Presto 在有赞的实践之路

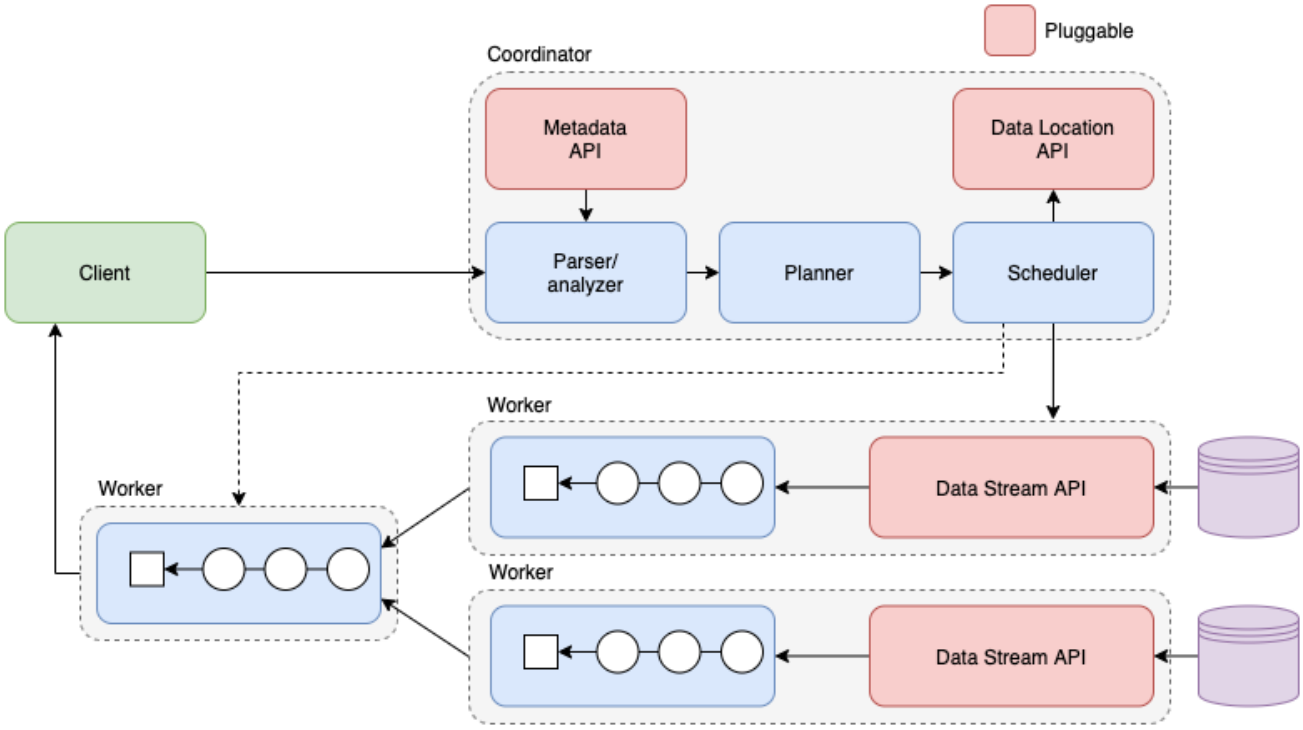
一、前言

本文主要介绍了 Presto 的简单原理，以及 Presto 在有赞的实践之路。

二、Presto 介绍

Presto 是由 Facebook 开发的开源大数据分布式高性能 SQL 查询引擎。起初，Facebook 使用 Hive 来进行交互式查询分析，但 Hive 是基于 MapReduce
为批处理而设计的，延时很高，满足不了用户对于交互式查询想要快速出结果的场景。为了解决 Hive 并不擅长的交互式查询领域，Facebook 开发了
Presto，专门为交互式查询所设计，提供分钟级乃至亚秒级低延时的查询性能。

