

5. 多种协议（Multi-protocol）

RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。

6. 多语言客户端（Many Clients）

RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。

7. 管理界面（Management UI）

RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。

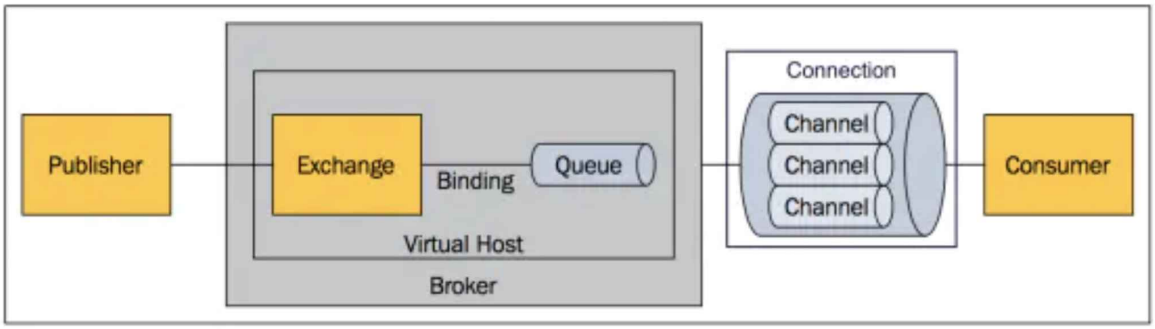
8. 跟踪机制（Tracing）

如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。

9. 插件机制（Plugin System）

RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

4.6.2基本概念



RabbitMQ 内部结构

- Message

  消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。

- Publisher

  消息的生产者，也是一个向交换器发布消息的客户端应用程序。

- Exchange

  交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。

- Binding

  绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

- Queue

  消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

- Connection

  网络连接，比如一个TCP连接。

- Channel

  信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内地虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

- Consumer

  消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

- Virtual Host

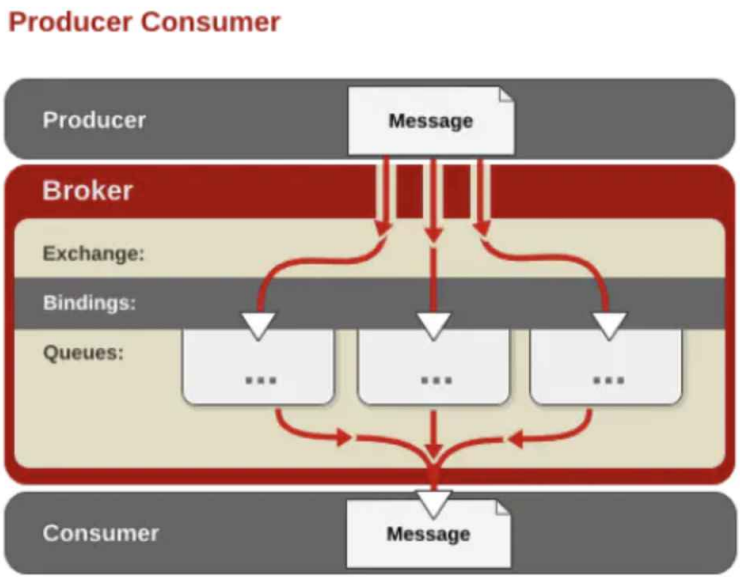
虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定。

- Broker

  表示消息队列服务器实体

4.6.3 AMQP 中的消息路由

AMQP 中消息的路由过程和 Java 开发者熟悉的 JMS 存在一些差别，AMQP 中增加了 Exchange 和 Binding 的角色。生产者把消息发布到 Exchange 上，消息最终到达队列并被消费者接收，而 Binding 决定交换器的消息应该发送到那个队列

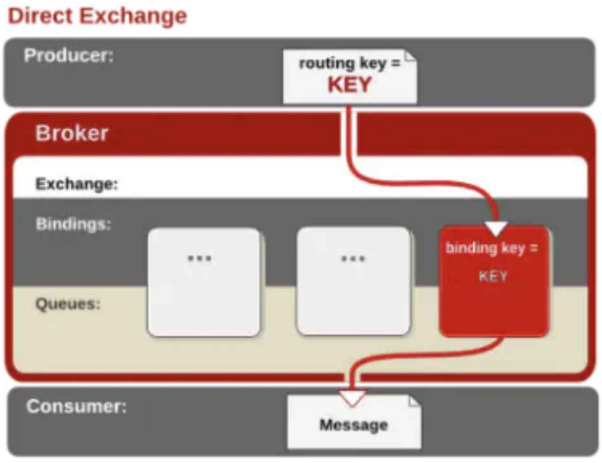


AMQP 的消息路由过程

4.6.4 Exchange 类型

Exchange分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers 。headers 匹配 AMQP 消息的 header 而不是路由键，此外 headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以直接看另外三种类型：

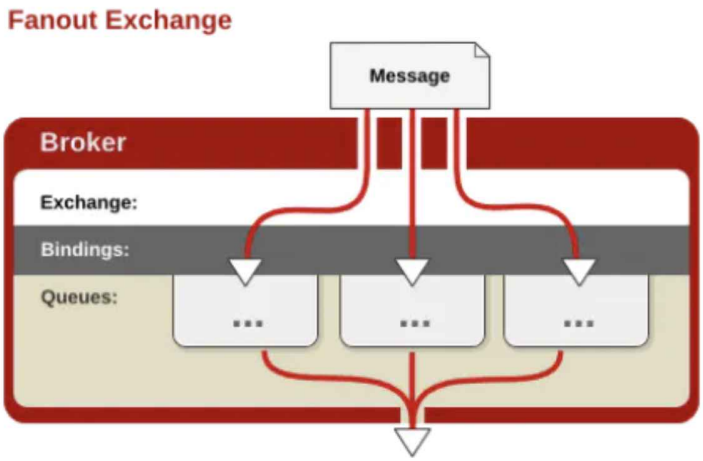
- direct



direct 交换器

消息中的路由键（routing key）如果和 Binding 中的 binding key 一致， 交换器就将消息发到对应的队列中。路由键与队列名完全匹配，如果一个队列绑定到交换机要求路由键为“dog”，则只转发 routing key 标记为“dog”的消息，不会转发“dog.puppy”，也不会转发“dog.guard”等等。它是完全匹配、单播的模式。

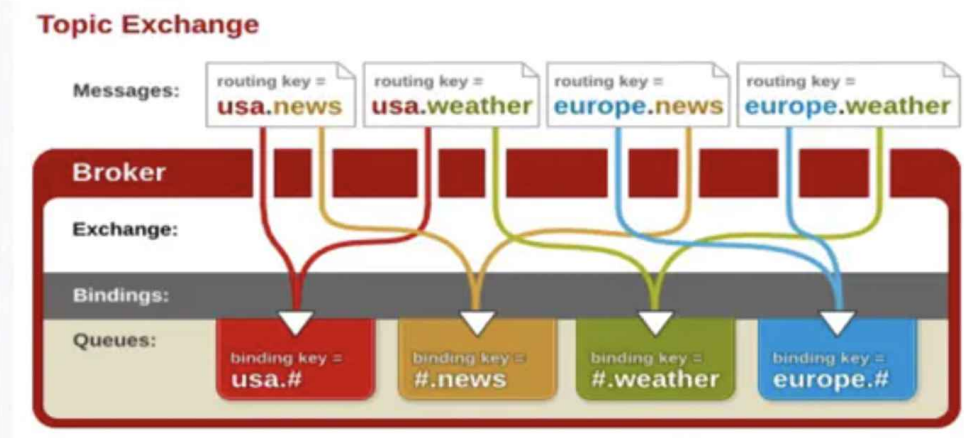
- fanout



fanout 交换器

每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。fanout 交换器不处理路由键，只是简单的将队列绑定到交换器上，每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 类型转发消息是最快的。

- Topic



topic 交换器

topic 交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符：符号“#”和符号“”。*#匹配0个或多个单词，*匹配不多不少一个单词。

5.云原生弹性数仓Snowflake

5.1 摘要

☐ 新型数据仓库、DaaS、multi-cluster、shared-data architecture

传统的数据仓库被设计为用于固定资源，数据的结构、容量、传入速度是可预测的，不能使用云的弹性，随着数据种类的越来越多、数据量越来越大，需要新型数据仓库

那么，什么是Snowflake呢?

下面用一句话概括什么是Snowflake：

Snowflake是整套系统部署在云上的（云原生）、DaaS（dataware house as a service）、serverless、面向OLAP的（基于列大量数据分析）、分布式数据仓库

snowflake有以下特点：

- 多租户 multi-tenant，pay-as-you-go，pay for use
- pure service、SaaS（集群配置、调优、运维无需关心，**Snowflake has only one tuning parameter: how much performance the user wants (and is willing to pay for).**

- 事务型，支持SQL,ACID，**支持ACID级别的事务一致性**
- 内置半结构化和无结构化数据（semi-structure data、no schema data）
- 高度弹性（compute、store）和持续可用（resilient to node failure，no downtime upgrade）
- 高可拓展（算力VWs（ET2）可拓展、存储S3可拓展）
- **多集群-共享数据架构**（multi-cluster，sharing-data architecture）
- **计算存储分离架构**
- 计算上云（计算节点使用虚拟机，EC2）
- 存储上云（run on AWS cloud，AWS S3 对象存储）
- 不基于Hadoop，处理引擎自主研发
- 端对端安全 end-to-end encrypted

5.2 存储和计算

☐ 计算和存储，应该采用什么架构，以及背后选择的原因

那个年代（2012~2014） Shared-nothing architecture成为数据仓库主流（如Teradata、Vertica），原因1：可拓展，原因2：廉价硬件

（虽然是大流，可我们snowflake偏偏不随大流

大流的share-nothing架构有什么特点呢?

