大数据时代快速SQL引擎-Impala

背景

随着大数据时代的到来，Hadoop在过去几年以接近统治性的方式包揽的ETL和数据分析查询的工作，大家也无意间的想往大数据方向靠拢，即使每天数据也就几十、几百M也要放到Hadoop上作分析，只会适得其反，但是当面对真正的Big Data的时候，Hadoop就会暴露出它对于数据分析查询支持的弱点。甚至出现《MapReduce: 一个巨大的倒退》此类极端的吐槽，这也怪不得Hadoop，毕竟它的设计就是为了批处理，使用用MR的编程模型来实现SQL查询，性能肯定不如意。所以通常我也只是把Hive当做能够提供将SQL语义转换成MR任务的工具，尤其在做ETL的时候。

在Dremel论文发表之后，开源社区涌现出了一批基于MPP架构的SQL-on-Hadoop(HDFS)查询引擎，典型代表有Apache Impala、Presto、Apache Drill、Apache HAWQ等，看上去这些查询引擎提供的功能和实现方式也都大同小异，本文将基于Impala的使用和实现介绍日益发展的基于HDFS的MPP数据查询引擎。

Impala介绍

Apache Impala是由Cloudera开发并开源的一款基于HDFS/Hbase的MPP SQL引擎，它拥有和Hadoop一样的可扩展性、它提供了类SQL（类Hsql）语法，在多用户场景下也能拥有较高的响应速度和吞吐量。它是由Java和C++实现的，Java提供的查询交互的接口和实现，C++实现了查询引擎部分，除此之外，Impala还能够共享Hive Metastore（这逐渐变成一种标准），甚至可以直接使用Hive的JDBC jar和beeline等直接对Impala进行查询、支持丰富的数据存储格式（Parquet、Avro等），当然除了有比较明确的理由，Parquet总是使用Impala的第一选择。

从用户视角

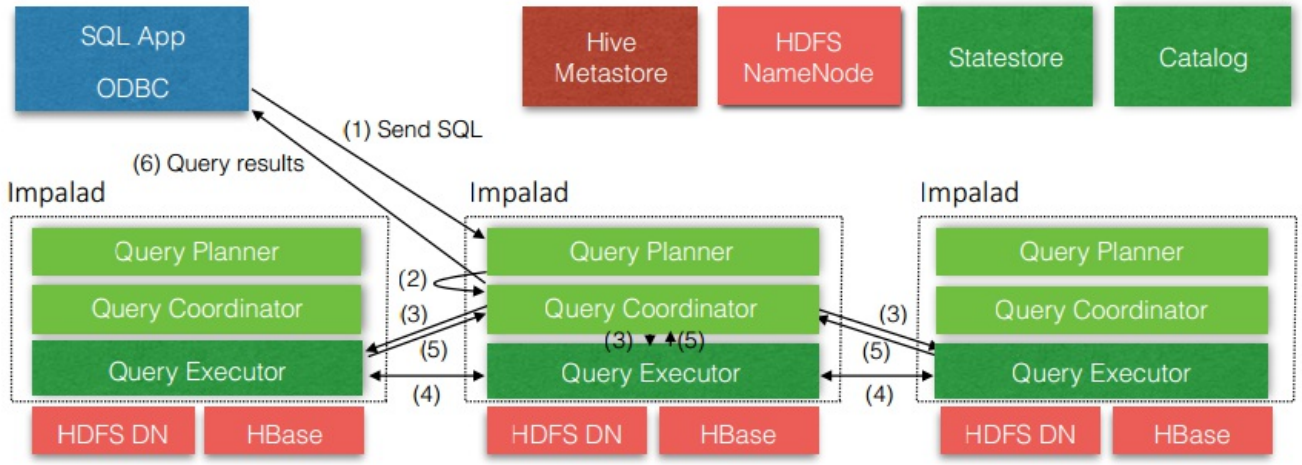
可以将Impala这类系统的用户分为两类，一类是负责数据导入和管理的数据开发同学，另一类则是执行查询的数据分析师同学，前者通常需要将数据存储到HDFS，通过CREATE TABLE的方式创建与数据match的schema，然后通过load data或者add partition的方式将表和数据关联起来，这一些流程串起来还是挺麻烦的，但是多亏了Hive，由于Impala可以共享Hive的MetaStore，这样就可以使用Hive完成此类ETL工作，然后将数据查询的工作交给Impala，大大简化工作流程（据我所知毕竟大部分数据开发同学还是比较熟悉Hive）。接下来对于数据分析师而言就是如何编写正确的SQ以表达他们的查询、分析需求，这也是它们最拿手的了，Impala通常可以在TB级别的数据上提供秒级的查询速度，所以使用起来可能让你从Hive的龟速响应一下提升到期望的速度。

Impala除了支持简单类型之外，还支持String、timestamp、decimal等多种类型，用户还可以对于特殊的逻辑实现自定义函数（UDF）和自定义聚合函数（UDAF），前者可以使用Java和C++实现，后者目前仅支持C++实现，除此之外的schema操作都可以在Hive上实现，由于Impala的存储由HDFS实现，因此不能够实现update、delete语句，如果有此类需求，还是需要重新计算整个分区的数据并且覆盖老数据，这点对于修改的实时性要求比较高的需求还是不能满足的，如果有此类需求还是期待Kudu的支持吧，或者尝试一下传统的MPP数据库，例如GreenPlum。