2.1 Presto 架构



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

2.2 Presto 执行查询过程

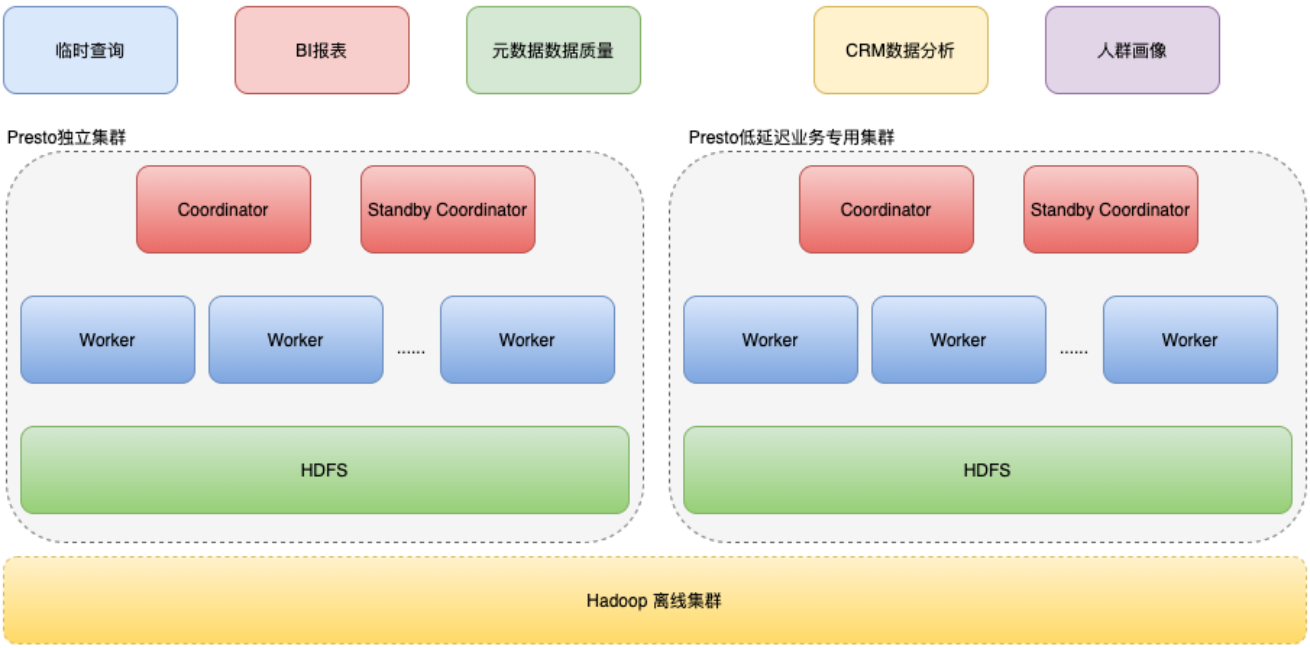
- Client 发送请求给 Coordinator。
- SQL 通过 ANTLR 进行解析生成 AST。

- AST 通过元数据进行语义解析。
- 语义解析后的数据生成逻辑执行计划，并且通过规则进行优化。
- 切分逻辑执行计划为不同 Stage，并调度 Worker 节点去生成 Task。
- Task 生成相应物理执行计划。
- 调度完后根据调度结果 Coordinator 将 Stage 串联起来。
- Worker 执行相应的物理执行计划。
- Client 不断地向 Coordinator 拉取查询结果，Coordinator 从最终汇聚输出的 Worker 节点拉取查询结果。

2.3 Presto 为何高性能

- Pipeline, 全内存计算。
- SQL 查询计划规则优化。
- 动态代码生成技术。
- 数据调度本地化，注重内存开销效率，优化数据结构，Cache，非精确查询等其它技术。

三、Presto 在有赞的使用场景



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

- 数据平台(DP)的临时查询: 有赞的大数据团队使用临时查询进行探索性的数据分析的统一入口，同时也提供了脱敏，审计等功能。
- BI 报表引擎：为商家提供了各类分析型的报表。
- 元数据数据质量校验等：元数据系统会使用 Presto 进行数据质量校验。
- 数据产品：比如 CRM 数据分析，人群画像等会使用 Presto 进行计算。

四、Presto 在有赞的演进之路

第一阶段: Presto 和 Hadoop 混合部署阶段:
起初，Presto 是和 Hadoop 离线集群混合在一起部署的。但是那时候用户经常会抱怨 Presto 执行性能不稳定，对于同样的 SQL，时快时慢。我们观察到同样的
Task，处理的数据量和花费的CPU Time 类似，但是有时候就会出现某些特别长的Elapsed Time的Task，从而拖慢整体的查询性能。经分析，是因为在这个时间点磁盘 IO 带宽被 Hadoop 离线任务打满导致的。虽然 Hadoop 离线集群一般任务都会在凌晨进行调度，但是
也有一些任务会在白天不定期地跑，这种时候往往会比较影响性能。于是我们决定完全独立 Presto 集群，并且单独安装 HDFS 环境。