每个query processor node有本地的磁盘，表是水平分区的、跨节点的（数据库分库分表），每个节点只负责本地磁盘上的行，适合星型图查询，每个节点在相同硬件上跑

计算存储耦合的缺点：

（补充，首先设计上不满足：高内聚，低耦合）

1. workload heterogeneous，不能弹性分配计算存储资源，需求变化不一致
2. 集群节点数改变：有的节点既要负责计算，又要负责数据转组（data shuffling），会大大影响弹性和可用性
3. 在线升级：如果需要频繁升级，频繁节点宕机，系统resizing，会对升级带来很大麻烦

考虑到大集群下的节点宕机频繁、集群节点数变更、满足在线升级这些需求

因此Snowflake采取**计算存储分离**的架构，**资源解耦，计算存储松耦合，两者是独立可拓展的服务**

- 计算：VW，Snowflake sharing-nothing engine
- 存储：AWS S3 对象存储（key-value），使用HTTP(s)即可，无限、可拓展，（采用友商亚马逊的存储服务，格局真大！）

多集群-共享数据架构

为了减少计算层和存储层的网络I/O通信，（虽然现在是万兆网卡，当然有本地存储空间还是用上一些），每个计算节点在本地磁盘缓存一些 table data，本地磁盘专门用来放置临时数据和缓存（建议用SSD），缓存变热后，性能将达到甚至超过share-nothing system，我们将这种架构称为**multi-cluster，shared-data architecture**（理解：**①同一个虚拟仓库下，工作节点共享‘一块’cache，②由于采用S3，每个VM可以访问所有数据**，这叫做共享数据sharing data）

5.3.架构

☐ 介绍了 “多集群-共享数据架构”

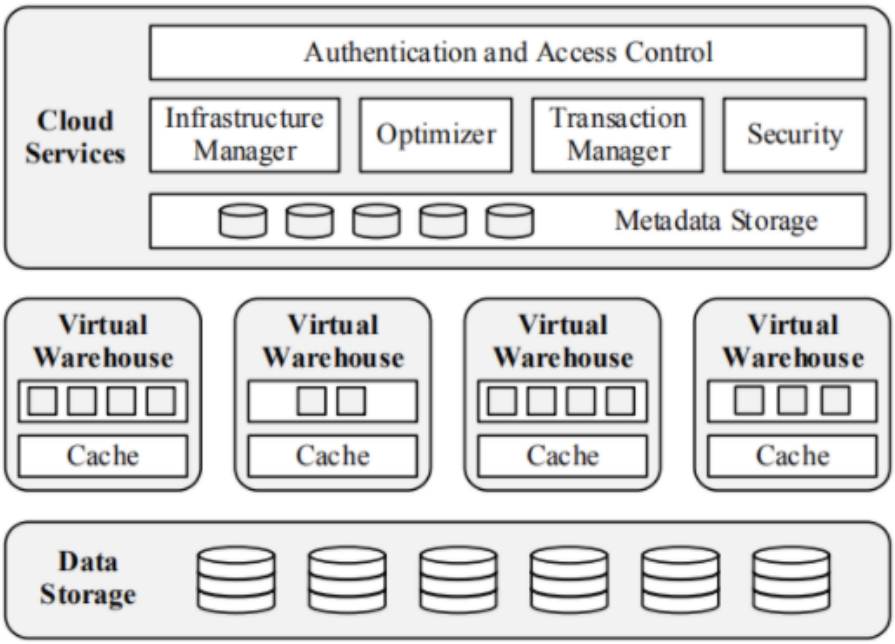


Figure 1: Multi-Cluster, Shared Data Architecture

3.0

Snowflake提供高可用，高容错，独立可拓展的企业级服务，服务通过RESTful结构进行通信，并分为三层，如上图所示。

3.1 Data Storage层

☐ 采用AWS S3存储**表数据**和**查询结果**，也存放**temp data & query results**，！注意元数据不是存这的！

采用S3对象存储的原因：发现尽管性能可能不同，但S3的高可用、耐用保障是无敌的，

选择S3“write once”、“append only”作为底层存储，而不采用Hadoop

和本地存储相比，S3有高出许多的访问延迟和更高的CPU开销，尤其是使用https时，但是更重要的是，**S3是具有相对简单的基于HTTP的PUT\GET\DELETE的blob存储区**

tables被水平划分到大块、等于块或者页的不可变的文件中，

S3支持GET请求得到部分数据，

查询query只需下载文件头和需要的列数据即可

Snowflake不仅将S3用于表数据，也用于查询生成的临时数据，一旦本地磁盘用尽，将数据spill（刷盘）到S3中

存储结果简单可见，可以用新形式的客户端交互并简化查询

表对象组成的元数据存储在第三层云服务层

3.2 Virtual Warehouses层

- [ ] **ET2实例集群，pure compute resource**，在由虚拟机组成的弹性集群上执行查询操作（query execution），共享cache，VMs弹性拓展，面向列的引擎

一个虚拟仓库VM，是一个用户独享的cluster

virtual warehouse 由EC2实例集群组成 （1个VW = n个EC2虚拟机 ，一个EC2虚拟机称为一个工作节点）

3.2.1 弹性和隔离

弹性（VW elasticity）：

VWs是计算资源，可以创建并按需调整大小，当用户没有query时，甚至鼓励关闭VWs集群，弹性使用计算资源

隔离性：

每一个查询只允许在一个VW上，工作节点在VMs（集群）中不共享，属于哪个VM的工作节点就归哪个VW，这就是查询的隔离性

（共享节点是未来的研究方向，适用于不太关注隔离性的场景，简单理解为VMs组成一件T-shirt（different VMs for different uses），VM里的某一个节点是T-shirt上的一块布，布有隔离性，用户对衣服大小需求是弹性的，当用户不需要这么大的衣服，衣服会变小，其中多余的布块会组成别的T-shirt）

当有新查询，在对应的VM中各个工作节点产生工作进程，工作进程仅存活query任务中，工作进程不会造成外部可见性影响

（理解：**VW是为了分布式计算，一个query交给一个VW处理，其中的多个工作节点MR处理任务，VWs集群的各个VW又可以并行运行，每个VW可以访问相同的数据，这就是存储共享，而这种存储共享是可无限拓展的**）

3.2.2 本地缓存和文件窃取

每个工作节点在本地磁盘维持一个表数据的cache，记录工作节点过去访问过的S3对象数据，cache保存**文件头和查询所用到的单独的列**，cache在工作节点的生命周期内生存，在并发和后续工作进程中共享，采用**LRU**缓存置换算法，为了提高缓存命中率、避免cache冗余，对连续、或并发查询相同文件，**一致性哈希**映射到同一个工作节点上

- 懒惰的一致性哈希：

当节点宕机或者VMs节点数改变（resize），不会马上data shuffle，通过多个查询替换cache内容

- 偏斜处理 skew handing：

简单来说就是节点的工作量负载均衡处理，**file stealing**，当一个peer发现他的peer节点还有文件未读取，改变ownership到当前query下面，帮助其他节点分摊workload

3.2.3 执行引擎

- 柱状的（columnar）

对于OLAP需要大量的数据分析的需求，**面向列存储和执行**能更好利用cpu缓存和SIMD指令

- vectorized向量化

避免了中间结果，数据以流水线的形式，一批批以列的格式处理，节省I/O，提高cache效率

- 基于推送

关系运算符将结果‘推’给下流运算符，而不是等待这些运算符拉取数据，从紧密的循环中删除了控制流，提高了cache efficiency

此外，还有以下特性：

- 查询过程中无需管理事务
- 查询在一组不可变文件上执行，也没有缓存池
- 当内存耗尽时，让大多操作（join、group by、sort）刷盘spill到磁盘

3.3 Cloud Service层

- [ ] 权限控制，查询优化，并发控制
- [ ] always online
- [ ] 微服务，K/V store（hard state）

相比于VWs层是**临时、用户专用**的资源

权限控制、查询优化器、事务manager等是**长期存在**，并**对多租户共享的**

服务高可用（单个节点故障不会导致数据丢失）、高拓展

3.3.1 查询管理和优化

用户所有的查询都要经过cloud service层，

查询处理过程：语法分析、对象解析、访问控制、优化

- 采用自上而下基于成本的优化，优化的信息会自动保存到数据加载和更新中
- **没有使用索引**
- 推迟执行，减少optimizor的错误
- 持续收集state of query、performance counter、detect node failure
- 用户通过图形界面监视、分析过去和正在进行的查询

3.3.2 并发控制

通过快照隔离（snapshot isolation），实现ACID事务

事务读取会看到数据库的一致性快照

快照隔离是实施在MVCC上的，每个更改的数据库对象会保存一段时间，

对表的（插入、更新、删除、合并）会产生一个新的表对象，（**理解：复制表对象t得到t‘，在t‘上进行操作，删除t，然后呢，t对象也不是真正删除，会最多保留90天，这样可以通过SQL读取早期的版本**）

文件的增加和删除记录在元数据中（一个全局的K/V strore）

time travel功能示例

```
SELECT * FROM my_table AT(TIMESTAMP =>
  'Mon, 01 May 2015 16:20:00 -0700'::timestamp);
SELECT * FROM my_table AT(OFFSET => -60*5); -- 5 min ago
SELECT * FROM my_table BEFORE(STATEMENT =>
  '8e5d0ca9-005e-44e6-b858-a8f5b37c5726');
```

One can even access different versions of the same table in a single query.

```
SELECT new.key, new.value, old.value FROM my_table new
JOIN my_table AT(OFFSET => -86400) old -- 1 day ago
ON new.key = old.key WHERE new.value <> old.value;
```

此外，还提供**CLONE关键字**来快速存储table，（不是物理clone，只是metadata clone），可以非常方便的提供数据库快照（snapshot