当完成数据导入之后，用户需要执行COMPUTE STATS <table>以收集和更新表的统计信息，这些统计信息对于CBO优化器提供数据支持，用于生成更优的物理执行计划。测试发现这个操作的速度还是比较快的，可以将其看做数据导入的一部分，另外需要注意的是这个语句不会自动执行，因此建议用户在load完数据之后手动的执行一次该命令。

系统架构

从用户的使用方式上来看，Impala和Hive还是很相似的，并且可以共享一份元数据，这也大大简化了接入流程，下面我们从实现的角度来看一下Impala是如何工作的。下图展示了Impala的系统架构和查询的执行流程。



从上图可以看出，Impala自身包含三个模块：Impalad、Statestore和Catalog，除此之外它还依赖Hive Metastore和HDFS，其中Imapalad负责接受用户的查询请求，也意味着用户的可以将请求发送给任意一个Impalad进程，该进程在本次查询充当协调者（coordinator）的作用，生成执行计划并且分发到其它的Impalad进程执行，最终汇集结果返回给用户，并且对于当前Impalad和其它Impalad进程而言，他们同时也是本次查询的执行者，完成数据读取、物理算子的执行并将结果返回给协调者Impalad。这种无中心查询节点的设计能够最大程度的保证容错性并且很容易做负载均衡。正如图中展示的一样，通常每一个HDFS的DataNode上部署一个Impalad进程，由于HDFS存储数据通常是多副本的，所以这样的部署可以保证数据的本地性，查询尽可能的从本地磁盘读取数据而非网络，从这点可以推断出Impalad对于本地数据的读取应该是通过直接读本地文件的方式，而非调用HDFS的接口。为了实现查询分割的子任务可以做到尽可能的本地数据读取，Impalad需要从Metastore中获取表的数据存储路径，并且从NameNode中获取每一个文件的数据块分布。

Catalog服务提供了元数据的服务，它以单点的形式存在，它既可以从外部系统（例如HDFS NameNode和Hive Metastore）拉取元数据，也负责在Impala中执行的DDL语句提交到Metatstore，由于Impala没有update/delete操作，所以它不需要对HDFS做任何修改。之前我们介绍过有两种方式向Impala中导入数据（DDL）——通过hive或者impala，如果通过hive则改变的是Hive metastore的状态，此时需要通过在Impala中执行REFRESH以通知元数据的更新，而如果在impala中操作则Impalad会将该更新操作通知Catalog，后者通过广播的方式通知其它的Impalad进程。默认情况下Catalog是异步加载元数据的，因此查询可能需要等待元数据加载完成之后才能进行（第一次加载）。该服务的存在将元数据从Impalad进程中独立出来，可以简化Impalad的实现，降低Impalad之间的耦合。

除了Catalog服务，Impala还提供了StateStore服务完成两个工作：消息订阅服务和状态监测功能。Catalog中的元数据就是通过StateStore服务进行广播分发的，它实现了一个Pub-Sub服务，Impalad可以注册它们希望获得的事件类型，Statestore会周期性的发送两种类型的消息给Impalad进程，一种为该Impalad注册监听的事件的更新，基于版本的增量更新（只通知上次成功更新之后的变化）可以减小每次通信的消息大小；另一种消息为心跳信息，StateStore负责统计每一个Impalad进程的状态，Impalad可以据此了解其余Impalad进程的状态，用于判断分配查询任务到哪些节点。由于周期性的推送并且每一个节点的推送频率不一致可能会导致每一个Impalad进程获得的状态不一致，由于每一次查询只依赖于协调者Impalad进程获取的状态进行任务的分配，而不需要多个进程进行再次的协调，因此并不需要保证所有的Impalad状态是一致的。另外，StateStore进程是单点的，并且不会持久化任何数据到磁盘，如果服务挂掉，Impalad则依赖于上一次获得元数据状态进行任务分配，官方并没有提供可靠性部署的方案，通常可以使用DNS方式绑定多个服务以应对单个服务挂掉的情况。