第二阶段: Presto 集群完全独立阶段:
我们准备将 Presto 单独规划出一个集群，并且单独安装 HDFS 环境，而离线 Hadoop 集群只需要将数据每天导入到这个 HDFS 环境中，此后离线 Hadoop 集群所有的任务都不会影响 Presto 集群。 第一个问题就遇到了如何去将现有离线 Hadoop 集群的数据表导入到新的集群。目前我们的方案是共同使用一个
Hive，通过为专门新建一个库，在创建库的时候指定Location的方式去关联到 Presto 集群的 HDFS NameService。后面用户在这个库下面建表就会将 Hive 表存储到 Presto 集群。
这时候我们的 Presto
性能就会相对稳定得多，基本不再会同样的task处理差不多数据量的时候有几个 Elapsed Time 特别高的情况了。

第三阶段: 低延时业务专用 Presto 集群阶段:
在第二阶段我们的业务之间的资源隔离主要还是靠 Resource Group，但是这种隔离方式相对比较弱，不能提供细粒度的隔离，任务之间还是会互相影响。此外，不同业务的sql类型，查询数据量，查询时间，可容忍的 SLA，可提供的最优配置都是不一样的。有些业务方需要一个特别低的响应时间保证，于是我们给这类业务部署了专门的集群去处理。 部署在这个集群上的业务要求低延时，通常是3秒内，甚至有些能够达到1秒内，而且会有一定量的并发。不过这类业务通常数据量不是非常大，而且通常都是大宽表，也就不需要再去 Join 别的数据，Group By 形成的 Group 基数和产生的聚合数据量不是特别大，查询时间主要消耗在数据扫描读取时间上。我们同样也提供了资源完全独立，具有本地 HDFS 的专用 Presto 集群给这类业务方去使用。此外，我们会为这种业务提供深度的性能测试，调整相应的配置，比如将 Task Concurrency
改成1，在并发量高的测试场景中，反而由于减少了线程间切换，性能会更好。

五、Presto 在有赞使用中的遇到的问题

5.1 HDFS 小文件问题

HDFS 小文件问题在大数据领域是个常见的问题。我们发现我们的数仓 Hive
表有些表的文件有几千个，查询特别慢。Presto 这两个参数限制了 Presto 每个节点每个 Task 可执行的最大 Split 数目。

node-scheduler.max-splits-per-node=100
node-scheduler.max-pending-splits-per-task=10

因此当查询有许多小文件的表的时候，问题就爆发出来了，查询起来特别慢。为了解决这个问题，我们分两步走:

- 适当调大了这两个参数;
- 在 Spark，Hive ETL 层面引入 Adaptive Spark 和小文件合并工具去解决这个小文件问题。

5.2 正则表达式指数级别回溯问题

有一天，有个用户一个临时查询跑了1个小时也没退出，通过
jstack，找到了对应代码，发现是在运行 Presto 里面的正则表达式引擎 Joni
库匹配的代码，后来发现他写的是一个会产生指数级别回溯的正则表达式。社区的反馈是可以将 Presto 的正则表达式配置成 Google RE2J，但是 RE2J 会牺牲掉一些正则表达式的语法。尽管 Presto 是个多线程执行引擎，但是 Joni 引擎在设计上还是可以被 Interrupt
的(如果是java原生的正则表达式库是无法被 Interrupt 的)，于是加上了查询最大运行时间的限制，并且通知了相关的用户。详见(https://github.com/prestodb/presto/issues/12191)

5.3 多个列 Distinct 的问题

有一些报表业务是使用 Presto 直接来算转化率的，这样的报表就会引起一个查询语句中有多个 count distinct 列的问题。 然而查看性能的时候会发觉这种语句特别慢，后来发觉，就算我手动将这个查询语句分成多个语句，每个语句去执行一个 count distinct 时，也比合起来要快。
于是深入调研了下，Spark，Hive TEZ，Calcite 之类的发觉 count distinct 在 SQL
优化器那边会被优化掉，来解决数据倾斜的问题。

简单来说: 单列的 count distinct:

select A, count(distinct B) from T group by A.

转换成

select A, count(B) from (select A, B from T group by A, B) group by A.