3.3.3 MAX-MIN剪枝

需要限制相关查询的数据的访问，以往都使用B+树当索引，限制数据访问，这种方法对事务处理很有效，而在Snowflake中会带来诸多问题，

① 依赖于随机访问，*由于S3存储和数据压缩?*

② 保存索引很占空间，也占数据加载时间

③ 需要用户显示创建索引，与云原生服务的理念大相径庭

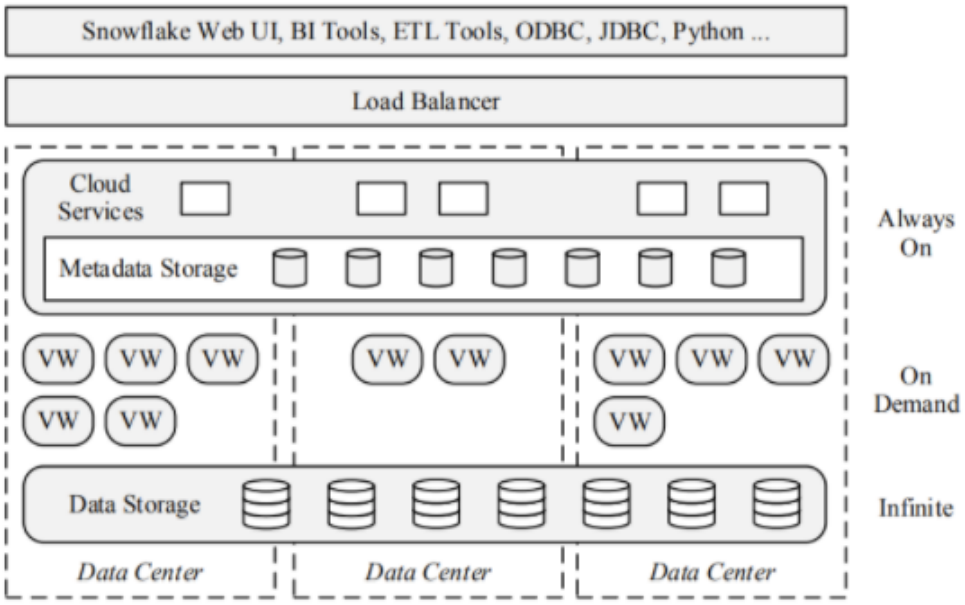
采用MAX-MIN剪枝

做区间判断是否可以跳过文件

优点：pure PaaS experience，对于顺序访问非常有效，可拓展性好，会给每个单独的表文件保留剪枝相关元数据

5.4 功能亮点

- [ ] Snowflake提供了关系型数据仓库的很多功能：全面的SQL支持，ACID事务，强大的性能和可拓展性，和接下来介绍它额外的独到之处



4.1 纯粹的 PaaS

用户可以从任何地方任何环境访问Snowflake，通过web UI

集群不需要关心配置、扩缩容、调优、后期运维

4.2 持续可用性

无停机时间，容错，在线升级

可用区（AZ availability zone）指的是跨越地区的数据中心

节点故障，其他节点会接管任务

云服务层其余服务由多个可用区的无状态节点组成

loader balancer负责负载均衡，平衡请求的服务

snowflake的元数据也存储在S3中

出于性能考虑，虚拟仓库VW不跨可用区（AZ）,跨区需要网络I/O开销过大

所有的服务是无状态的

hard state存储到 事务型K/V store中，kv store 也是通过映射层 mapping layer来访问，使用元数据版本、结构演变

改变元数据结构，向前兼容

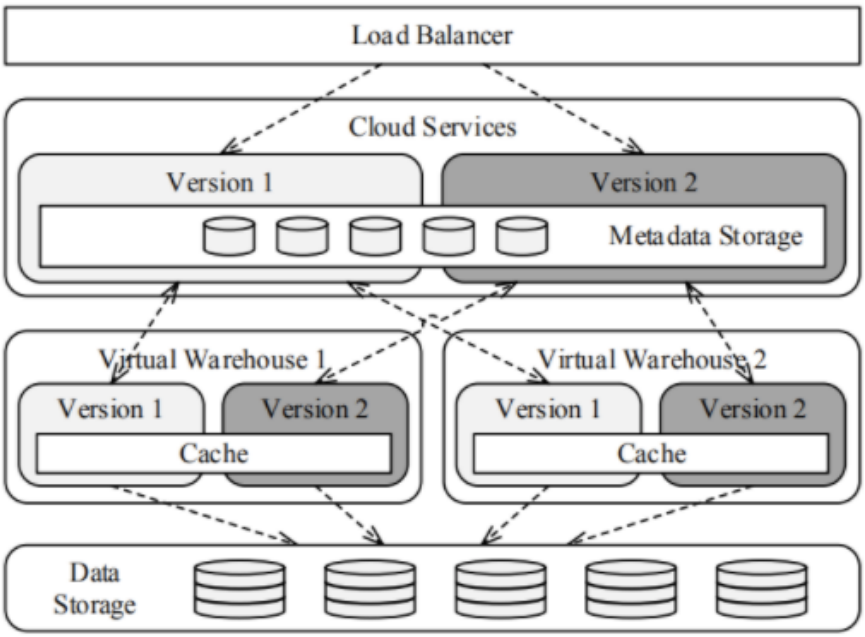


Figure 3: Online Upgrade

在线升级过程：

两个版本snowflake并行运行，一个版本的云服务层只和匹配的VWs通信，

两个版本的云服务层共享相同数据，不同版本的VWs（虚拟仓库集群）共享相同的工作节点和缓存，升级后无需重新装填缓存

4.3 半结构化和无结构化数据

以VARIANT，ARRAY，OBJECT拓展SQL类型

VARIANT类型可以存储任何SQL中的类型（DATE VARCHAR等）

VARIANT列可以用作连接键、分组键、排序键，这使得可用于ETL

用户可以从 json、avro、xml格式转化为variant列

如果需要transformation，可以使用并行数据库，包括join、sorting、aggregatation等操作

4.3.1关系型操作

- extraction提取操作

可以通过字段名或者偏移量提取，支持SQL语法、JavaScript语法类似的提取操作

- flattening展平操作

旋转嵌套的文档分成多行，flattening操作可递归，将文档层次结构转化为适合SQL操作的关系表

- aggregatation聚合操作

与展平操作功能相反

4.3.2面向列的存储和处理

面向列的数据库更适合OLAP

面向行的数据库更适合OLTP

Snowflake采用新型自动类型推断和列存储的方法

将数据存储在混合列格式中

同一类型的数据，系统会从表中删除列，提取出来单独存储

4.3.3 乐观转换

一些SQL里面的类型如（date、time）在JSON、XML里面表示为字符串，在写入、读取时需要反复转化，写入：字符串-->实际类型，读取：实际类型-->字符串，**实际情况是读多写少，那么写入时转化一次就行**

自动转化过程中，一些模棱两可的字段保留转化后的数据和原始数据（双重存储）

5.5 展望

The biggest future challenge for Snowflake is the transition to a full self-service model, where users can sign up and interact with the system without our involvement at any phase.

It will bring a lot of security, performance, and support challenges. We are looking forward to them.

6.项目思路

计划做一个离线的数据仓库，虽然调研发现**未来的演变方向是实时数仓、云原生数仓、流批一体数仓、湖仓一体数仓**，但在短短的一个月实现还是非常困难，因此本项目目标更多是

- [ ] 1.学习并使用各种大数据组件
- [ ] 2.搭建成集群
- [ ] 3.能够顺利构建离线的批处理的数仓
- [ ] 4.进行批量数据分析和可视化展示

完成以上任务便达到预期目标了

预计使用的组件与框架

Hadoop、Hive、Spark、Zookeeper、Sqoop、Flume、Kafka、任意一种OLAP分析引擎（impala、presto、kylin）、可视化组件

7.参考文献

1. 《Major Technical Advancements in Apache Hive》https://dl.acm.org/doi/pdf/10.1145/2588855.2595630
2. 《Apache Hive: From MapReduce toEnterprise-grade Big Data Warehousing》https://dl.acm.org/doi/pdf/10.1145/3299869.3314045
3. 《Industry Paper: Kafka versus RabbitMQ》https://dl.acm.org/doi/pdf/10.1145/3093742.3093908
4. 《Snowflake Elastic Data WareHouse》https://dl.acm.org/doi/pdf/10.1145/2882903.2903741
5. 本人博客 https://blog.csdn.net/weixin_39666736

作业2:通讯数据处理与分析-设计报告

周彬韬 2001210564

2021年-4月-26日

1.项目背景与架构设计

1.1 项目背景

通讯记录的**元信息**蕴含着巨大的价值，我们获取不到加密后的通话内容（当然这也是敏感信息），但是可以获取号码的拨打方和接收方的所在城市，可以获取通话的时间长短、常用沟通的时间等等，这些信息都称为元数据。