Impalad模块

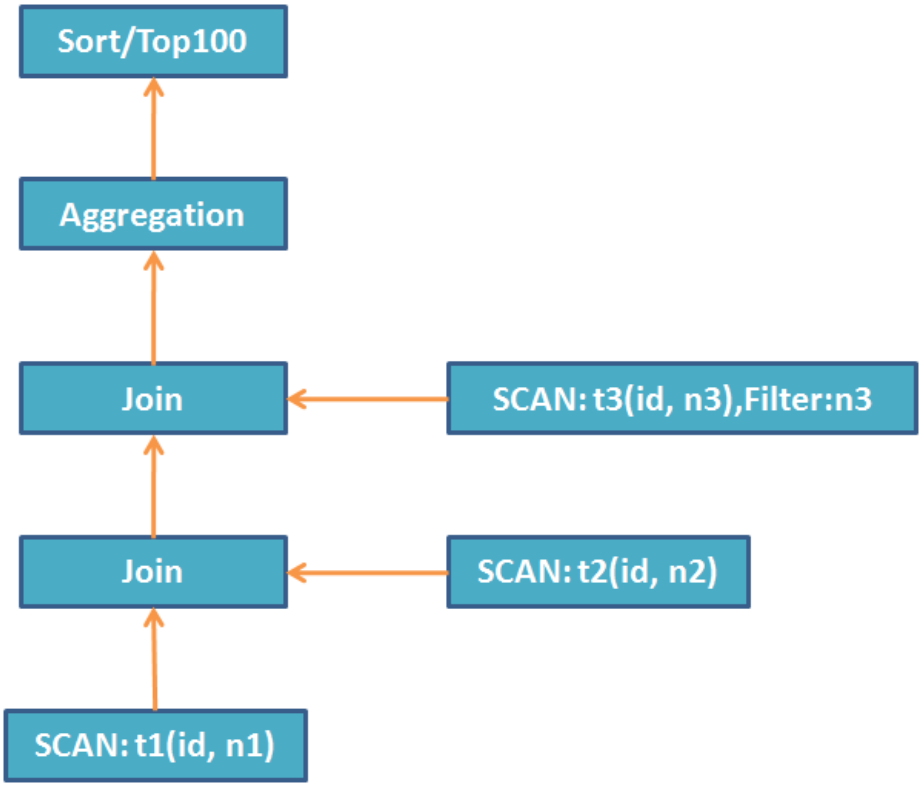
从Impalad的各个模块可以看出，主要查询处理都是在Impalad进程中完成，StateStore和Catalog帮助Impalad完成元数据的管理和负载监控等工作，其实更进一步可以将Query Planner和Query Coordinator模块从Impalad移出单独的作为一个入口服务存在，而Impalad仅负责数据读写和子任务的执行。

在Impalad进行执行优化的时候根本原则是尽可能的数据本地读取，减少网络通信，毕竟在不考虑内存缓存数据的情况下，从远端读取数据需要磁盘->内存->网卡->本地网卡->本地内存的过程，而从本地读取数据仅需要本地磁盘->本地内存的过程，可以看出，在相同的硬件结构下，读取其他节点数据始终本地磁盘的数据读取速度。

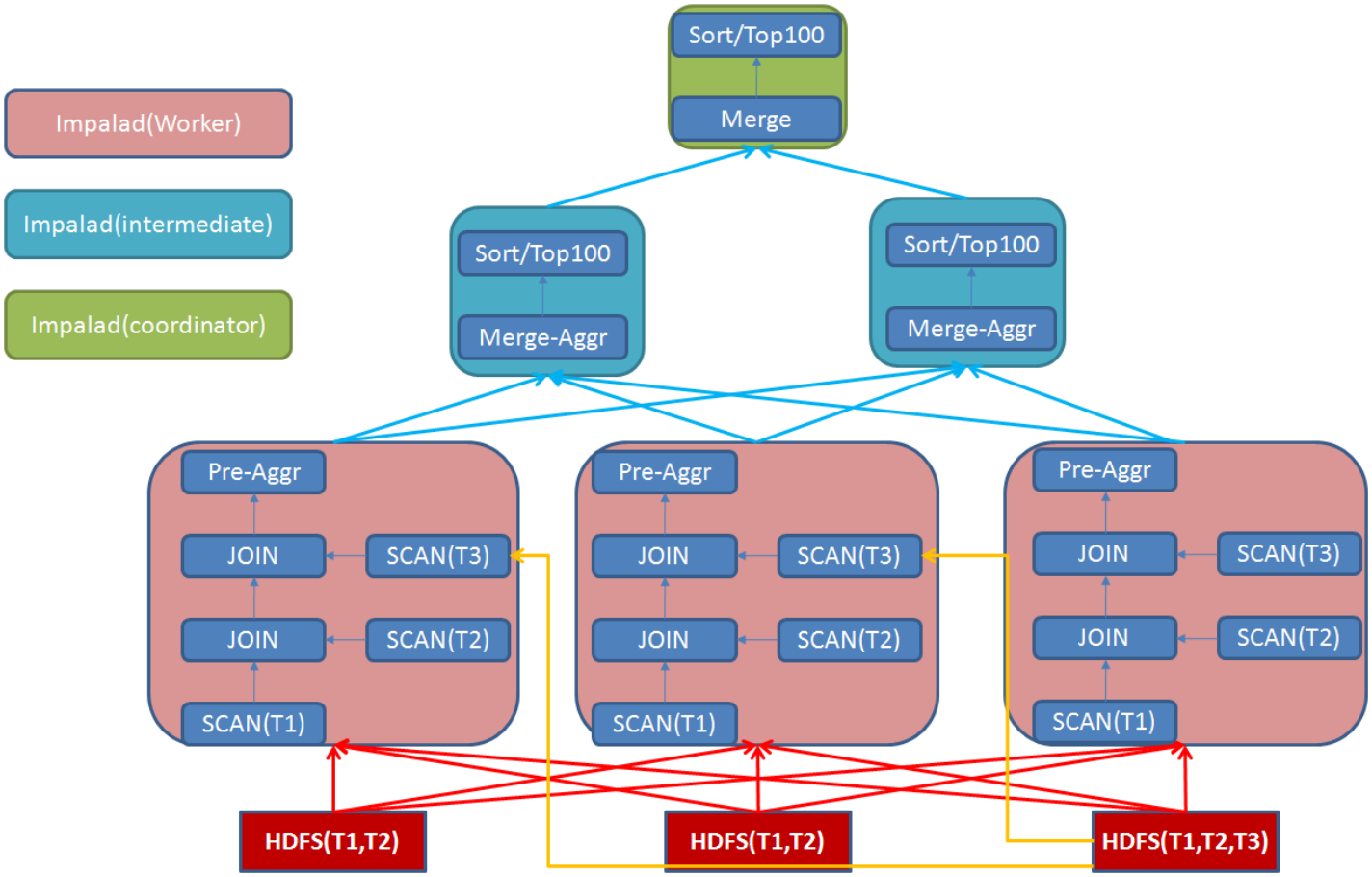
Impalad服务由三个模块组成：Query Planner、Query Coordinator和Query Executor，前两个模块组成前端，负责接收SQL查询请求，解析SQL并转换成执行计划，交由后端执行，语法方面它既支持基本的操作（select、project、join、group by、filter、order by、limit等），也支持关联子查询和非关联子查询，支持各种outer-join和窗口函数，这部分按照通用的解析流程分为查询解析->语法分析->查询优化，最终生成物理执行计划。对于Query Planner而言，它生成物理执行计划的过程分成两步，首先生成单节点执行计划，然后再根据它得到分区可并行的执行计划。前者是根据类似于RDBMS进行执行优化的过程，决定join顺序，对join执行谓词下推，根据关系运算公式进行一些转换等，这个执行计划的生成过程依赖于Impala表和分区的统计信息。第二步是根据上一步生成的单节点执行计划得到分布式执行计划，可参照Dremel的执行过程。在上一步已经决定了join的顺序，这一步需要决定join的策略：使用hash join还是broadcast join，前者一般针对两个大表，根据join键进行hash分区以使得相同的id散列到相同的节点上进行join，后者通过广播整个小表到所有节点，Impala选择的策略是依赖于网络通信的最小化。对于聚合操作，通常需要首先在每个节点上执行预聚合，然

