15 | Hive：来来去去的DSL，永生不死的SQL

2021-10-25 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 19:06 大小 17.50M

你好，我是徐文浩。

通过过去几篇论文的解读，相信现在你已经深入掌握好了大数据系统的基本知识。而在 Google 的这些论文发表之后，整个工业界也行动起来了。很快，我们就有了开源的 GFS 和 MapReduce 的实现 Hadoop，以及 Bigtable 的实现 HBase。

这些系统，的确帮助我们解决了很多海量数据处理的问题。并且这些系统设计得也还算易用，作为工程师，我们基本不太需要对分布式系统本身有深入地了解，就能够使用它们

不过，这些系统都还很原始和粗糙，随便干点什么都很麻烦。所以自然而然地，工程师们就会通过封装和抽象，来提供更好用的系统。这次的系统，不再是来自于 Google，而是 Facebook 了，它的名字叫做 Hive。

Facebook 在 2009 年发表了 Hive 的论文《Hive: a warehousing solution over a map-reduce framework》，并把整个系统开源。而在 2010 年，Facebook 又把这篇论文丰富了一下，作为《Hive-a petabyte scale data warehouse using hadoop》发表出来。这两篇论文其实内容上基本是一致的，后一篇的内容会更完整、详细一些，你可以按照自己的需要有选择性地阅读。

Hive 基于 Hadoop 上的 HDFS 和 MapReduce，提供了一个基本和 SQL 语言一致的数据仓库方案。我们今天就来深入剖析一下 Hive 的论文，看看 Facebook 是怎么基于 Hadoop 搭建这个数据仓库的。在学完这节课之后，你会理解到这样两个关键点：

在数据库系统设计的时候，如何把查询语言和计算框架分离，做好对现有系统的复用。

为什么选择“标准”很重要，选择了合适的“标准”会让你的项目和产品更有生命力。

那接下来，就和我一起来读一读 Hive 的论文吧。

Hive 的设计目标

在 Hive 论文的摘要和概述中，其实已经把 Hive 系统的设计目标讲得非常清楚了。

对于 Facebook 当时的数据体量来说，如果使用商业的关系型数据库，**面临的瓶颈是计算时间**，可能一个每日生成的数据报表一天都跑不完，或者一个临时性的分析任务（ad-hoc）也需要等待很长时间。**从这个角度来看，我们等不起机器的时间。**

而如果用 Hadoop 和 MapReduce 来实现，系统的伸缩性的确是没有问题了。通过拥有大量节点的 Hadoop 集群，这些任务都能比较快地跑完。但是，撰写 MapReduce 的程序就变成了一个麻烦事儿。

这是因为，MapReduce 提供的编程模型和接口仍然太“底层”（low-level）了。即使做一个简单的 URL 访问频次的统计，你也需要通过写一段代码来完成。而不少分析师并不擅长写程序，稍微复杂一些的统计逻辑，可能需要好几个 MapReduce 任务，花上个几天时间。**从这个角度来看，我们等不起人的时间。**

所以针对这两个瓶颈问题，Hive 的解法和思路也很明确，那就是通过一个系统，我们可以**写 SQL 来执行 MapReduce 任务**。这样，分析师只需要花上几分钟写个 SQL，而

Hadoop 再通过十几分钟运行这个 SQL。最好还能有个 Web 界面而不需要命令行，我们就可以鱼和熊掌兼得，做出既快又好的数据分析工作。

其实，我在 MapReduce 的论文解读里，已经给过一个例子。一个通过 Group By 关键字进行统计的 SQL，和一个 MapReduce 任务，是等价的。

```
SELECT URL, COUNT(*) FROM url_visit_logs GROUP BY URL ORDER BY URL
```

> 你可以回头看一下第 6 讲里面对于 MapReduce 任务的讲解。

Hive 的数据模型

但是，从原始的 MapReduce 任务，到 SQL 语言之间，其实有很多鸿沟。

首先就是序列化和类型信息，基于 SQL 的数据库，有明确的表结构，每个字段的类型也都是明确的。而原先的 MapReduce 里，是没有明确的字段以及字段类型的定义的。所以，填补这个鸿沟的**第一步，就是先要在数据的输入输出部分加上类型系统**。

Google 的 MapReduce 论文里，把对于输入数据的解析，完全交给了开发 MapReduce 程序的用户自己。对于开发者来说，输入就是一行字符串。而到了 Hadoop 这个开源系统，这一部分稍微进化了一下，Hadoop 把针对输入输出的解析从 MapReduce 的业务程序里面抽离了出来。

