17 | 从Dremel到Parquet（二）：他山之石的MPP数据库

2021-11-01 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 20:18 大小 18.61M

你好，我是徐文浩。

在上节课里，我们看到了 Dremel 这个系统的数据存储是怎么回事儿的。不过，只是一个支持复杂嵌套结构的列存储，还没有发挥 Dremel 百分之百的威力。像 Hive 也在 2011 年推出了自己的列存储方案 RCFile，并在后续不断改进提出了 ORC File 的格式。

列存储可以让一般的 MapReduce 任务少扫描很多数据，让很多 MapReduce 任务执行的时间从十几分钟乃至几个小时，下降到了几分钟。更短的反馈时间，使得数据分析师去探索数据，根据拿到的数据反馈不断从不同的角度去尝试分析的效率大大提高了。

不过，人们总是容易得陇望蜀的。当原先需要花几天时间写 MapReduce 程序才能分析数据的时候，我们希望能够通过写 SQL 跑分析数据。当原先 SQL 运行要 30 分钟、一个小时

的时候，我们通过列存储把 SQL 执行的时间缩短到 5 分钟。但是在这 5 分钟里，我们的数据分析师该干嘛呢？只能去倒杯咖啡发个呆么？所以，我们自然希望 SQL 在大数据集上，也能在几十秒，甚至是十几秒内得到结果。

所以 Google 并没有在列存储上止步，而是借鉴了多种不同的数据系统，搭建起了整个 Dremel 系统，真的把在百亿行的数据表上，常见 OLAP 分析所需要的时间，缩短到了 10 秒这个数量级上。那么，这节课我们就来看看 Dremel 是通过什么样的系统架构，做到这一点的。

和所有工程上的进展一样，Dremel 也是从很多过去的系统中汲取了养分：

第一，它从传统的 **MPP 数据库**，学到了数据分区和行列混合存储，并且把计算节点和存储节点放在同一台服务器上。

第二，它从**搜索引擎的分布式索引**，学会了如何通过一个树形架构，进行快速检索然后层层归并返回最终结果。

第三，它从 **MapReduce** 中借鉴了推测执行（Speculative Execution），来解决了少部分节点大大拖慢了整个系统的整体运行时间的问题。

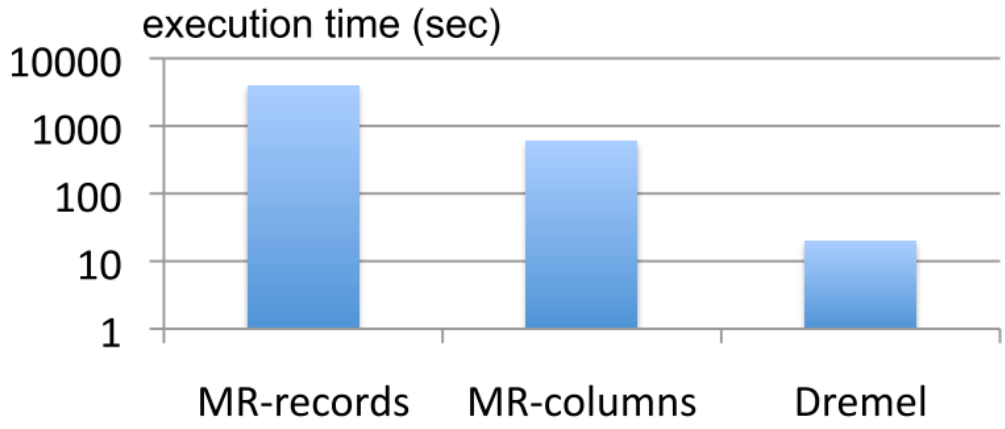
而这三个的组合，就使得 Dremel 最终将百亿行数据表的分析工作缩短到了 1 分钟以内。

通过这节课的学习，我希望你不仅能够学到 Dremel 的具体架构的设计，更能够学会在未来的架构设计工作中，博采众长，做出让人拍案叫好的系统设计。

瓶颈并不出现在硬件

Dremel 采用的列存储，已经极大地减少了我们扫描数据的浪费。在论文里，Google 给出了这样一组数据：在 3000 个节点上，查询一个 87TB、一共有 240 亿条数据的数据集，查询的内容是一个简单的 Word Count 程序。如果采用 MapReduce 去读取行存储的数据，那么需要读取 87TB 的数据。而如果采用列存储的话，因为只需要读取一列数据，所以只扫描了 0.5TB 的数据，整个 MapReduce 程序执行的时间，也缩短了整整一个数量级。