后再根据聚合键的值进行hash将结果散列到多个节点再进行一次merge，最终在coordinator节点上进行最终的合并（只需要合并就可以了），当然对于非group by的聚合运算，则可以将每一个节点预聚合的结果交给一个节点进行merge。sort和top-n的运算和这个类似。

下图展示了执行select t1.n1, t2.n2, count(1) as c from t1 join t2 on t1.id = t2.id join t3 on t1.id = t3.id where t3.n3 between 'a' and 'f' group by t1.n1, t2.n2 order by c desc limit 100;查询的执行逻辑，首先Query Planner生成单机的物理执行计划，如下图所示：



和大多数数据库实现一样，第一步生成了一个单节点的执行计划，利用Parquet等列式存储，可以在SCAN操作的时候只读取需要的列，并且可以将谓词下推到SCAN中，大大降低数据读取。然后执行join、aggregation、sort和limit等操作，这样的执行计划需要再转换成分布式执行计划，如下图。



这类的查询执行流程类似于Dremel，首先根据三个表的大小权衡使用的join方式，这里T1和T2使用hash join，此时需要按照id的值分别将T1和T2分散到不同的Impalad进程，但是相同的id会散列到相同的Impalad进程，这样每一个join之后是全部数据的一部分。对于T3的join使用boardcast的方式，每一个节点都会收到T3的全部数据（只需要id列），在执行完join之后可以根据group by执行本地的预聚合，每一个节点的预聚合结果只是最终结果的一部分（不同的节点可能存在相同的group by的值），需要再进行一次全局的聚合，而全局的聚合同样需要并行，则根据聚合列进行hash分散到不同的节点执行merge运算（其实仍然是一次聚合运算），一般情况下为了较少数据的网络传输， intermediate节点同样也是worker节点。通过本次的聚合，相同的key只存在于一个节点，然后对于每一个节点进行排序和TopN计算，最终将每一个Worker的结果返回给coordinator进行合并、排序、limit计算，返回结果给用户。

Impalad优化

上面介绍了整个查询大致的执行流程，Impalad的后端使用的是C++实现的，这使得它可以针对硬件做一些特殊的优化，并且可以比使用JAVA实现的SQL引擎有更好的资源使用率。另外，后端的实现使用了LLVM，它是一个编译器框架，可以在执行器生成并编译代码。官方测试发现使用动态生成代码机制可以使得后端执行性能提高1—5倍。

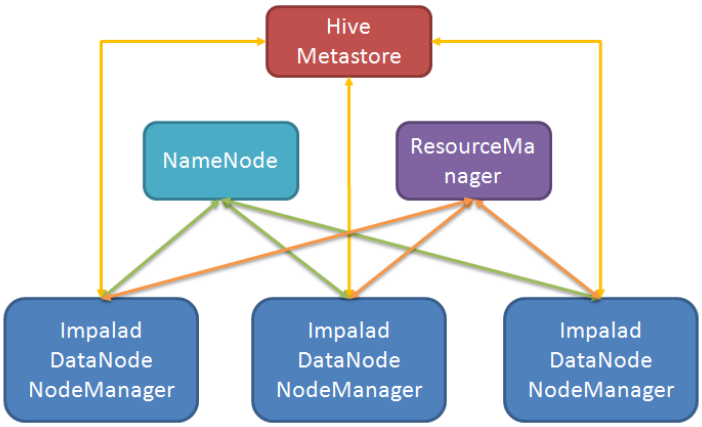
在数据访问方面，Impalad并没有使用通用的HDFS读取数据那一套流程，毕竟Impalad一般部署在DataNode上，访问数据完全不需要再走NameNode了，因此它使用了HDFS提供的Short-Circuit Local Reads机制，它提供了直接访问DataNode的方案，可以参考Hadoop官方文档和HDFS-347了解详情。