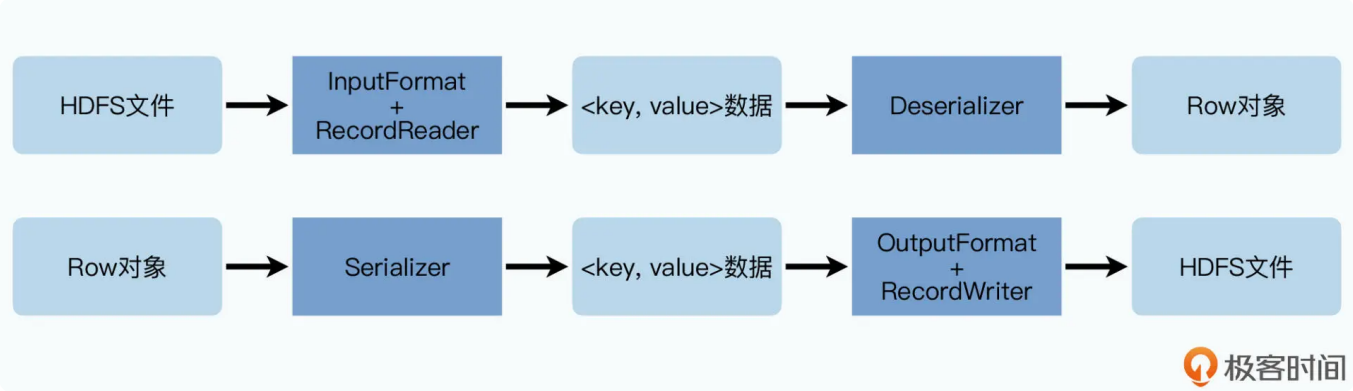
在 Hadoop 里，你可以自定义一系列的 InputFormat/OutputFormat，以及对应 RecordReader/RecordWriter 的实现。这些接口，把通过 MapReduce 读写 HDFS 上的文件内容，以及进行序列化和反序列化的过程单独剥离出来了。这样，开发人员在实现具体的 map 和 reduce 函数的时候，只需要关心业务逻辑就好了，因为在 map 和 reduce 函数里输入输出的 key 和 value，都是已经包含了类型的 Java 对象了。

```
public class ExampleMapper extends Mapper<CustomKey, CustomValue, CustomKey, I
    @Override
    protected void map(CustomKey key, CustomValue value, Mapper.Context contex
        context.write(key,value.getIntValue());
    }
```

```
6 }
```

> 在 Hadoop 里，通过 InputFormat/OutputFormat，分离了序列化和 map/reduce 函数，简化了工程师的工作。

而 Hive 则很好地利用了这个 Hadoop 的这个功能特性。对于 Hive 的使用者来说，不再有“输入文件”和“输入文件的格式”这样的概念了，它读取的直接是 Hive 里的一张“**表**”（Table），拿到的“格式”，也是和数据库概念里面一样的“**行**”（Row）。通过 InputFormat 解析拿到的一个 key-value 对，其实就是一“行”数据。



Hive通过在InputFormat/OutputFormat外，添加了一层自己的SerDer来把数据转化成数据库里的“行”

不过，既然是一张数据库表，那么 Hive 需要的也不是 key-value 对，而是“行”里面的一个个预先已经定义好的“**列**”（Column）。所以，Hive 在拿到 key-value 对之后，会再通过论文里所说的 SerDer，也就是序列化器 Serializer 和反序列化器 Deserializer，变成一个实际的 Row 对象，里面包含了一个个 Column 的值。

对于所有这些列，Hive 支持以下这些类型：

> 整数类型里，它支持 1 字节（tinyint）、2 字节（smallint）、4 字节（int32）、8 字节（int64）的整数；
>
> 浮点数类型里，它支持单精度（float）和双精度（double）的浮点数；
>
> Hive 当然也支持字符串，并且没有根据字符串的长度区分类型。

而在这些常见的类型之外，Hive 和我们之前讲解过的 Thrift 一样，还支持结构化的类型，包括数组（list）、关联数组（associated-array），以及自定义的结构体（struct）。

这些支持就让 Hive 的使用变得更加地灵活，这也就类似于现在的 MySQL 支持 JSON 字段一样，使得我们每一行数据可以支持更复杂的结构。

比如说，我们在一节课里，可能会有两篇推荐阅读的文章链接，我们就可以通过一个 List 的结构存储下来。这一点其实和 Bigtable 类似，在 Hive 里我们一样会选择用一张“**宽表**”，把一个对象的所有字段都通过一张表存下来。