这也是为什么，在 Dremel 论文发表之后，开源社区很快跟进了这个支持嵌套的列存储的存储格式，也就催生了 Parquet 这个开源项目。



来自论文中的图10
同样的数据集，读取列存储比行存储快一个数量级，而使用Dremel比MapReduce快一个数量级

不过，如果你去看论文中的图 10，你会发现，**使用 Dremel 比传统的 MapReduce 读取列存储的数据还要再快一个数量级**。那这又是怎么做到的呢？请你和我接着一起往下看。

我们之前提过很多次，MapReduce 虽然伸缩性非常好，非常适合进行大规模的数据批处理，但是它也有一些明显的缺陷，其中很重要的一个问题，就是每个任务都有相对比较大的额外开销（overhead）。

所以即使有了 Hive，可以让分析师不用写程序，可以直接写 SQL，另外列存储也让我们需要扫描的数据大大减少了，但是 MapReduce 这个额外开销，始终还是会让我们的分析程序的运行时间在分钟级别。

而前面的图里我们可以看到，Dremel 则可以让我们这样的 SQL 跑在 10 秒级别。说实话，刚看到这个数据的时候，我是有点难以置信的。事实上，不只是我这样的工程师这么想，著名的Wired在 Dremel 发表之后就报道过，像 Berkeley 的教授阿曼多·福克斯（Armando Fox）就说，“如果你事先告诉我 Dremel 可以做什么，那么我不相信你可以把它做出来”。

> If you had told me beforehand me what Dremel claims to do, I wouldn’t have believed you could build it.

不过，回过头来，从硬件的性能来说，这看起来又是完全做得到的。论文里给出的实验数据里，是用 3000 个节点，去分析 0.5TB 的数据，这意味着每个节点只需要分析 167MB 的数据。即使是传统的 5400 转的机械硬盘，顺序读写的确也只需要数秒钟，再加上网络传输和 CPU 的计算时间，的确也就是个 10 秒钟上下的时间。

Dremel 系统架构

Dremel 之所以这么快，是因为它的底层计算引擎并不是 MapReduce。Dremel 一方面继承了很多 GFS/MapReduce 的思路，另一方面也从传统的 MPP（Massively Parallel Processing）数据库和搜索引擎的分布式检索模块，借鉴了设计思路。其实它的核心思路就是这四条：

第一点是**让计算节点和存储节点放在同一台服务器上**。MPP 数据库和搜索引擎的分布式索引的架构也是这样的。

第二点是**进程常驻，做好缓存，确保不需要大量的时间去做冷启动**。这一点，也跟 MPP 数据库和分布式索引采用的架构和优化手段类似。

第三点是**树状架构，多层聚合**，这样可以让单个节点的响应时间和计算量都比较小，能够快速拿到返回结果。这个架构，和搜索引擎的分布式索引架构是完全相同的。

最后一点则仍然来自于 GFS/MapReduce，一方面是**即使不使用 GFS，数据也会复制三份存放到不同的节点**。然后在计算过程中，Dremel 会**监测各个叶子服务器的执行进度**，对于“落后”的计算节点，会调度到其他计算节点，这个方式和 MapReduce 的思路是一样的。更进一步的，Dremel 还会**只扫描 98% 乃至 99% 的数据，就返回一个近似结果**。对于 Top K，求唯一数，Dremel 也会采用一些近似算法来加快执行速度。这个方法，也是我们在 MapReduce 中经常用到的。

那么下面，我们就对着论文中 Dremel 的系统架构图，一起来看一下它是如何组合 GFS/MapReduce、MPP 数据库，以及搜索引擎的系统架构，来实现一个能够在数十秒内返回分析结果的 OLAP 系统的。