对这些数据进行离线分析，可以统计月度、季度、年度话单，可以获取通话记录，从而构建一张巨大的社交网，可以洞察某个人的好友圈，可以推断、洞察从而进行精准化推荐。

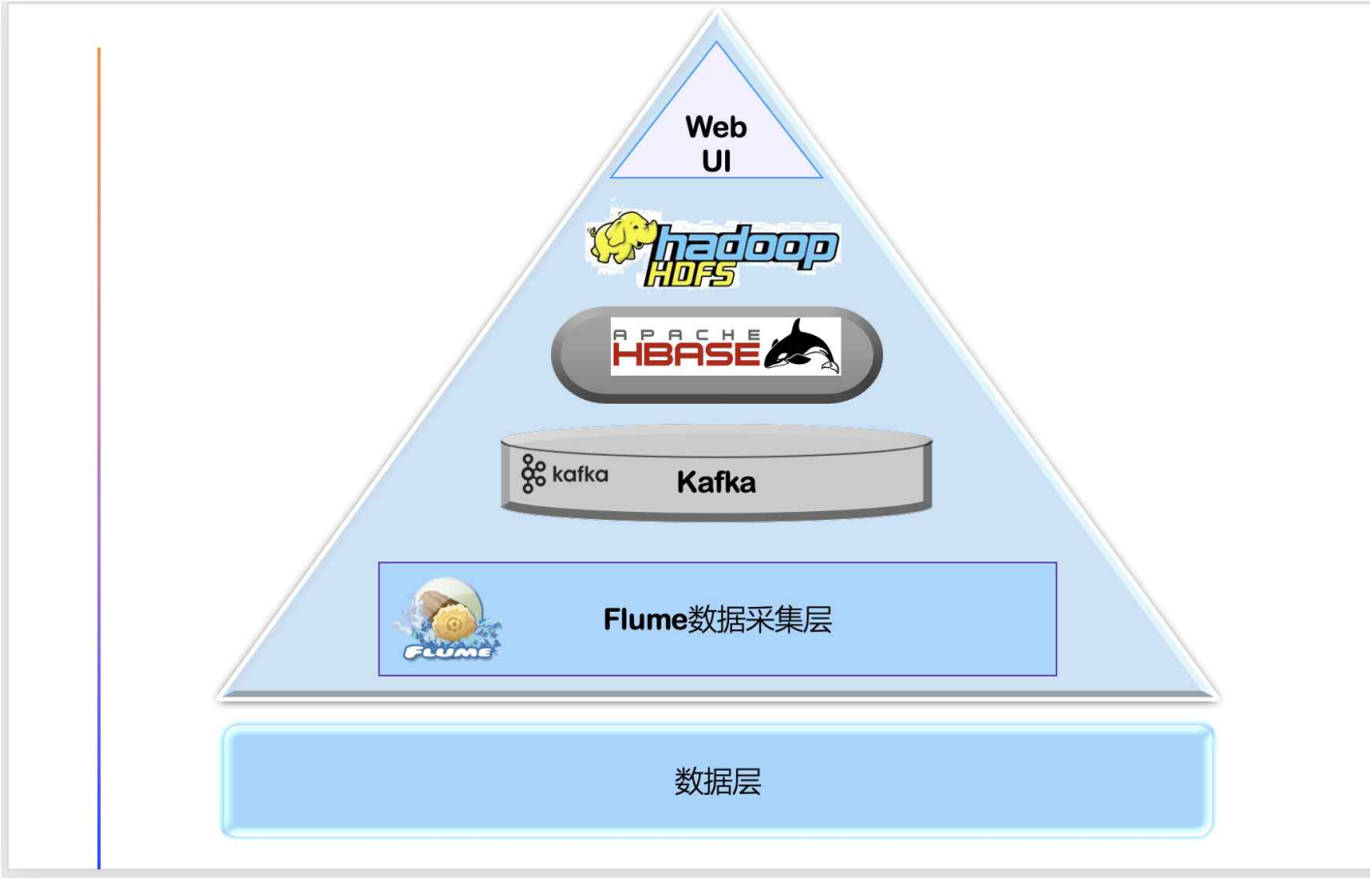
当前收集的数据：统计每天、每月以及每年的每个人的通话次数及时长以及部分个人信息。

1.2 技术选型调研

技术选型需要考虑到：数据量大小、业务需求、行业内经验、技术成熟度、开发维护成本等等因素

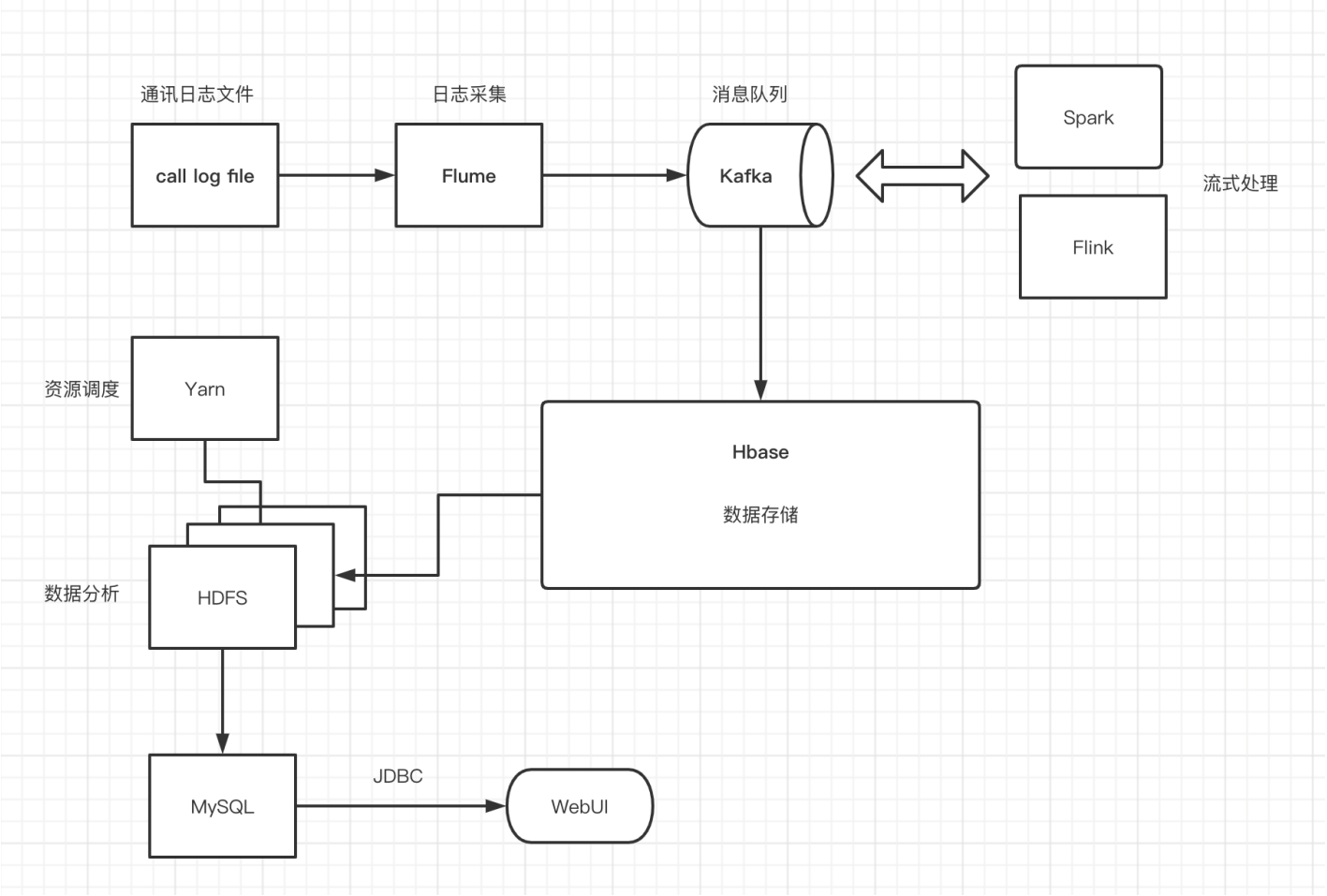
- 数据采集、传输：Flume，Kafka，Sqoop，DataX
- 数据存储：MySQL，HDFS，HBase，Redis，MongoDB
- 数据计算：Hive，Spark，Flink，Storm
- 数据查询：Presto，Kylin，Impala，Druid
- 数据可视化：Superset，QuickBI，DataV
- 任务调度：Azkaban、Oozie
- 集群监控：Zabbix
- 元数据管理：Atlas

1.3 架构设计

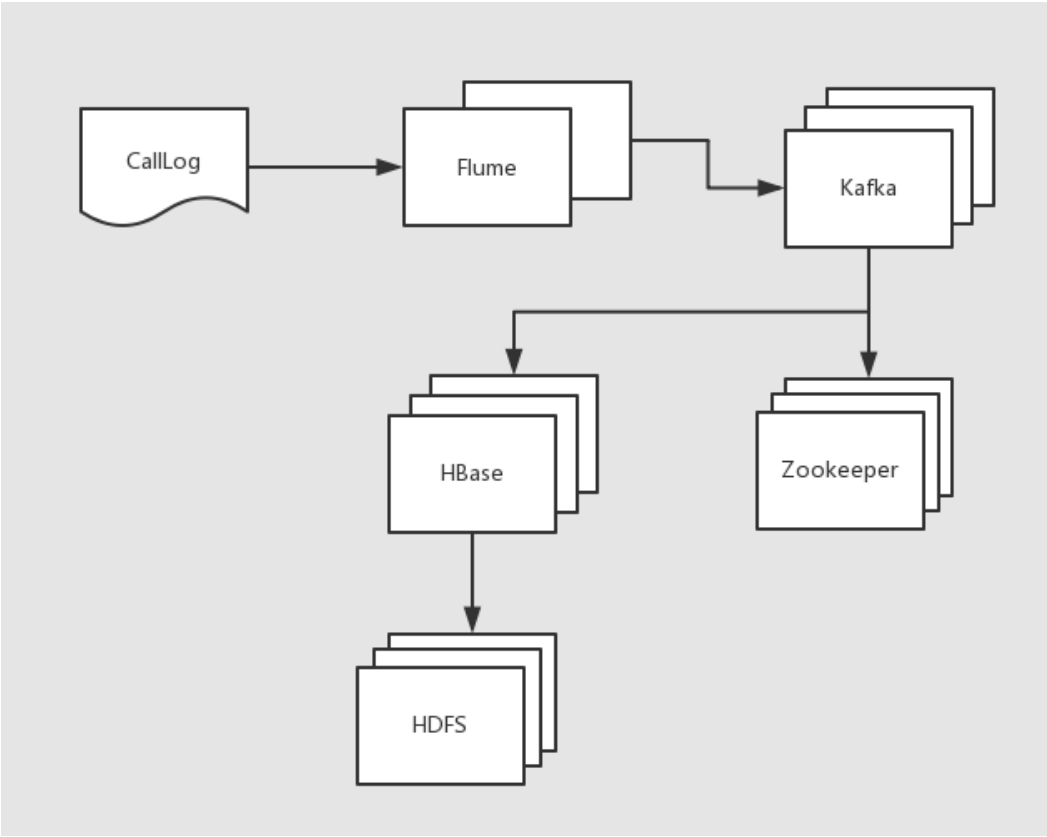


数据层是通话日志文件，由于**通话日志**是**log文件**，需要使用**海量日志采集框架Flume**进行采集，采集后的数据存储在**消息队列kafka**中，均衡生产者和消费者的生产消费速率，**kafka流出的数据输出到Hbase中**（对于kafka中流式数据分析可以采用spark和storm，本项目采用离线批处理，所以没有使用到storm）对存储的数据采用yarn-mapreduce的方式进行分析，由于HBase分析聚合后的数据量不大，所以统计分析后的结果存储在关系型数据库MySQL中，通过JDBC直接访问MySQL，方便将分析后的数据展示在前端WebUI上。