在大数据领域，表通常不会轻易通过数据库里的“外键”进行多张表的关联，而是都会用这种“宽表”的形式，主要原因是表和表之间的 Join 操作很有可能要跨越不同的服务器，这会带来比较高的成本和比较差的性能。

Hive 的数据存储

看到这里，相信你也已经想到了，**Hive 的表的底层数据，其实就是以文件的形式存放在 HDFS 上的。**而且存储的方式也非常直观，就是一张 Hive 的表，就占用一个 HDFS 里的目录，里面会存放很多个文件，也就是实际的数据文件。而通过 Hive 运行 HQL，其实也是通过 MapReduce 任务扫描这些文件，获得计算的结果。

不过，通过 MapReduce 来执行 SQL 进行数据分析，有一个巨大的问题，就是**每次分析都需要进行全表扫描**。和 MySQL 不同，HDFS 上的这些文件可没有什么索引，而 Hive 要一直到 0.6.0 版本才会加上基于列存储的 RCFile 格式。对于一开始的 Hive 版本来说，你可以认为所有的数据，都是用类似于 CSV 这样的纯文本的格式存储下来的。

而随着时间的累积，我们的数据会越来越多。可能一开始每天只有 100GB 的访问日志，作为一张表存放在 Hive 里了，但是 3 个月过去之后，这张表就变成了 9TB 的大小。即使我们只是要访问过去一天的数据，也都必须扫描这 9TB 的数据。这样的效率，我们当然是无法接受的。

自然而然地，Hive 采用了一个数据库里面常用的解决办法，也就是**分区**（Partition）。Hive 里的分区非常简单，其实就是**把不同分区的文件，放到表的目录所在的不同子目录下**。

论文里举的例子就是按照日期和所在的国家进行分区的情况，通过在表的目录里添加两层目录结构，并分别通过 ds=20090101 和 ctry=CA 标注出来。这样一来，当我们执行的

SQL 的查询条件，有日期或者国家的过滤条件的时候，就可以不用再扫描表目录下所有的子目录了，而是只需要筛选出符合条件的那些子目录就好了。

```
/wh/T/ds=20090101/ctry=CA
/wh/T/ds=20090101/ctry=US
/wh/T/ds=20090102/ctry=CA
/wh/T/ds=20090102/ctry=US
……
/wh/T/ds=20091231/ctry=CA
/wh/T/ds=20091231/ctry=US
```

> 通过两层目录结构，我们如果按照国家或者日期过滤数据进行分析的时候，读取的文件数量就可以大大减少。

```
FROM (SELECT a.status, b.school, b.gender
      FROM status_updates a JOIN profiles b
           ON (a.userid = b.userid and
               a.ds='2009-03-20')
     ) subq1
```

> 论文中的这段子查询，会根据ds='2009-03-20'这个查询条件自动只获取ds=2009-03-20这个分区，也就是子目录里的文件。

而在分区之外，Hive 还进一步提供了一个**分桶**（Bucket）的数据划分方式。

在分区之后的子目录里，Hive 还能够让我们针对数据的某一列的 Hash 值，取模之后分成多个文件。这个分桶，虽然不能让我们在分析查询数据的时候，快速过滤掉数据不进行检索，但是却提供了一个**采样分析**的功能。

比如，我们按照日志的 log id 分成 100 个桶，那么我们可以很容易地在分析数据的时候，指定只看 5 个桶的数据。这样，我们就有了一个采样 5% 的数据分析功能。

这样的数据采样，可以帮助数据分析师快速定性地判断问题，等到有了一些初步结论之后，再在完整的数据集上运行，获得更精确的结果。

复制代码

```
/wh/T/ds=20090101/ctry=CA/000000_0
/wh/T/ds=20090101/ctry=CA/000001_0
/wh/T/ds=20090101/ctry=CA/000002_0
……
/wh/T/ds=20090101/ctry=CA/000098_0
/wh/T/ds=20090101/ctry=CA/000099_0
```

> 如果分成 100 个桶，即使分区里的文件大小比一个 Block 小，仍然会根据桶拆分成多个文件。

Hive 的架构与性能

前面我们说过，数据表的字段和类型会通过 SerDer 来解析确定，而数据的分区分桶其实就是通过目录和文件命名来区分的。所以，Hive 的整个系统架构并不复杂，其实也就是三个部分。