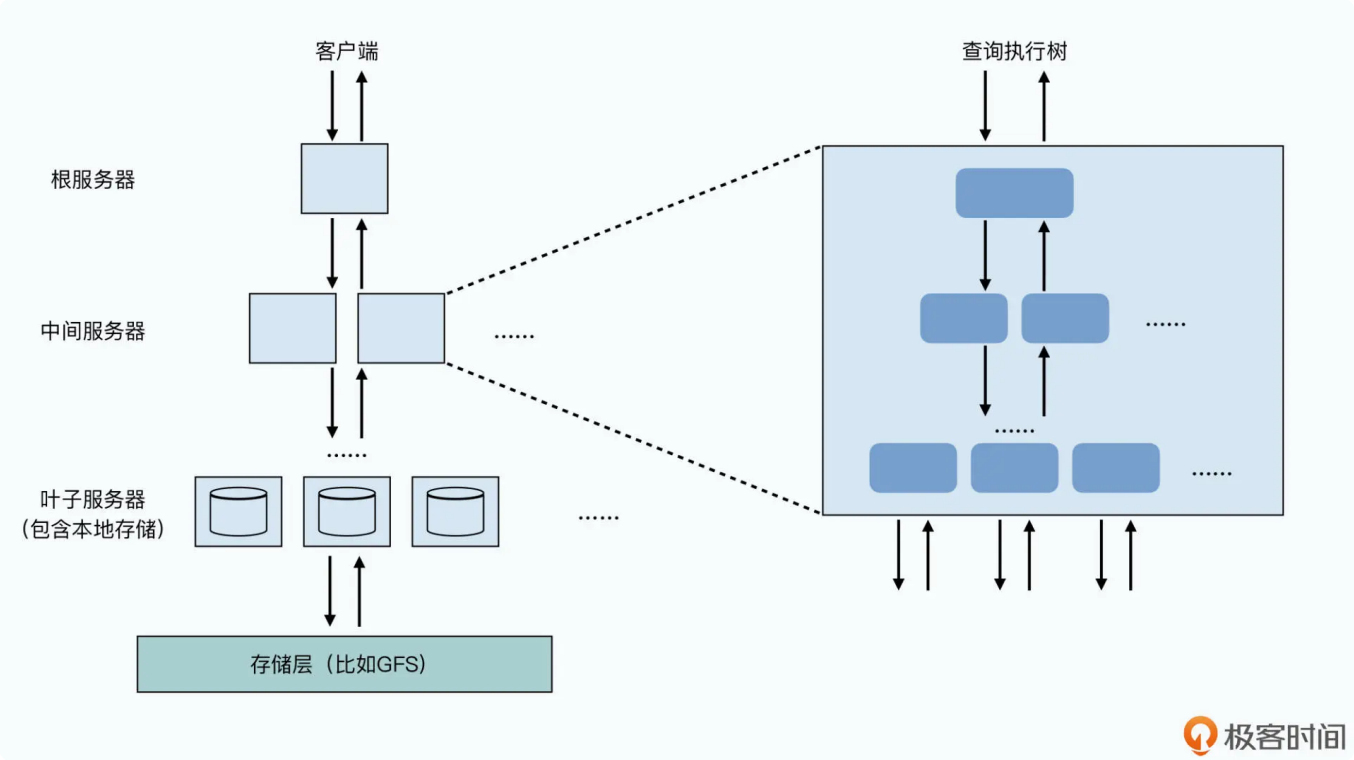
Dremel 采用了一个多层服务树的架构，整个服务树里面有三种类型的节点：

首先是**根服务器（root server）**，用来接收所有外部的查询请求，并且读取 Dremel 里各个表的 METADATA，然后把对应的查询请求，路由到下一级的服务树（serving tree）中。

然后是一系列的**中间服务器（intermediate servers），**中间服务器可以有很多层。比如第一层有 5 个服务器，那么每个服务器可以往下再分发下一层的 5 个服务器，它是一个树状结构，这也是服务树的这个名字的由来。我们所有查询 Dremel 系统的原始

SQL，每往下分发一层，就会重写（rewrite）一下，然后把结果在当前节点做聚合，再返回给上一层。

最下面是一层**叶子服务器（leaf servers）**，叶子服务器是最终实际完成数据查询的节点，也算是我们实际存储数据的节点。



来自论文中的图7， Dremel的整体系统架构图

光这样讲系统的架构实在还是太抽象，我们还是来看看论文里给到的 SQL 的例子：

```
1 SELECT A, COUNT(B) FROM T GROUP BY A
```

复制代码

这是一个我们在日常数据分析中很常见的 SQL，它是从某一个表里 T，按照某一个维度 A（比如国家、时间），看某一个统计指标 B（比如页面访问量、唯一用户数）这样的数据。这个 SQL 在 Dremel 上执行的过程是这样的。