最后Impalad后端支持对中文件格式和压缩数据的读取，包括Avro、RC、Sequence、Parquet，支持snappy、gzip、bz2等压缩，看来Impala不支持可能也不打算支持ORC格式啦，毕竟有自家主推的Parquet，而ORC则在Presto中广泛使用。关于Parquet和ORC等列式存储格式可参考这里，这里，还有这里。

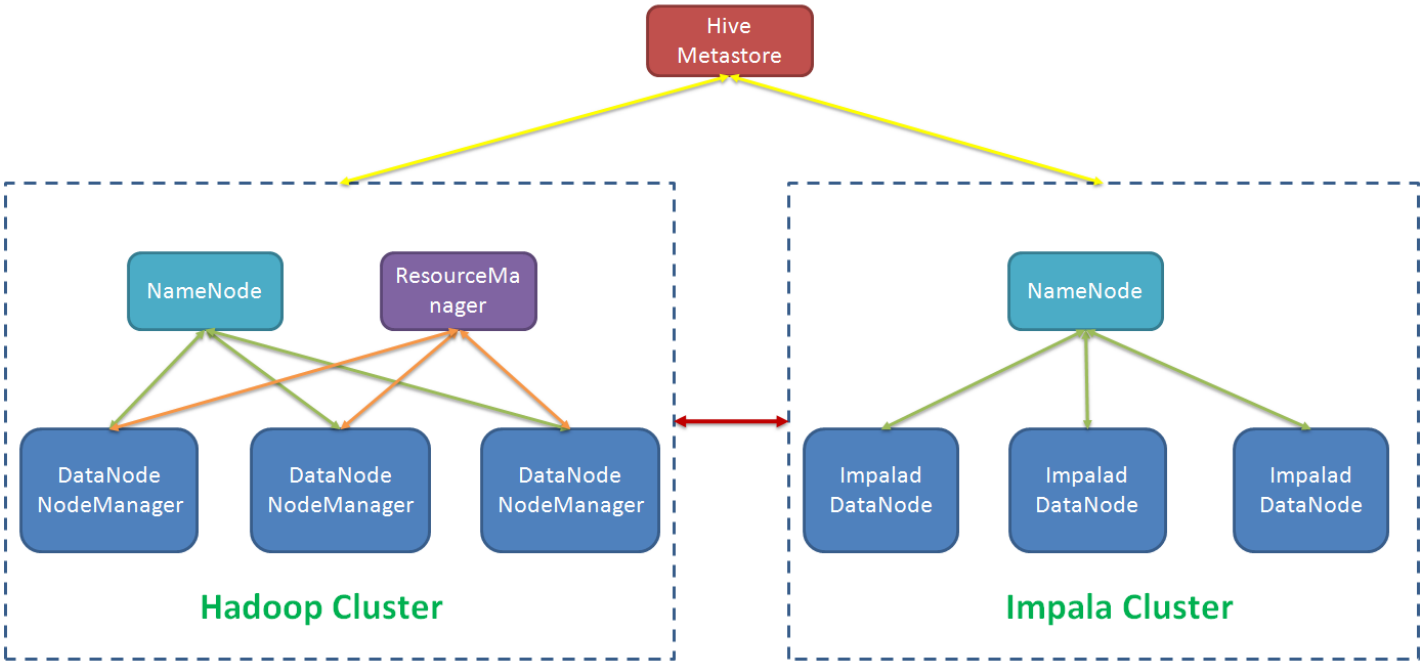
部署方式

通常情况下，我们会考虑两种方式的集群部署：混合部署和独立部署，下图分别展示了混合部署与独立部署时的各节点结构。混合部署意味着将Impala集群部署在Hadoop集群之上，共享整个Hadoop集群的资源；独立部署则是单独使用部分机器只部署HDFS和Impala，前者的优势是Impala可以和Hadoop集群共享数据，不需要进行数据的拷贝，但是存在Impala和Hadoop集群抢占资源的情况，进而可能影响Impala的查询性能（MR任务也可能被Impala影响），而后者可以提供稳定的高性能，但是需要持续的从Hadoop集群拷贝数据到Impala集群上，增加了ETL的复杂度。两种方式各有优劣，但是针对前一种部署方案，需要考虑如何分配资源的问题，首先在混合部署的情况下不可能再让Impalad进程常驻（这样相当于把每一个NodeManager的资源分出去了一部分，并且不能充分利用集群资源），但是YARN的资源分配机制延迟太大，对于Impala的查询速度有很大的影响，于是Impala很早

就设计了一种在YARN上完成Impala资源调度的方案——Llama（Low Latency Application MAster），它其实是一个AM的角色，对于Impala而言。它的要求是在查询执行之前必须确保需要的资源可用，否则可能出现一个Impalad的阻塞而影响整个查询的响应速度（木桶原理），Llama会在Impala查询之前申请足够的资源，并且在查询完成之后尽可能的缓存资源，只有当YARN需要将该部分资源用于其它工作时，Llama才会将资源释放。虽然Llama尽可能的保持资源，但是当混合部署的情况下，还是可能存在Impala查询获取不到资源的情况，所以为了保证高性能，还是建议独立部署。