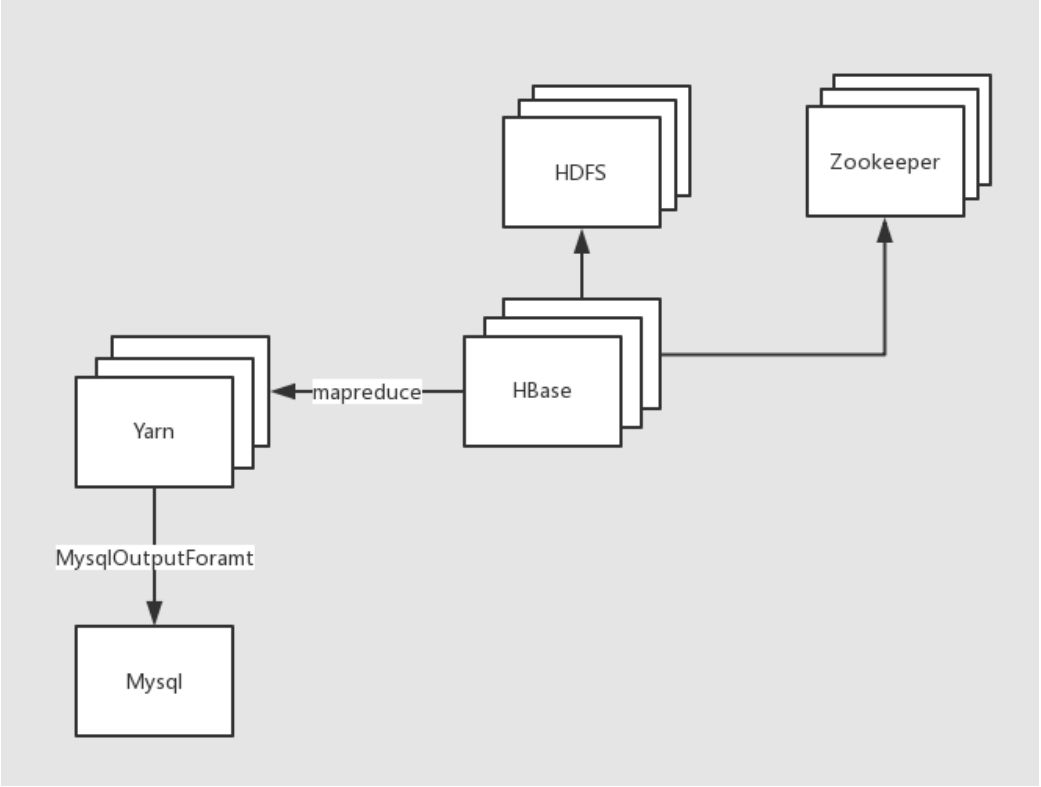
详细的架构流程图如下：


通讯日志文件
日志采集
消息队列
call log file
Flume
Kafka
Spark
Flink
流式处理
资源调度
Yarn
Hbase
数据存储
数据分析
HDFS
MySQL
JDBC
WebUI

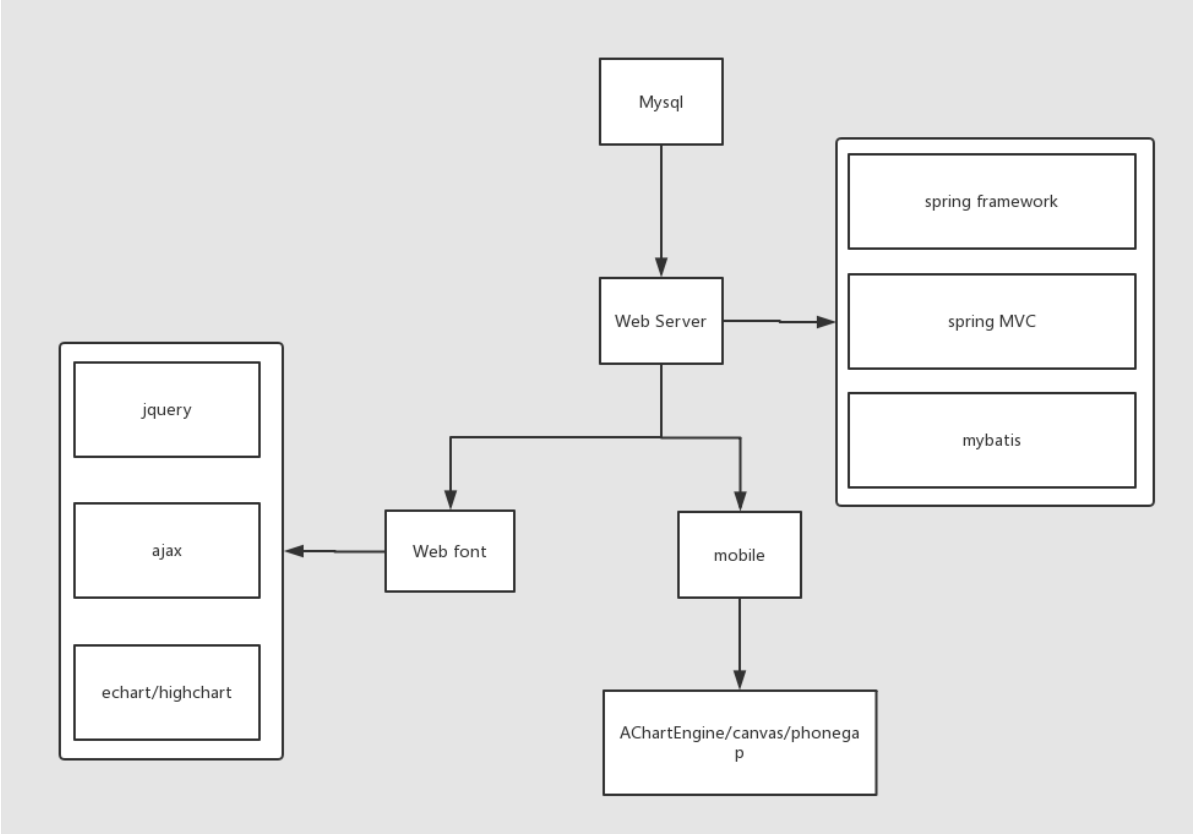
数据消费流程图如下：


CallLog
Flume
Kafka
HBase
Zookeeper
HDFS

数据处理流程图如下：



数据从MySQL到前端展示流程图如下：



2.环境搭建

2.1 系统环境

系统	版本
windows	10
Linux	CentOS7

2.2 集群搭建

框架、组件版本选择

框架	版本
Hadoop	3.1.3
Flume	1.9.0
Kafka	2.4.1
Spark	3.0.0
HBase	2.0.5
MySQL	5.7.6

集群分配

服务名称	子服务	服务器 hadoop102	服务器 hadoop103	服务器 hadoop104
HDFS	NameNode	✅		
	DataNode	✅	✅	✅

	SecondaryNameNode			✅
Yarn	NodeManager	✅	✅	✅
	Resourcemanager		✅	
Zookeeper	Zookeeper Server	✅	✅	✅
Flume				✅
Kafka		✅	✅	✅
Hbase	HMaster	✅		
	HRegionserver	✅	✅	✅

2.3硬件环境

	hadoop102	hadoop103	hadoop104
内存	4G	2G	2G
CPU	2核	1核	1核
硬盘	50G	50G	50G

3.数据生成

3.1 数据结构设计

搭建好了集群环境，接下来需要采集callLog日志文件

由于真实环境下的通信数据在尝试过后发现难以获取，爬取得到的数据杂乱不全，因此选择模拟数据产生

我们将在HBase中存储两个电话号码，以及通话建立的时间和通话持续时间，最后再加上一个flag作为判断第一个电话号码是否为主叫。姓名字段的存储我们可以放置于另外一张表做关联查询，设计数据结构和字段如下：

通话讯息表

字段名 / 列名	含义解释	举例
call1	拨打方 - 电话号码	127-1231-1231
call1_name	拨打方 - 姓名	小明
call2	接收方 - 电话号码	158-2131-7867
call2_name	接收方 - 姓名	小方
start_time	建立通话的时间	2021-04-01 18:37
start_time_ts	建立通话的时间戳	
duration	通话持续时间（秒）	600

个人基本信息表

字段名 / 列名	含义解释	举例（字段可能为空）
call	电话号码	127-1231-1231
call_name	名字	小明
flag	主动打电话还是被动接电话。 拨打方：true，接收方：false	true
age	年龄	24
address	地址	北京-海淀区