首先，SQL 会发送到根服务器，根服务器会把整个 SQL 重写成下面这样的形式。

> SELECT A, SUM(c) FROM (R_1^1 UNION ALL ... R_n^1) GROUP BY A

其中的每一个 R_1^1 ... R_n^1，都是服务树的下一层的一个 SQL 的计算结果，那么下一层的 SQL 是这样的：

> R_i^1 = SELECT A, COUNT(B) AS c FROM T_i^1 GROUP BY A

这个解决办法其实一看就能看懂。因为原始的 SQL 是进行统计计数，那么我们只需要让中间服务器，分别去统计一部分分区数据的统计计数，再把它们累加到一起，就可以拿到最终想要的结果 1。这里的 R_i^1 就是对应中间服务器的中间结果，T_i^1 就是对应分配给当前中间服务器，需要计算的数据的分区。

事实上，这里面的 R_i^1 可以再用根服务器重写 SQL 的方式，进行再次重写，再往下拆分，我们可以有两层、三层乃至更多层的中间服务器。而到了最后一层，分发给叶子服务器的时候，就不能再往下分发了，叶子服务器会在它所分配到的分区上，执行对应的 SQL 并且返回。

行列混合存储的 MPP 架构

上节课我们学习过列存储的内容，我们知道 Dremel 的列存储本质上是**行列混合存储**的。所以每一个节点所存储的数据，是一个特定的分区（Partition），但是里面包含了这个分区所有行的数据。这样当数据到达叶子节点的时候，叶子节点需要执行的 SQL 只需要访问一台物理服务器。在这种情况下，我们可能有两种方案：

一种是对应的数据，就直接存放在叶子节点的服务器的本地硬盘上。这种方式，也是传统的 MPP 数据库采用的方式，也是 Dremel 系统在 2006 年，在 Google 内部开始使用的时候采用的方式，直到论文发表的 2009 年，这还是 Dremel 系统主要采用的方案。

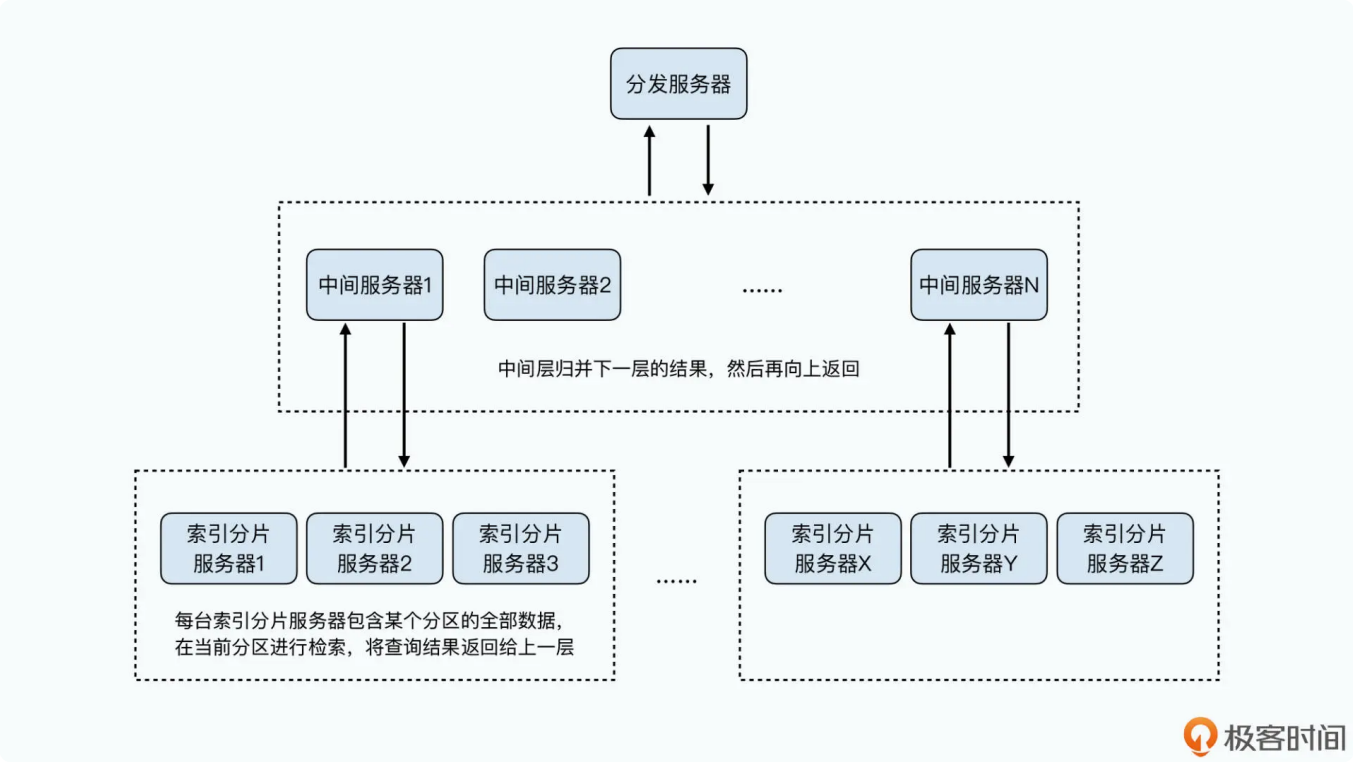
另一种方式，则是叶子节点本身不负责存储，而是采用一个共享的存储层，比如 GFS。Dremel 从 2009 年开始，就逐步把存储层全部迁移到了 GFS 上。