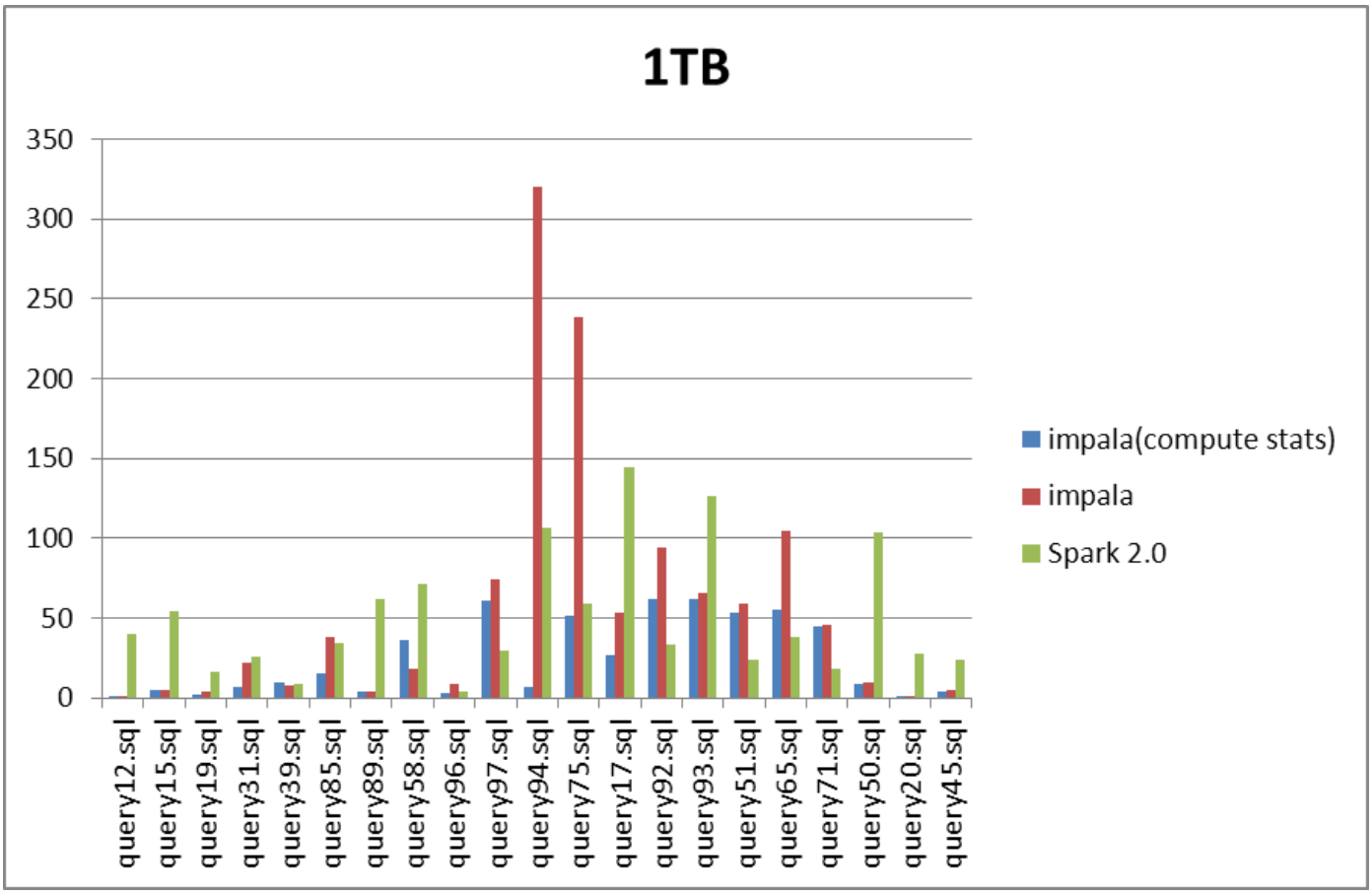
混合部署方式



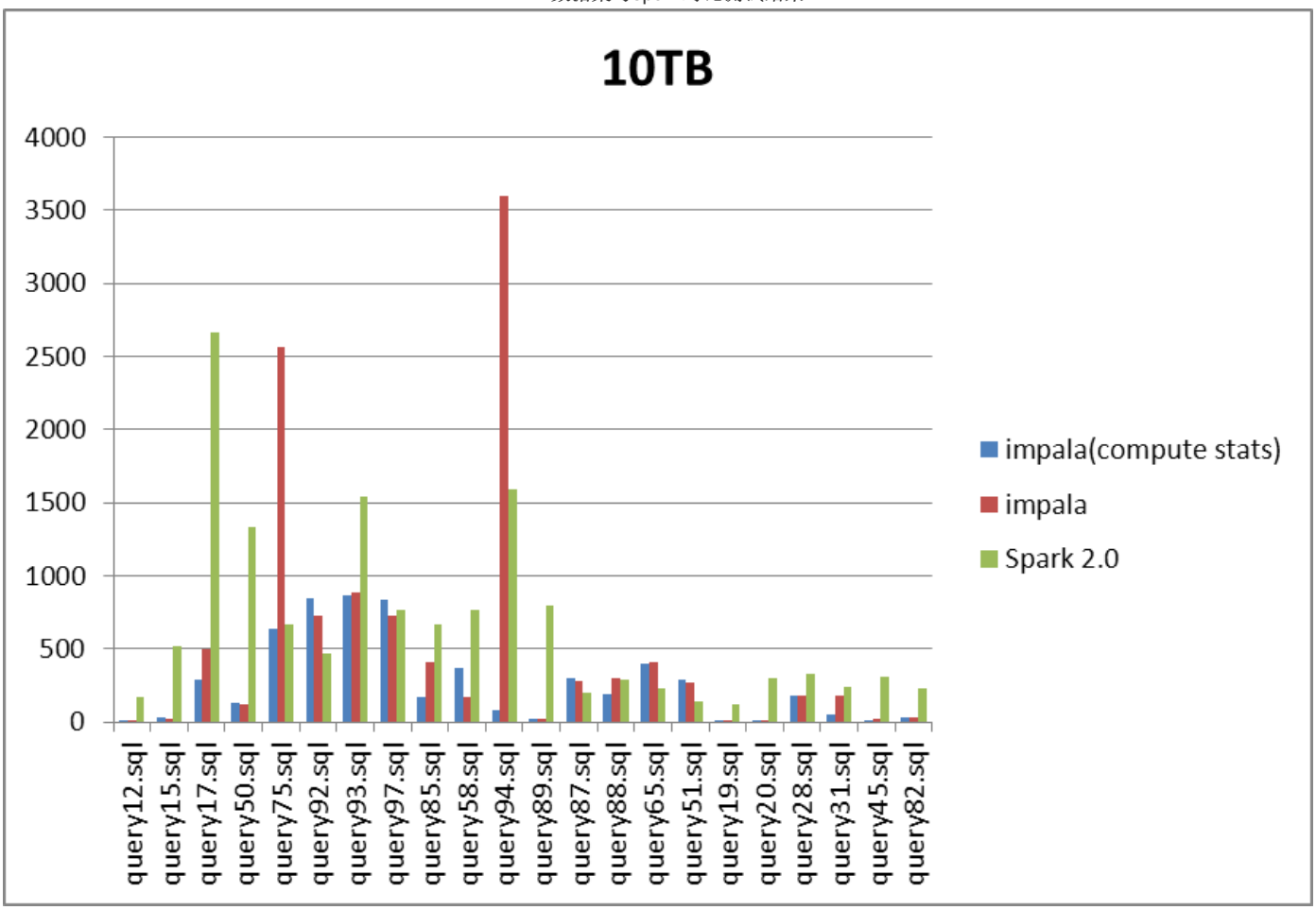
独立部署方式

测试

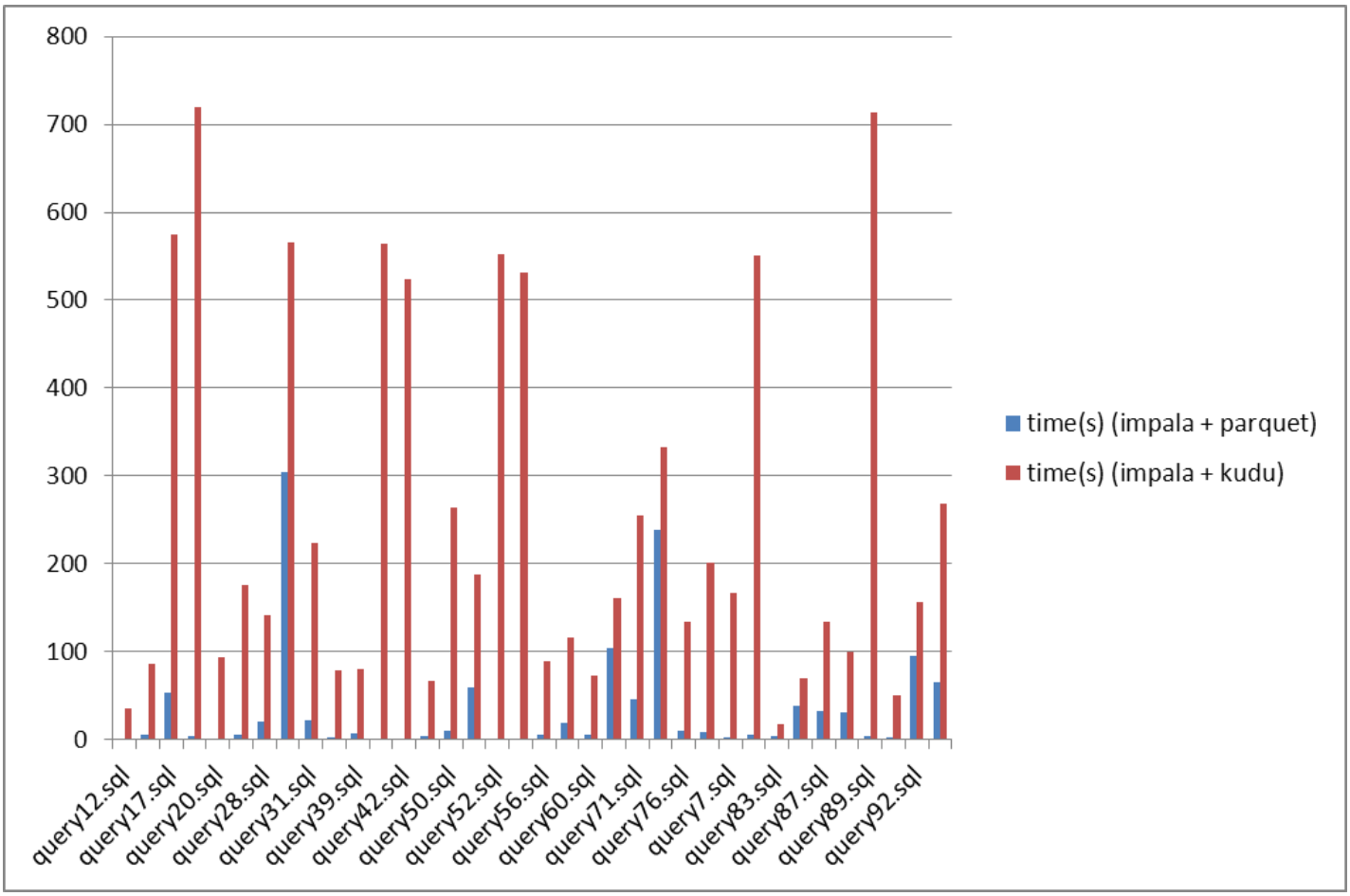
我们小组的同事对Impala做了一次基于TPCDS数据集的性能测试，分别基于1TB和10TB的数据集，可以看出，它的查询性能较之于Hive有数量级级别的提升，对比Spark SQL也有几倍的提升，Compute stat操作可以给Impala带来一定的查询优化，但是偶尔反而误导查询优化器以至于性能下降，最后我们还测试了Impala on Kudu，发现它并没有达到意料中的性能（几倍的差别）。唯一的缺憾是我们并没有对多用户并发场景下进行测试，不过从单个查询的资源消耗来看，C++实现的Impala对资源的消耗也是最少的，可以推断出在多用户下它仍然能满足快速响应的需求，最后是官方给出的多用户场景下的对比结果（有点故意黑Presto的感觉）。