第一部分：一系列的对外接口

Hive 支持命令行、Web 界面，以及 JDBC 和 ODBC 驱动。而 JDBC 和 ODBC 的驱动则是通过 Hive 提供的 Thrift Service，来和实际的 Hive 服务通信。这些接口最终都把外部提交的 HQL，交给了 Hive 的核心模块，也就是“驱动器”。

第二部分：驱动器

Hive 的核心“驱动器”也不复杂，它其实并不需要提供任何分布式数据处理的功能，而只是做了把 HQL 语言变成一系列待执行的 MapReduce 的任务这件事情。最终的实际任务，是由 Hadoop 里的 MapReduce 计算框架做到的。这个驱动器主要也可以拆分成三个过程。

首先是一个编译器。

编译器会把 HQL 编译成一个逻辑计划（Logical Plan）。也就是解析 SQL，SELECT 里的字段需要通过 map 操作获取，也就是需要扫描表的数据。Group By 的语句需要通过 reduce 来做分组化简，而 Join 则需要两个前面操作的结果的合并。当然，在逻辑计划里

还没有 MapReduce 的概念，而是一个编译器里实际的抽象语法树（AST），树的每一个节点都是一个操作符。

然后是一个优化器。

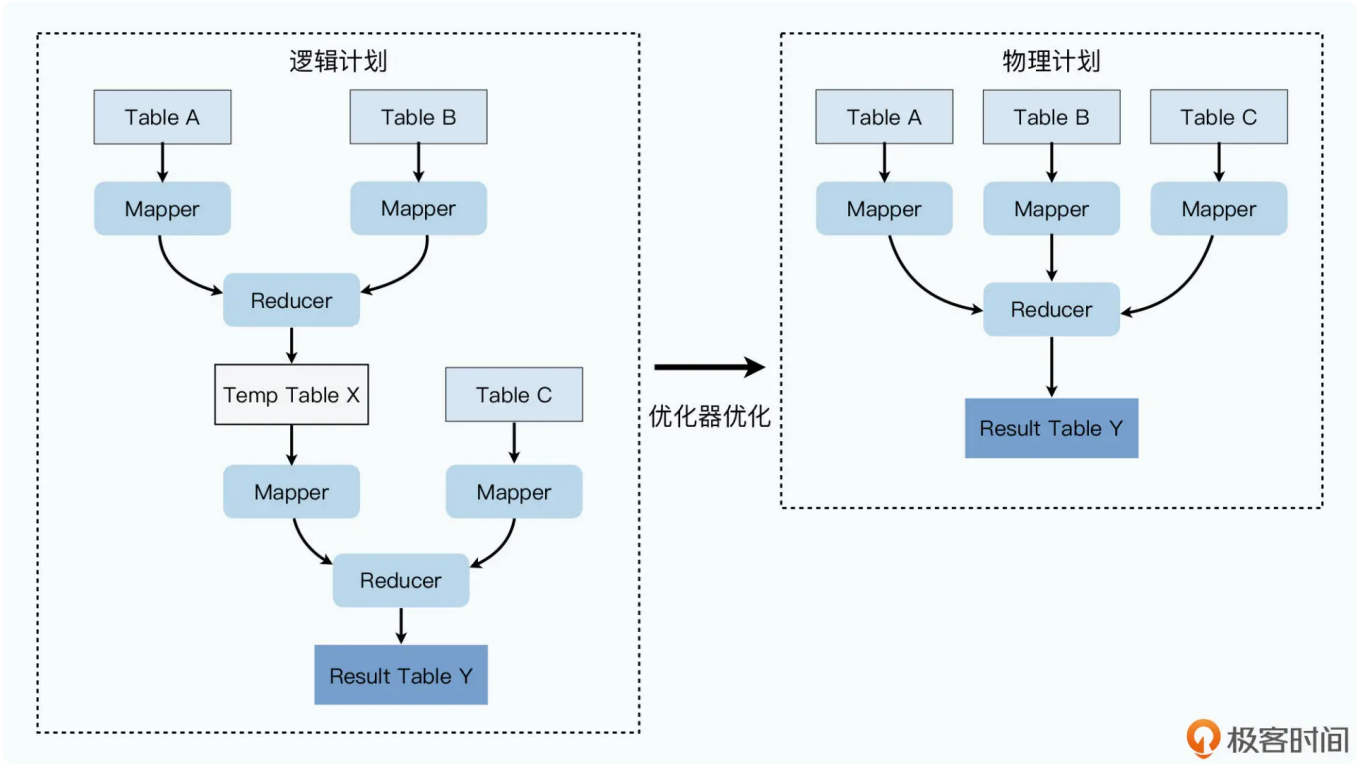
优化器会拿到逻辑计划，然后根据 MapReduce 任务的特点进行优化，变成一个物理计划（Physical Plan）。