把数据存储和计算放在同一个节点，以及将用户 SQL 查询重写，并行分发到多个节点并且汇总所有节点的查询结果，是 MPP 数据库的常见方案。这也是为什么 Dremel 论文里说，它从 MPP 数据库里借鉴了很多解决问题的思路。

树形分发的搜索引擎架构

而这个一层层服务树分发的机制，则是借鉴了搜索引擎的分布式检索机制。数据分区到不同的叶子节点上，就是相当于我们把不同的文档**分片**到不同的索引分片服务器上。

每一个索引分片服务器，会完成自己分片数据上的检索工作，然后把结果返回给上一层的中间服务器。中间服务器也会在自己这一层，把检索结果再进行合并处理，再往上一层层返回，直到根服务器。



搜索引擎的分布式索引，也是采取类似的树状结构，每一个叶子节点本质都是某一个分区的数据检索结果返回到中间层服务器进行归并，通常归并的方式是Top K

我们可以拿一个例子来看看，Dremel 和搜索引擎的分布式索引有哪些相像之处。最合适的一个例子，就是求一个数据集中排序的 Top K，也就是前 K 项的返回结果，它对应的 SQL 就是这样的：

```
1 SELECT A, B, C FROM T ORDER BY D LIMIT K
```

复制代码

然后这个查询，在根服务器就会被重写成这样：

> SELECT A, B, C FROM (R_1^1 UNION ALL ... R_n^1) ORDER BY D LIMIT K

里面的每一个 R_1^1 ... R_n^1，都是服务树的下一层的一个 SQL 的计算结果，那下一层的 SQL 是这样的：

> R_i^1 = SELECT A, B, C AS c FROM T_i^1 ORER BY D LIMIT K