3.2 编程实现

数据模拟产生的代码逻辑：

a) 创建Java集合类存放模拟的电话号码和联系人；

b) 随机选取两个手机号码当作“主叫”与“被叫”（注意判断两个手机号不能重复），产出call1与call2字段数据；

c) 创建随机生成通话建立时间的方法，可指定随机范围，最后生成通话建立时间，产出date_time字段数据；

d) 随机一个通话时长，单位：秒，产出duration字段数据；

e) 将产出的一条数据拼接封装到一个字符串中；

f) 使用IO操作将产出的一条通话数据写入到本地文件中；

模拟生成的数据拼接成日志，并写入本地文件

Java

```java
//拼接日志
 private String productLog() {
    int call1Index = new Random().nextInt(phoneList.size());
    int call2Index = -1;
    String call1 = phoneList.get(call1Index);
    String call2 = null;
    while (true) {
        call2Index = new Random().nextInt(phoneList.size());
        call2 = phoneList.get(call2Index);
        if (!call1.equals(call2)) break;
    }
   //随机生成通话时长(30分钟内_0600)
    int duration = new Random().nextInt(60 * 30) + 1;
    //格式化通话时间，使位数一致
    String durationString = new DecimalFormat("0000").format(duration);
    //通话建立时间:yyyy-MM-dd,月份：0~11，天：1~31
    String randomDate = randomDate("2017-01-01", "2018-01-01");
    String dateString = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(Long.parseLong(randomDate));
    //拼接log日志
    StringBuilder logBuilder = new StringBuilder();
    logBuilder.append(call1).append(",").append(call2).append(",").append(dateString).append(",")
            .append(durationString);
```

```
        System.out.println(logBuilder);
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return logBuilder.toString();
    }
    //将产生的日志写入到本地文件calllog中
    public void writeLog(String filePath, ProduceLog productLog) {
        OutputStreamWriter outputStreamWriter = null;
        try {
            outputStreamWriter = new OutputStreamWriter(new FileOutputStream(filePath, true), "UTF-8");
            while (true) {
                String log = productLog.productLog();
                outputStreamWriter.write(log + "\n");
                outputStreamWriter.flush();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                assert outputStreamWriter != null;
                outputStreamWriter.flush();
                outputStreamWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

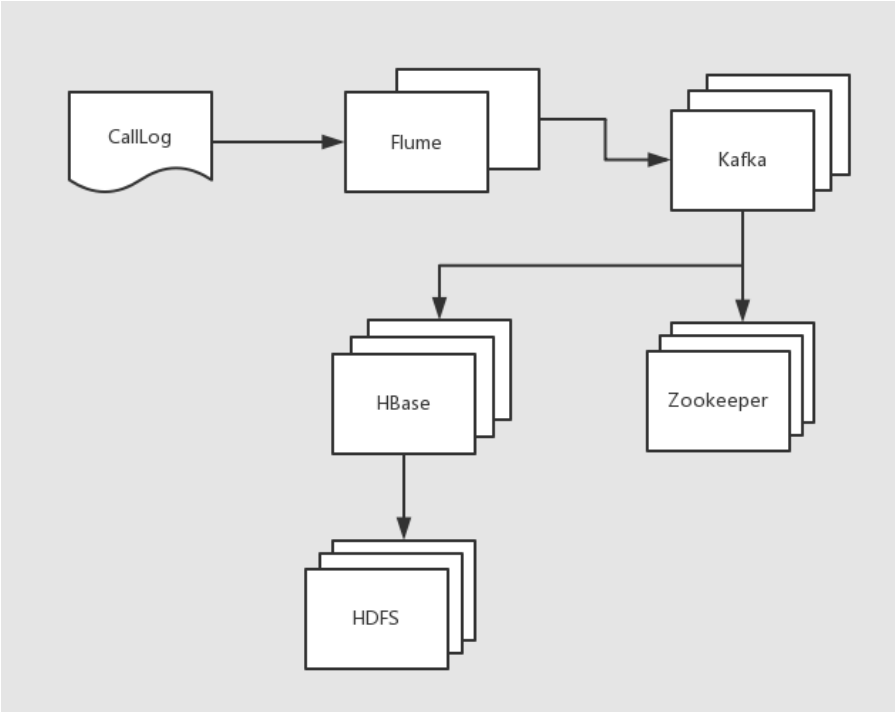
日志生成任务编写脚本，模拟源源不断产生日志数据

Bash