一个典型的例子是，我们有三张数据表 A、B、C，分别存放了用户的基础信息、用户的标签，以及用户的支付数据，然后通过每张表里的 user_id 把这三张表 Join 在一起。如果按照逻辑计划直接一个个操作执行，那么我们会先 A Join B，把结果输出出来变成一个中间结果 X，然后再通过 X Join C 拿到最后的结果 Y。

这个过程，每个 Join 都是一个 MapReduce 的任务，但是当优化器看到两个 Join 的 key 是相同的，就可以通过一个 MapReduce，在 Map 端同时读入三个表，然后通过 user_id 进行 Shuffle，并在 Reduce 里进行对应的数据提取处理。这样，我们就可以“优化”到只有一个 MapReduce 就能完成整个过程。

复制代码

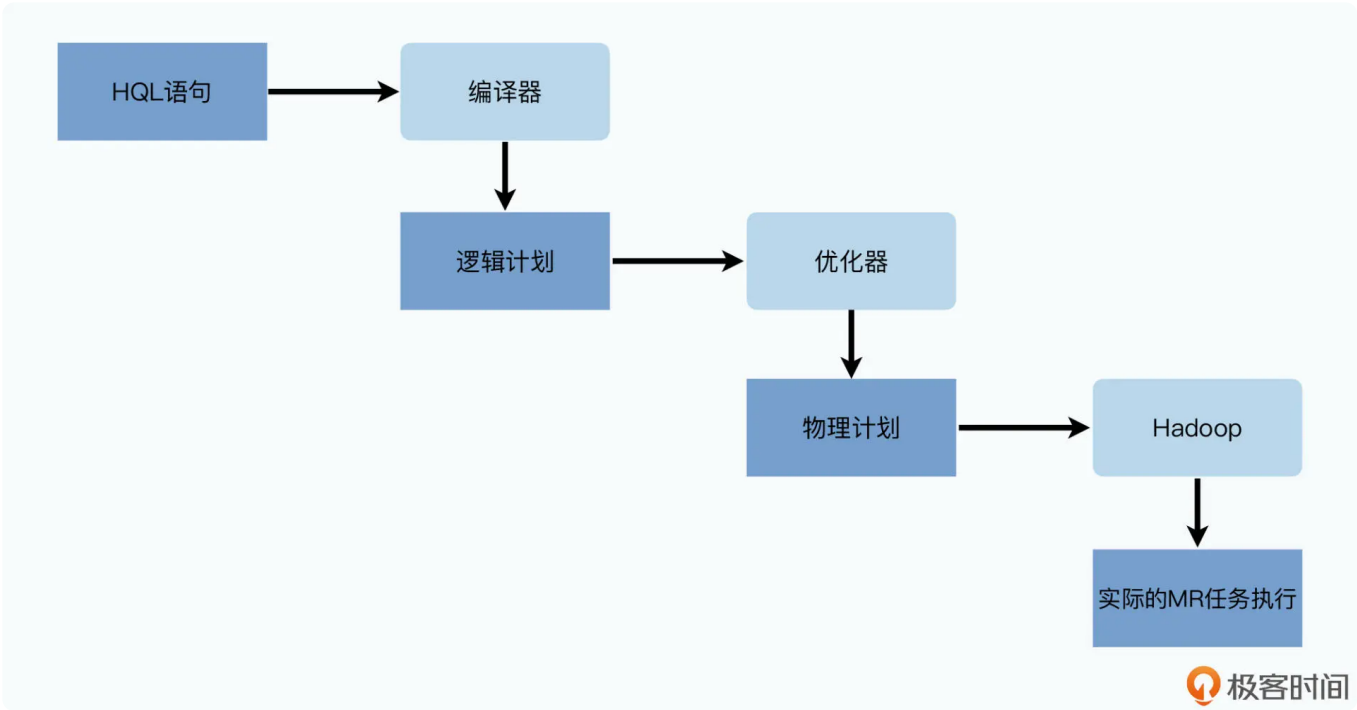
```
SELECT A.user_id, MAX(B.user_interests_score), SUM(C.user_payments)
FROM A
LEFT JOIN B on A.user_id = B.user_id
LEFT JOIN C on A.user_id = C.user_id
GROUP BY A.user_id
```