而多个 count distinct 列的原理类似，就是会使用 grouping sets 去将多个 group by
整合到一起来提升

SELECT a1, a2,..., an, F1(b1), F2(b2), F3(b3), ..., Fm(bm), F1(distinct c1), ..., Fm(distinct cm) FRO M Table GROUP BY a1, a2, ..., an

转换为

```
SELECT a1, a2,..., an, arbitrary(if(group = 0, f1)),...., arbitrary(if(group = 0, fm)), F(if(group = 1, c1
)), ...., F(if(group = m, cm)) FROM
    SELECT a1, a2,..., an, F1(b1) as f1, F2(b2) as f2,...., Fm(bm) as fm, c1,..., cm group FROM
      SELECT a1, a2,..., an, b1, b2, ... ,bn, c1,..., cm FROM Table GROUP BY GROUPING SETS ((a1,
a2,..., an, b1, b2, ... ,bn), (a1, a2,..., an, c1), ..., ((a1, a2,..., an, cm)))
    GROUP BY a1, a2,..., an, c1,..., cm group
GROUP BY a1, a2,..., an
```

Presto 对于多个 count distinct 列这方面并没有去实现。

这边我们目前采用的方案是:

- 修改代码去实现，并且提交了 Issue 和 PR 给社区，一个被 merge 了，还有一个还在 review 中，后续还会继续跟进。
    1. Issue: [Optimize distinct aggregation on multi column (https://github.com/prestosql/presto/issues/613)
    2. PR1: Fix Count(*) on empty relation returns NULL when optimizemixeddistinct_aggregation is turned on Merged
    3. PR2: [Optimize distinct aggregation on multiple columns (https://github.com/prestosql/presto/pull/624) Reviewing
- 让业务方可以容忍非精确去重的选用 approxmate_distinct 去实现。

5.4 HDFS Namenode 导致有少数查询会相对慢一点

在我们给用户做专用presto集群独立的性能测试时，我们发现同样的SQL会有很少数查询慢一点，后来研究了下发现 Presto Coordinator 去通过

```
public RemoteIterator<locatedfilestatus> listLocatedStatus(final Path f)
```

调用请求 HDFS NameNode 的时候，有时候会延迟1秒后返回。后来发觉这个时候正好是 NameNode 在做 Edit Log Rolling 的时候，由于这个时候 NameNode 会去拿读写锁的写锁，从而阻塞了读请求获得读锁，因此有时候延迟1秒后返回。

这个问题目前由于基本可容忍，现阶段也满足了业务方的 SLA，所以后面没有去解决:
我个人觉得，HDFS 并不是为了在线服务设计的，要提高 HDFS RPC 请求的稳定性，有以下几种方式:

- 参考[Uber 引入了 Observer NameNode]。(https://eng.uber.com/scaling-hdfs/)
- 使用 Alluxio，我们简单测试了下 Alluxio，Alluxio Master 好像不会出现这种问题，在后面对未来的展望小节中，我们提到了 Alluxio + Presto

的意义。
- 尝试更换 NameNode 的盘为 SSD 盘，减少 Edit Log Rolling 的时间。

六、对未来的展望

6.1 Presto + Alluxio

Alluxio 通过能够细粒度的去控制内存，会比纯粹的靠 OS Page Cache
去控制页级别缓存更具有优势。你可以将一个表加载到 Alluxio 里面，然后每次对它的访问 IO
这块花费的时间基本可以说是快速且恒定的。当然，我们也需要理性看待
Alluxio，从原理本质上来讲，就 Presto 读取数据这块，这个要视情况而论.
我们测试过 Presto 提供的 HiveFileFormatBenchmark，大家也可以自己跑一下，结论就是单个 CPU 核读取 TPCH 的一个 Lineitem 表，ORC ZLIB 压缩方式大概是在40MB/s, 当然不同数据格式，不同压缩比会有所不同。因此，现代磁盘顺序读写的速度可以达到150MB/s，如果就一个任务是不会有瓶颈的。这时候 CPU 是瓶颈， 但是现实是一个查询多个任务跑，多个查询并行跑，你这个时候就很难保证磁盘顺序读写，吞吐，以及是否在 OS Page Cache
中，这个时候就很有可能磁盘 IO 是瓶颈了。因此 Alluxio 还是有用武之地的，至少可以把磁盘 IO 这个不可控因素给恒定下来。

6.2 Presto session property managers

新版本的 Presto 实现了 Session property manager 对于不同的 WorkLoad，不同的业务 SQL 类型，数据量，通过不同的配置能够达到最好的效果。这个靠用户自己去设置 Session Property 是不太现实的，必须在 Presto 服务端进行管理。

6.3 Presto多租户隔离

目前 Presto 官方并没有实现和 Apache Ranger 结合的多租户隔离机制，我们目前有一个 Sql Parser服务，去解析 Presto，Hive，Spark
三种引擎的语法，去做脱敏，审计，智能选择等功能，后面会去做结合 Apache Ranger 通过 sql 重写来实现数据隔离，类似于现在的脱敏实现。

本文原文：Presto 在有赞的实践之路