```
#!/bin/bash
java -cp /home/zbt/call/ producer.jar com.atguigu.producer.ProductLog /home/zbt/call/calllog.csv
```

4.数据处理

数据生成描述了如何模拟产生数据，这些数据最终是存在log文件中的，接下来需要进行真正的采集和处理，大体流程是将源源不断产生的实时数据通过flume采集到kafka然后供给hbase消费存储。



常用pipeline：线上数据 --> flume --> kafka --> HDFS

4.1 数据采集与消息队列存储过渡

思路：

a) 配置kafka，启动zookeeper和kafka集群；

b) 创建kafka主题；

c) 启动kafka控制台消费者（此消费者只用于测试使用）；

d) 配置flume，监控日志文件；

e) 启动flume监控任务；

f) 运行日志生产脚本；

g) 观察测试。

4.2 数据消费与存储

这个阶段可以引入flink和spark streaming进行流式数据实时分析，由于时间有限，没有在项目实践中加这个环节。

仅仅将消息队列kafka中读取出来的数据引入到HBase中存储

需要新建一些数据处理类，如HbaseConsumer、PropertiesUtil、HBaseUtil、HBaseDAO等等

4.3 数据查询

使用scan查看HBase中是否正确存储了数据，同时尝试使用过滤器查询扫描指定通话时间点的数据。进行该单元测试前，需要先运行数据采集任务，确保HBase中已有数据存在。

涉及到的查询代码较多，举例说明：已知要查询的手机号码以及起始时间节点和持续时间，查询该节点范围内的该手机号码的通话记录

拼装startRowKey和stopRowKey，即扫描范围， rowkey： 分区号_手机号码1_通话建立时间_手机号码2_主(被)叫标记_通话持续时间 01_15837312345_20170527081033_1_0180

比如按月查询通话记录，则startRowKey举例： regionHash_158373123456_20170501000000

stopRowKey举例： regionHash_158373123456_20170601000000

如果查找所有的，需要多次scan表，每次scan设置为下一个时间窗口即可，该操作可放置于for循环中

4.4 查询优化设计

使用HBase查找数据时，尽可能的使用rowKey去精准的定位数据位置，而非使用ColumnValueFilter或者SingleColumnValueFilter，按照单元格Cell中的Value过滤数据，这样做在数据量巨大的情况下，效率是极低的——如果要涉及到全表扫描。所以尽量不要做这样可怕的事情。注意，这并非ColumnValueFilter就无用武之地。现在，我们将使用协处理器，将数据一分为二。

思路：

a) 编写协处理器类，用于协助处理HBase的相关操作（增删改查）

b) 在协处理器中，一条主叫日志成功插入后，将该日志切换为被叫视角再次插入一次，放入到与主叫日志不同的列族中。

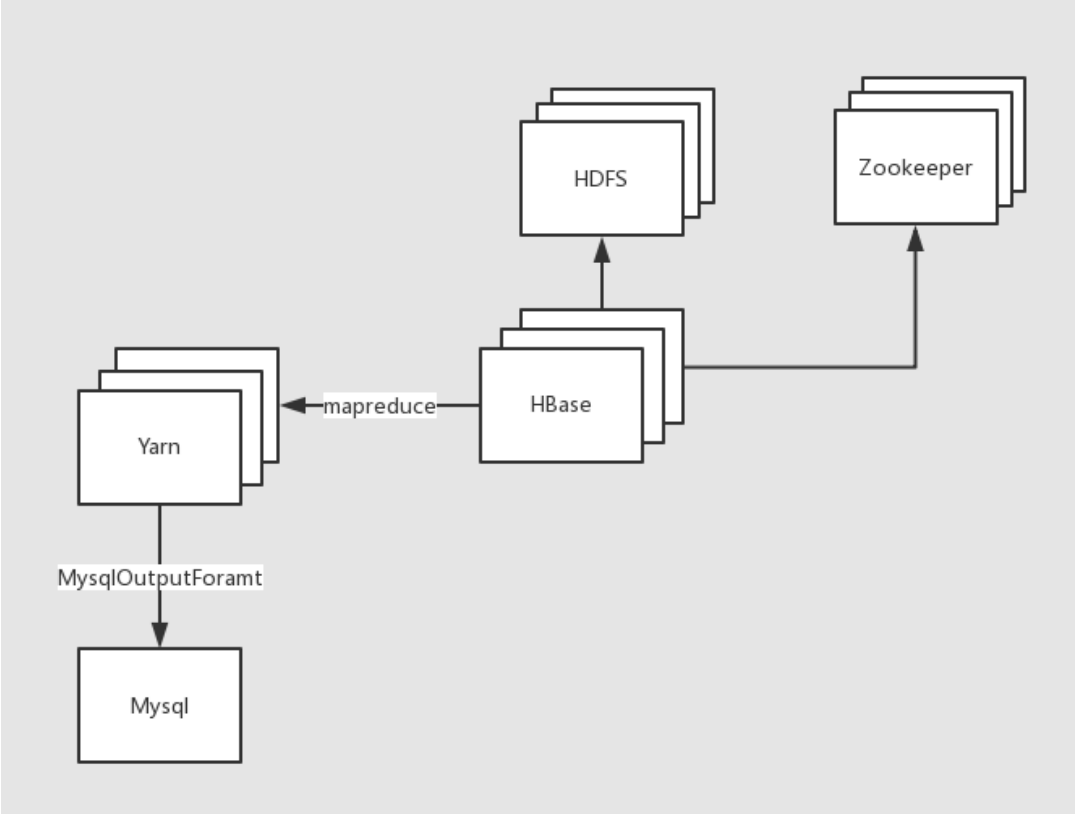
c) 重新创建hbase表，并设置为该表设置协处理器。

d) 编译项目，发布协处理器的jar包到hbase的lib目录下，并群发该jar包

e) 修改hbase-site.xml文件，设置协处理器，并群发该hbase-site.xml文件

5.数据挖掘与可视化

数据已经完整的采集到了HBase集群中，这次我们需要对采集到的数据进行分析，统计出我们想要的结果。注意，在分析的过程中，我们不一定会采取一个业务指标对应一个mapreduce-job的方式，如果情景允许，我们会采取一个mapreduce分析多个业务指标的方式来进行任务。



业务指标：

a) 用户每天主叫通话个数统计，通话时间统计。

b) 用户每月通话记录统计，通话时间统计。

c) 用户之间亲密关系统计。（通话次数与通话时间体现用户亲密关系）

5.1 需求分析

根据需求目标，设计出下述表结构。我们需要按照时间范围（年月日），结合MapReduce统计出所属时间范围内所有手机号码的通话次数总和以及通话时长总和。

思路：