SQL需要JOIN三张表，逻辑计划会拆分成的操作步骤如果直接执行，会变成多个MapReduce的过程
但是优化器因为它们Join的key相同，会把它们合并成一个

最后是一个执行引擎和一个有向无环图。

最后的物理计划，会通过一个执行引擎以及一个有向无环图（DAG），来按照顺序进行执行。执行引擎在这里就是 Hadoop 的 MapReduce，未来，Hive 还会扩展到使用 Spark 等其他的计算引擎里。



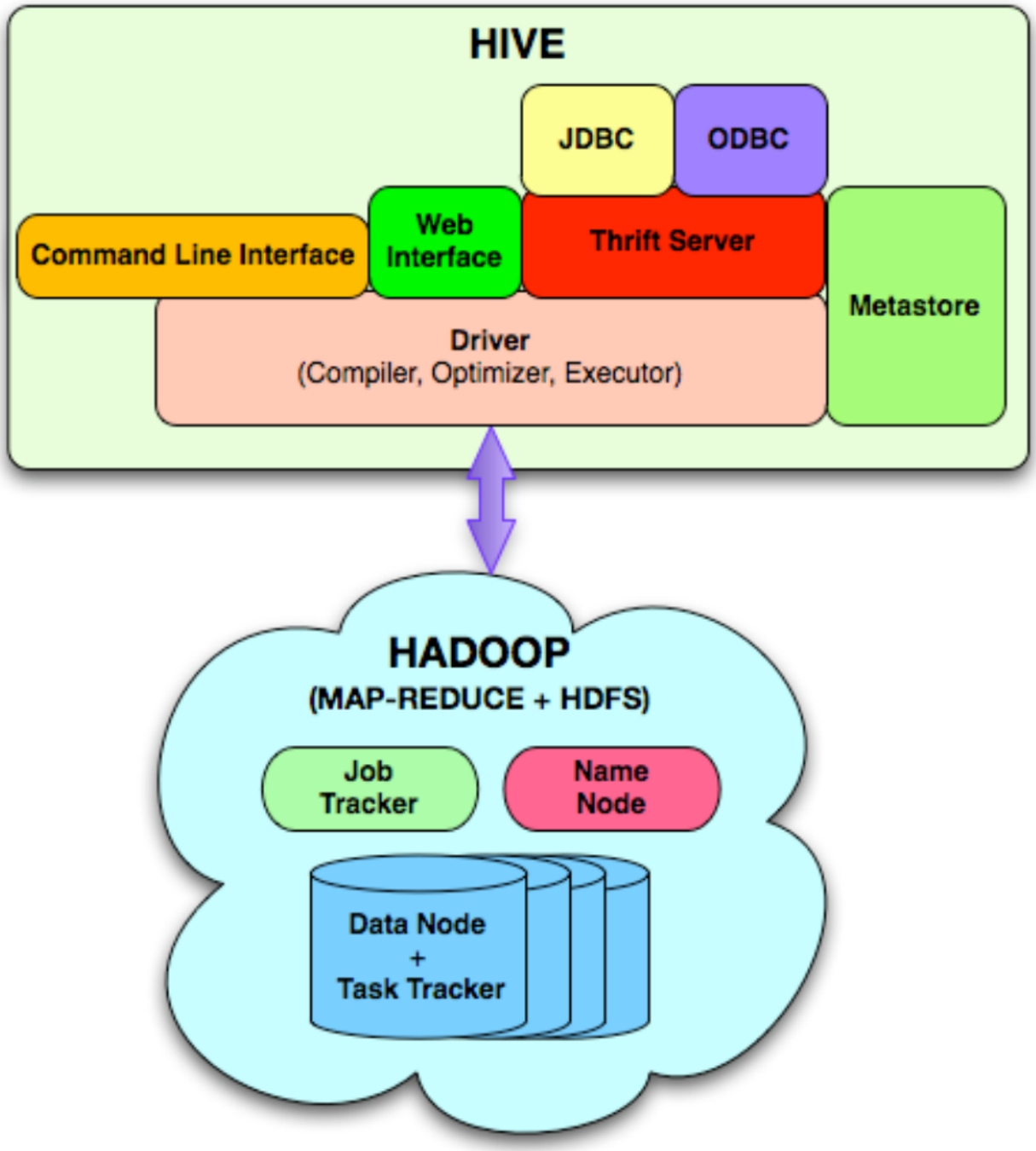
第三部分：Metastore

在接口和驱动器之外，Hive 最后一个模块则是 Metastore，用来存储 Hive 里的各种元数据。无论是表的名称、位置、列的名称、类型都会存放在这里。这个 Metastore，我们通常是使用中心化的关系数据库来进行存储的。