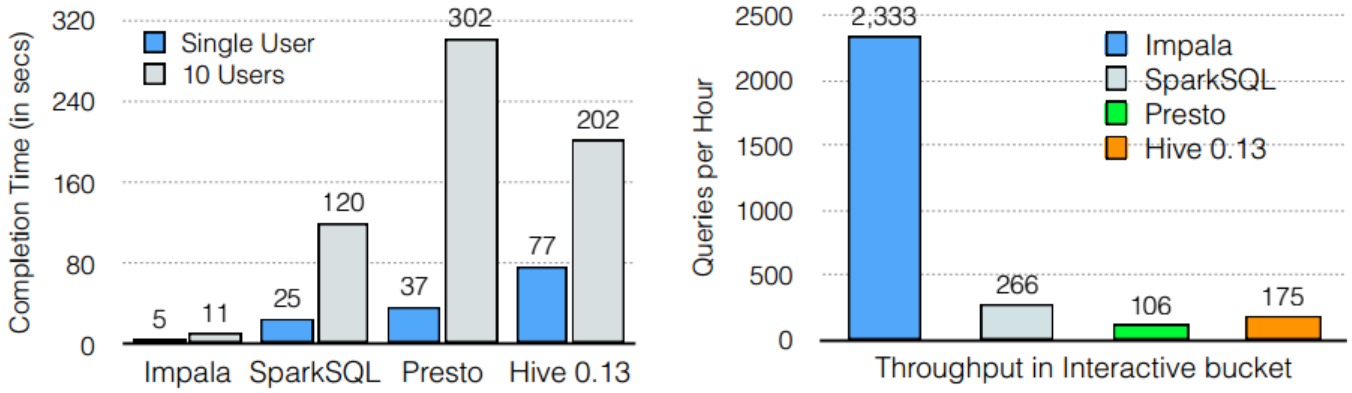
1TB数据集与spark对比测试结果



10TB数据集与spark对比测试结果



Impala on parquet与Impala on Kudu对比测试结果



并发测试结果

总结

本文主要介绍了Impala这个高性能的ad-hoc查询引擎，分别从使用、原理和部署等方面做了详细的分析，最终基于我们的测试结果也证实了它的高性能，区别于传统DBMS的MPP解决方案，例如Greenplum、Vertica、Teradata等，Impala更好的融入大数据（Hadoop/Spark）生态圈，更好的实现数据之间的流通，而传统MPP数据库，更倾向于数据自制。当然基于HDFS的实现导致Impala无法实现单条数据的实时更新，而只能批量的追加或者覆盖数据，虽然Cloudera也提供了Impala对于Kudu的支持，但是从性能测试结果看，目前查询性能还是不理想，而传统MPP数据库不仅可以支持单条数据的实时更新，甚至能够在保证查询性能的情况下支持较复杂的事务，这也是SQL-on-Hadoop查询引擎所望尘莫及的。但是无论如何，这类的查询引擎毕竟支持SQL引擎而不是一个完整的数据库系统，它提供给用户在大数据圈中高性能的查询服务，这也能够满足了大部分用户的需求。

参考

Impala: A Modern, Open-Source SQL Engine for Hadoop

Dremel: interactive analysis of web-scale datasets

Impala原理及其调优

Impala：新一代开源大数据分析引擎

Apache Impala Documents