a) 维度，即某个角度，某个视角，按照时间维度来统计通话，比如我想统计2021年所有月份所有日子的通话记录，那这个维度我们大概可以表述为2021年*月*日

b) 通过Mapper将数据按照不同维度聚合给Reducer

c) 通过Reducer拿到按照各个维度聚合过来的数据，进行汇总，输出

d) 根据业务需求，将Reducer的输出通过Outputformat把数据

数据输入：HBase

数据输出：Mysql

5.2 MySQL表结构设计

我们将分析的结果数据保存到Mysql中，以方便Web端进行查询展示。

1) 表：db_telecom.tb_contacts

用于存放用户手机号码与联系人姓名

无法复制加载中的内容

2) 表：db_telecom.tb_call

用于存放某个时间维度下通话次数与通话时长的总和。

无法复制加载中的内容

3) 表：db_telecom.tb_dimension_date

用于存放时间维度的相关数据

无法复制加载中的内容

4) 表：db_telecom.tb_intimacy

用于存放所有用户用户关系的结果数据。（作业中使用）

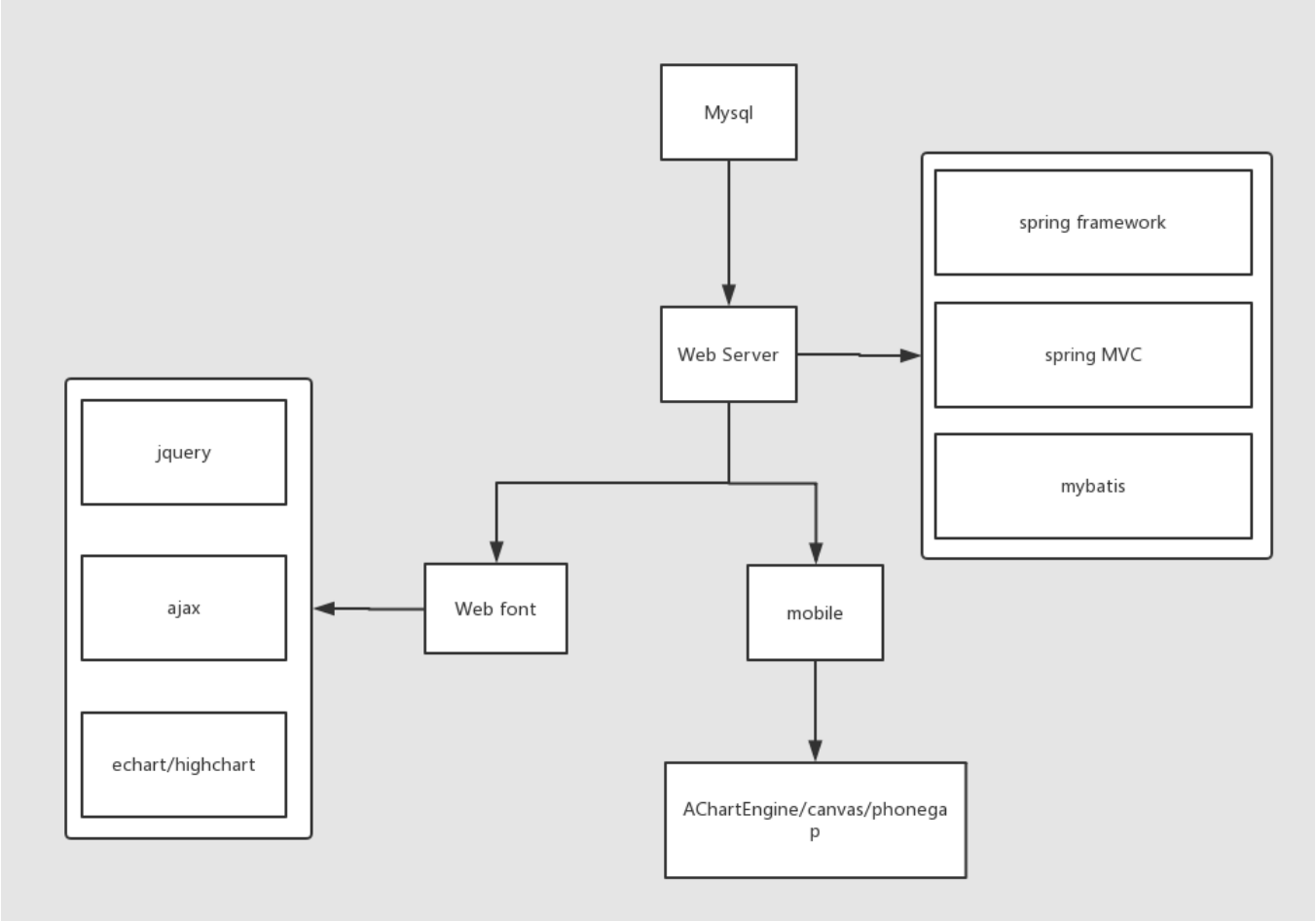
无法复制加载中的内容

5.3 类设计

无法复制加载中的内容

5.3 数据展示

接下来需要将用户按照不同维度查询出来的结果，展示到web页面上。



类表设计如下

无法复制加载中的内容

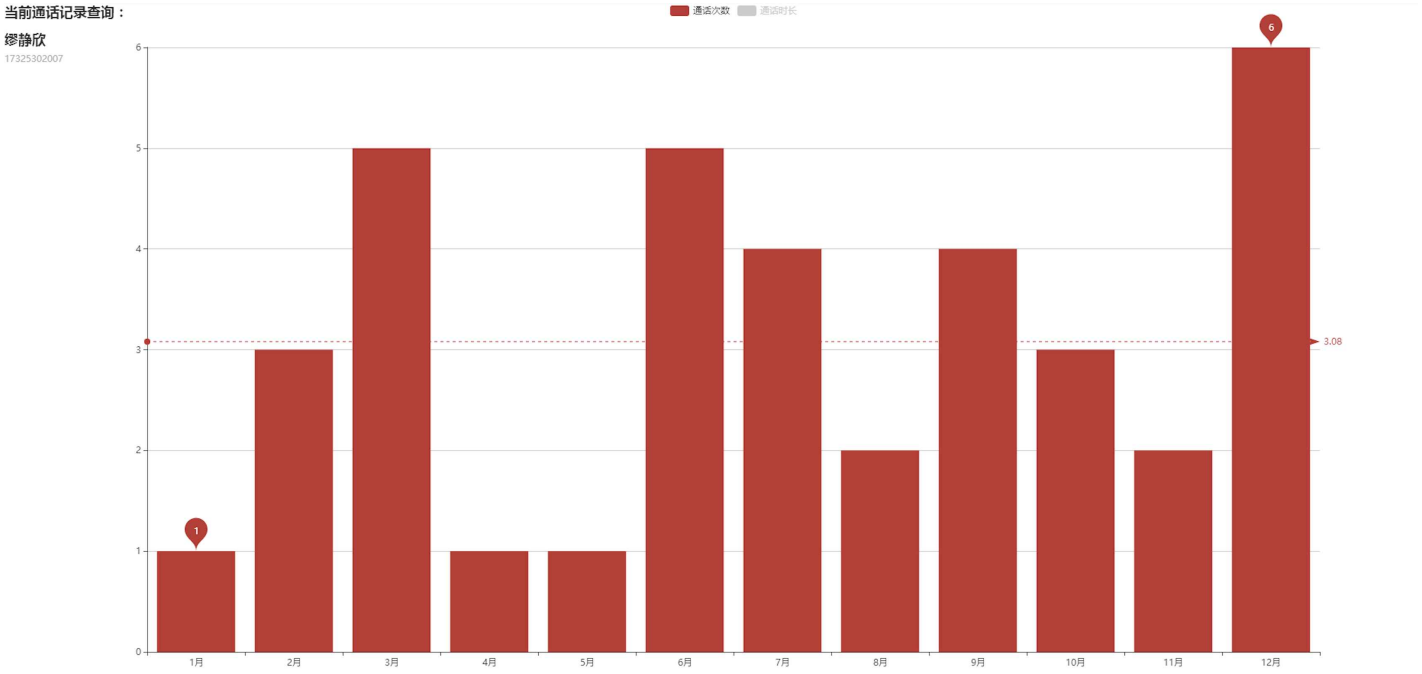
5.4 定时脚本

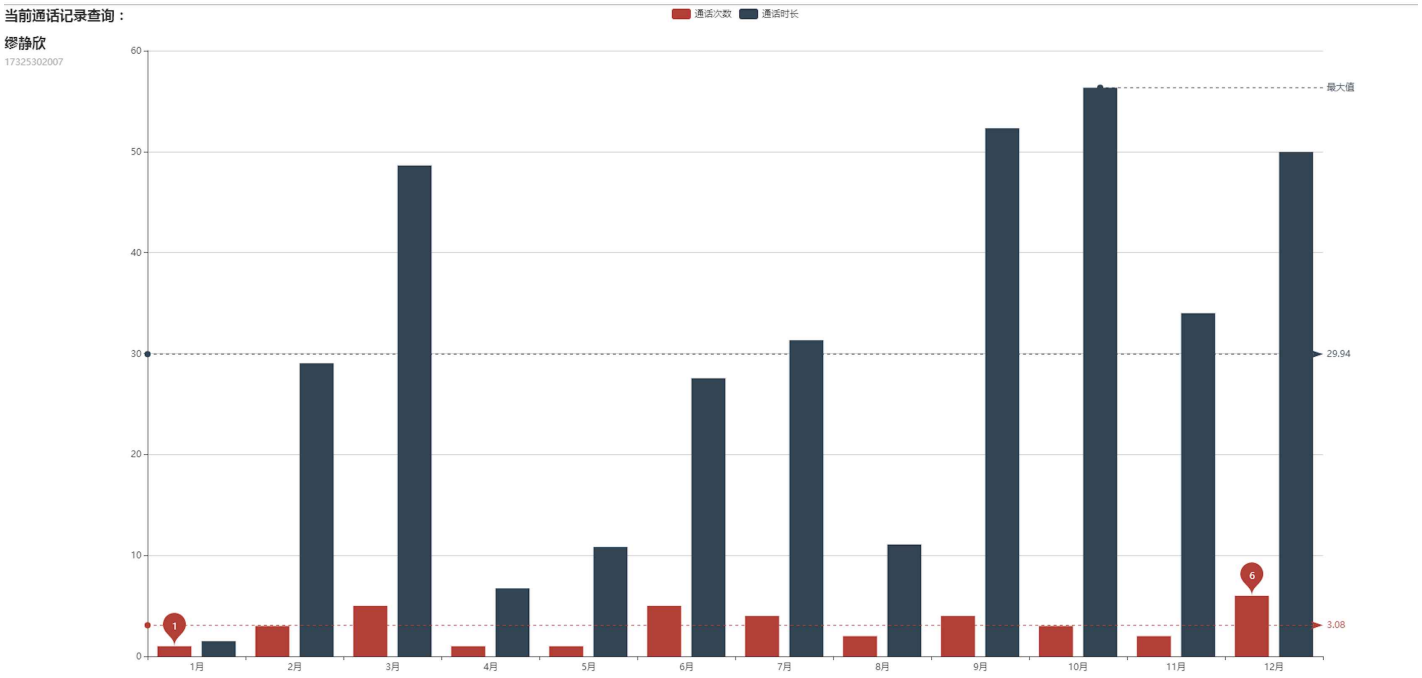
新的数据每天都会产生，所以我们每天都需要更新离线的分析结果，所以此时我们可以用各种各样的定时任务调度工具来完成此操作。此例我们使用crontab来执行该操作。

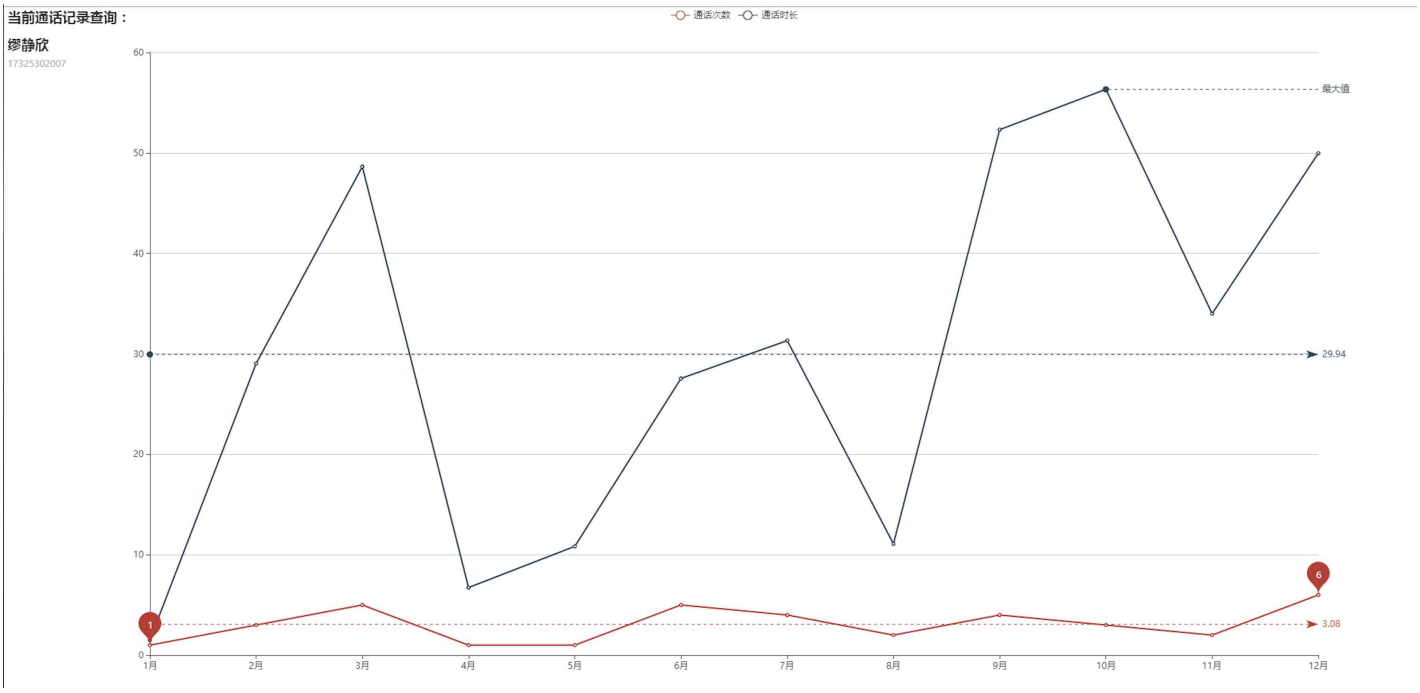
Shell

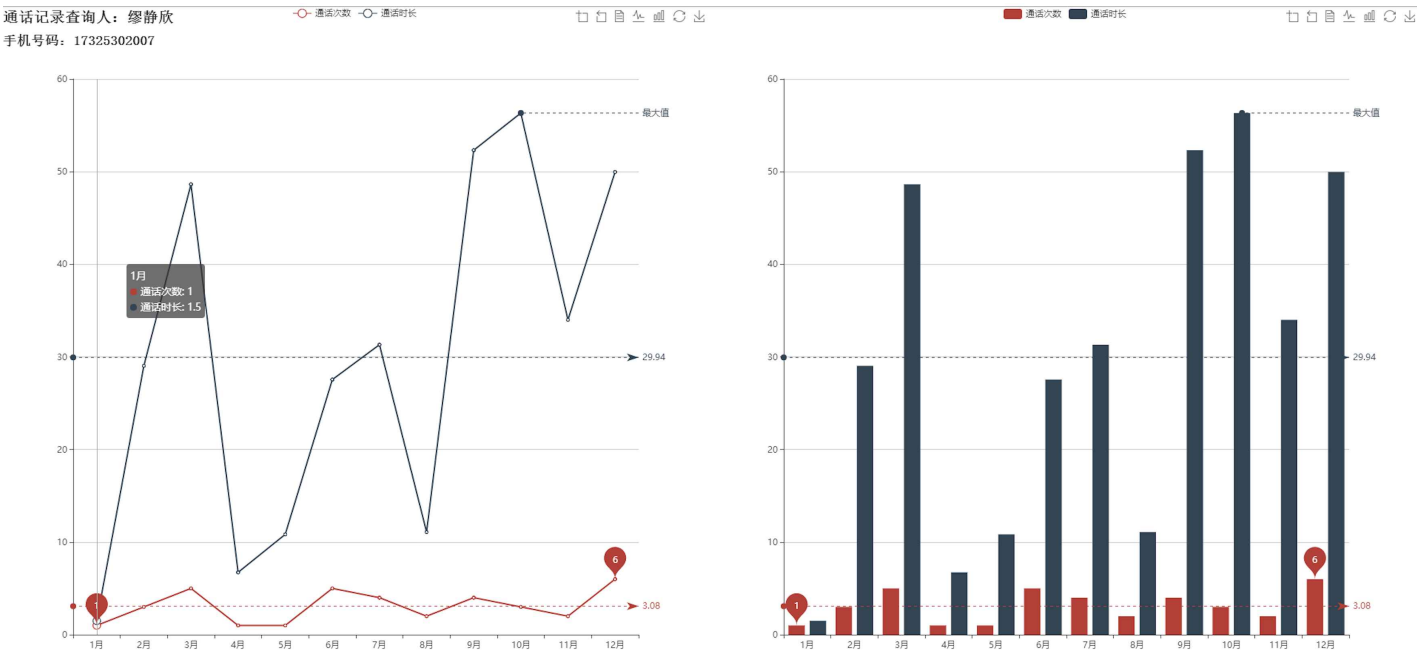
```
# .----------------------------------------minute(0~59)
# " .---------------------------------------hours(0~23)
# " " .--------------------------------------day of month(1~31)
# " " " .-------------------------------------month(1~12)
# " " " " .------------------------------------day of week(0~6)
# " " " " " .-----------------------------command
# " " " " " "
# " " " " " "
0 0 * * * /home/zbt/call/analysis.sh
```

6.项目运行截图


当前通话记录查询：
缪静欣
17325302007
通话次数
通话时长
6
3.08
1
1月
2月
3月
4月
5月
6月
7月
8月
9月
10月
11月
12月


当前通话记录查询：
缪静欣
17325302007
通话次数
通话时长
最大值
29.94
3.08
6
1
1月
2月
3月
4月
5月
6月
7月
8月
9月
10月
11月
12月





7.总结

7.1 项目难点

在设计和实现过程中遇到的主要难点有：

- [ ] 各个大数据组件的兼容问题，利用三台虚拟机进行集群环境的搭建
- [ ] 数据爬取由于数据脏乱、字段缺失，爬取ip被封等等问题，尝试失败后，转换思路，设计数据结构，模拟数据生成，并写出日志到本地文件系统。
- [ ] 各个组件的环境配置部分。
- [ ] linux下一些脚本的编写，如：文件集群xsyc分发、crontab定时脚本的编写等。
- [ ] 查询过程中一行行扫描，速度太慢，从而进行查询优化的设计。
- [ ] 最终mysql表的设计与MySQL到使用superset进行前端可视化展示。

7.2 小结

本文是通讯数据处理分析的设计报告，先从宏观上介绍了项目背景，接下来介绍了技术选型与环境搭建，然后分章节介绍了数据分析处理与可视化的流程和大体设计思路，最后是本设计文档的总结部分，感谢帮助过我的小伙伴们。