不要小看 Metastore，**Hive 的所有数据表的位置、结构、分区等信息都在 Metastore 里**。如果没有这个 Metastore，Hive 就更像 Pig 这样，只是一个方便运用的 DSL。

而正是这个 Metastore，让 Hive 成了一个完整的数据仓库解决方案。我们的驱动器里的各个模块，也需要通过 Metastore 里面的数据，来确定解析 HQL 的时候，是否所有字段都存在并且合法，以及确认最终执行计划的时候，从哪些目录读取数据。

到这里，我们可以看到，Hive 在系统架构上并不复杂，核心模块其实就是一个 HQL 编译器 + 优化器。其余无论是数据存储还是分布式计算，仍然是依托于 Hadoop，也就是 GFS+MapReduce 这两篇论文里的底层逻辑。不过，这也是大数据系统迭代中我们会看到的规律。**在现实的工程界，很少有石破天惊、完全重新设计开发的新系统，而更多的是尽可能复用现有的成熟系统和成熟模块，在上面迭代、改进以及创新。**



论文里的Hive系统架构图，其实也非常简单清晰

其实，把 HQL 编译成 MapReduce 来执行的创意并不新鲜，早在 2006 年，Google 就已经发表了 Sawzall 的论文，而 Apache Pig 也在 2008 年 9 月 11 日就首次发布了。它们都是通过一门 DSL 来编译成 MapReduce 任务加以执行，从而完成数据分析工作。我在下面放了一小段示例代码，你可以看到，其实 Pig 这样的 DSL 也很容易进行数据的统计和分析。

复制代码

```
1 A = LOAD '/data/logs/20090101/*' as (logid:string, uid:string, url:string);
2 B = GROUP A BY url;
```

```
C = FOREACH B GENERATE url, COUNT(*);
DUMP C;
```

> 通过简单的 GROUP BY 关键字，Pig 一样可以快速进行分组的数据统计
> 但是，相比于 Hive，Pig 缺了几个很重要的环节。

首先是没有选择 SQL 作为 DSL。SQL 其实是在大数据出现之前，大部分数据分析师日常使用的“标准”语言。对应的，市场上也围绕 SQL 有大量的 BI 类型的产品，进行各种可视化分析。

也因此，Facebook 不仅为 Hive 提供了 HQL 的命令行界面和 WebUI，还通过 Thrift Server 提供了 JDBC 和 ODBC 的接口。这些标准接口使得传统的 BI 数据分析工具，可以很容易地接入到 Hive 中来。而 Facebook 在持续开发 Hive 的过程中，一个很重要的关注点就是如何努力让 Hive 支持更多的 SQL 语义。

其次是没有提供一个足够好的“UI”。这个 UI 不只是一个 Web 界面，让你可以输入运行 SQL，也包括我们前面说的 JDBC/ODBC 这样的驱动层。更重要的是，Hive 通过 Metastore 这个元数据存储模块，统一管理了所有数据表的位置、结构、字段名称，以及 SerDer 的信息。

而当你去使用 Pig 的时候，这些都需要自己在 Pig 脚本里写。当分析师想要去看一看现在有哪些数据表，以及每个数据表是什么格式的时候，Hive 就比 Pig 要方便许多。

当我们只有一两张数据表的时候，这个优势还不明显。而当我们像 Facebook 一样，有数百 TB 的数据、数不清的数据表的时候，这个 Metastore 变成一个不可或缺的部分了。

我自己在很长一段时间里也是 Pig 的重度用户，我觉得对于开发人员来说，Pig 更加灵活。但是历史证明，遵循一个主流标准，在使用上提供更好的 UI，才能让一个产品更加发展壮大。即使大数据技术已经迭代过好几轮了，Hive 仍然有着顽强的生命力，而 Pig 则已经慢慢消逝在历史的背影之中了。

小结

不知道你有没有发现，这一讲的讲解顺序，其实类似于我之前在加餐中讲过的结构化阅读的方法。我们先是看了摘要和概述，理解了 Hive 是为了解决商业数据库的性能不足，以及

由于 MapReduce 撰写数据分析程序太麻烦而诞生的。整个论文的结构也可以拆分成数据模型（Section III）、数据如何存储（Section IV）和 Hive 系统的整体架构（Section V）这三个部分。

数据模型里，Hive 通过引入明确的类型系统和 SQL-like 的 HQL，使得普通的熟悉 SQL 的分析师也可以快速分析海量数据。而整个系统的实现，则很好地复用了 Hadoop 的基础架构。

数据存储里，Hive 的数据是直接放在 HDFS 上，通过 HDFS 上不同的目录和文件来做到分区和分桶。分区和分桶，使得 Hive 下很多的 MapReduce 任务可以减少需要扫描的数据，提升了整个系统的性能。

整体架构上，Hive 其实是三个核心组件的组合，第一个是提供了 HQL 到 MapReduce 任务转换的 Driver，它提供了编译器、优化器以及执行器；第二个是存储了表结构、分区、存储位置等这些元数据的 Metastore；第三个，则是一系列对终端用户提供的 UI 组件，包括命令行、WebUI、JDBC/ODBC 驱动以及下面的 Thrift Server。

可以说，Hive 并没有自己去实现任何分布式计算和分布式存储的底层工作，实际的所有计算和存储仍然是基于 HDFS 和 MapReduce。完全复用 HDFS 和 MapReduce，使得实现 Hive 不再是一个那么困难的工作了，而这个也是我们在系统设计中常见的一个进化过程。

并且，因为 Hive 只是作为一个底层计算引擎的“驱动”（Driver）出现，使得它可以去适配其他的计算引擎。这也是为什么，今天我们已经很少使用原始的 MapReduce，而 Hive 还能够通过 Hive on Spark 这样使用更快的执行引擎的方式，来保持旺盛的生命力。

而通过实现 SQL 标准来实现 OLAP 的数据分析操作，就让 Hive 打败了 Sawzall 和 Apache Pig 这样的 DSL 系统，也让我们体会到了“标准”和用户群对于一个产品、一个项目的生命力的重要作用。

Hive 这样通过 SQL 形式的语法，去运行 MapReduce 的想法并不难想到，但这也是整个大数据领域里至关重要的一步。通过 HQL，大数据不再是工程师的专利，而是能够惠及到所有的数据分析人员了。

推荐阅读

这一讲里，我并没有深入讲解 HQL 的编译和优化过程。想要深入理解这一部分，需要比较多的编译原理的基础知识，我建议你可以先学习一些编译原理的课程。

之后，如果你想对 HQL 具体是怎么编译成 MapReduce 的执行计划，除了去阅读 Hive 的源码，还可以去看一看美团技术团队写的《Hive SQL 的编译过程》这篇文章。

另外，你也可以去读一读 Google 在 2006 年发表的 Sawzall 这个 DSL 的论文，这也是思考如何通过 DSL 来替代撰写 MapReduce 代码的第一篇文章。

思考题

最后给你留一道思考题，Hive 可以通过目录结构来进行分区和分桶，使得最终执行的 MapReduce 需要扫描的数据变得很少。那么，如果我们把很多字段都去分区和分桶，是不是可以大大提升 Hive 的性能呢？我们如果这样操作，会有什么负面影响吗？

上一篇 复习课（四）| Thrift

下一篇 16 | 从Dremel到Parquet（一）：深入剖析列式存储

精选留言 (3)

写留言

峰
2021-10-25

最明显的问题就是小文件过多，不按照分区查必然需要open巨量文件，而且对元数据也是个负担。虽然分区分桶都是为了快速定位定位到要操作的文件集合，但却物理分割了数据，而不像索引依然保证数据是连续的。

展开

1

在路上
2021-10-26

徐老师好，从存储的角度上讲，分区和分桶都是在增加文件数量，合理使用分区和分桶，可以让文件不至于太大，减少查询时扫描的数据量，如果过度使用，会增加很多小文件，而HDFS并不适合存储小文件。从查询的角度讲，当我们按a、b、c、d四个字段分区时，会形成/a=*/b=*/c=*/d=*的四级目录，只有a、b、c、d同时作为查询条件时，才能利用这种存储方式的好处，否则需要检索很多文件，不如把它们放在同一个文件。只有经常使...

展开

1

xunix
2021-10-26

在微软内部有一套大数据处理语言叫Scope 可以看做是兼容了SQL的Pig（对外的版本叫U-S

QL 可以在Azure上体验）少数文件的话感觉用起来还是挺顺手的
但随着数据量增加 我们使用的时候逐步加入了各种存储路径的管理 现在看起来也是和met
astore差不多的形式

展开