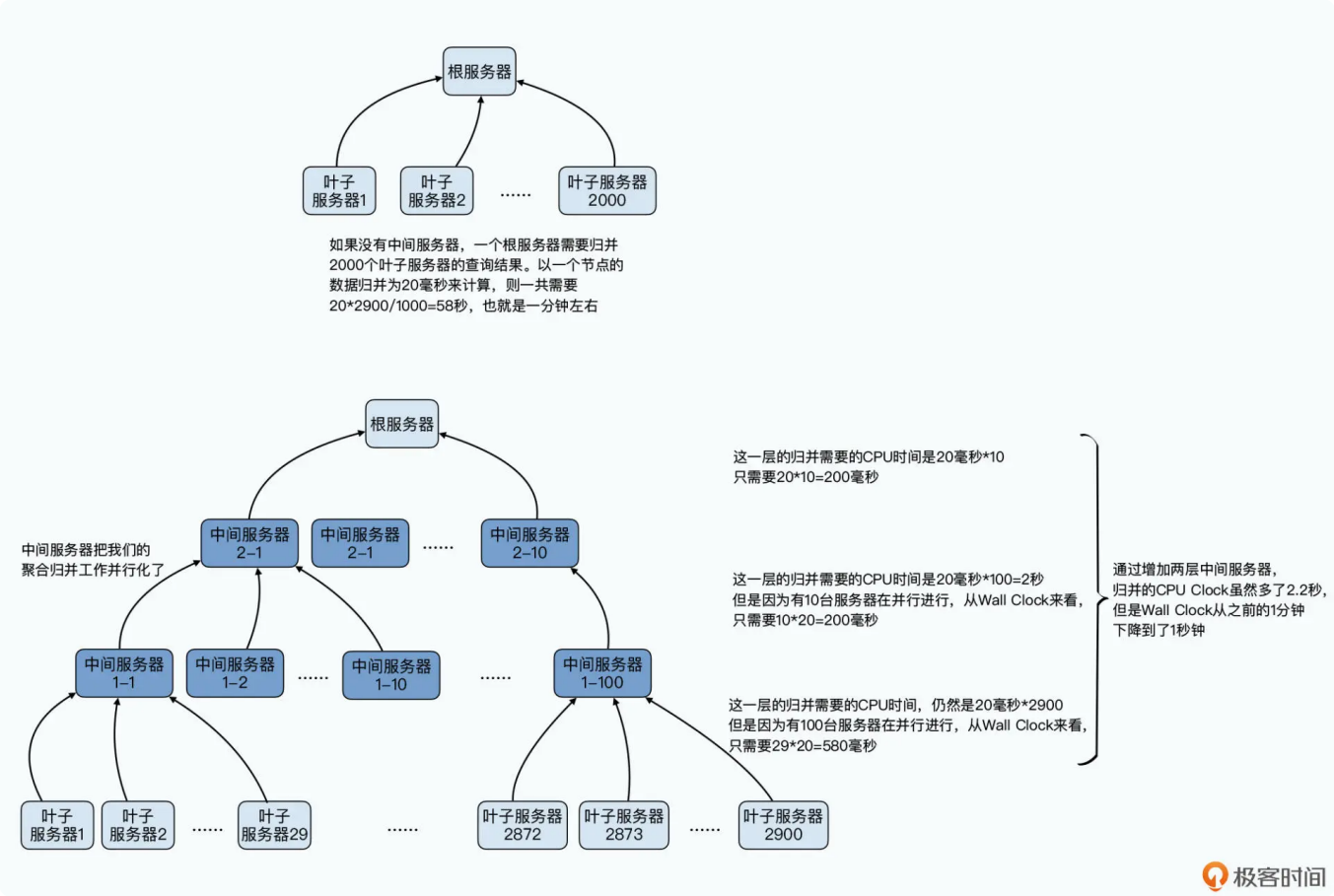
然后每一个 R_i^1 可以再用根服务器重写 SQL 的方式，进行再次重写，再往下拆分。也就是叶子服务器还是会获取自己分片数据的 TOP K，每一层都会去归并下一层的返回结果，并再计算一次 TOP K。

这个和搜索引擎的分布式索引的架构是完全一样的，唯一的差别是，搜索引擎计算 TOP K 的方式更加复杂一些，需要利用倒排索引，以及根据搜索的关键词，计算文档的一个“分数”来进行排名而已。

这个架构中最核心的价值，在于可以通过中间服务器来进行“垂直”扩张。并且通过“垂直”扩张，可以在计算量基本不变的情况下，通过服务器的并行，来缩短整个 SQL 所花费的时间。也就是通过增加更多的服务器，**让系统的吞吐量（Throughoutput）不变，延时（Latency）变小。**这个“垂直”扩张，并不是所谓的对硬件升级进行 Scale-Up，而是增加中间层服务器，增加归并聚合计算的并行度。

因为实际扫描数据，是在最终的叶子节点进行的，所以这一层花费的时间和性能是固定的。如果我们没有中间服务器，而是所有的叶子节点数据都直接归并到根服务器，那么性能瓶颈就会在根服务器上。

根服务器需要和 3000 个节点传输数据，并在根节点进行聚合。而这个聚合又在一个节点上，只能顺序进行，即使每一个叶子节点返回的数据，在根节点进行数据聚合只需要 20 毫秒，那么我们也需要 1 分钟才能完成 3000 个节点的数据聚合。



通过树形的中间层服务器的分发架构，我们增加了数据归并聚合的并行度，缩短了实际的SQL查询的响应时间

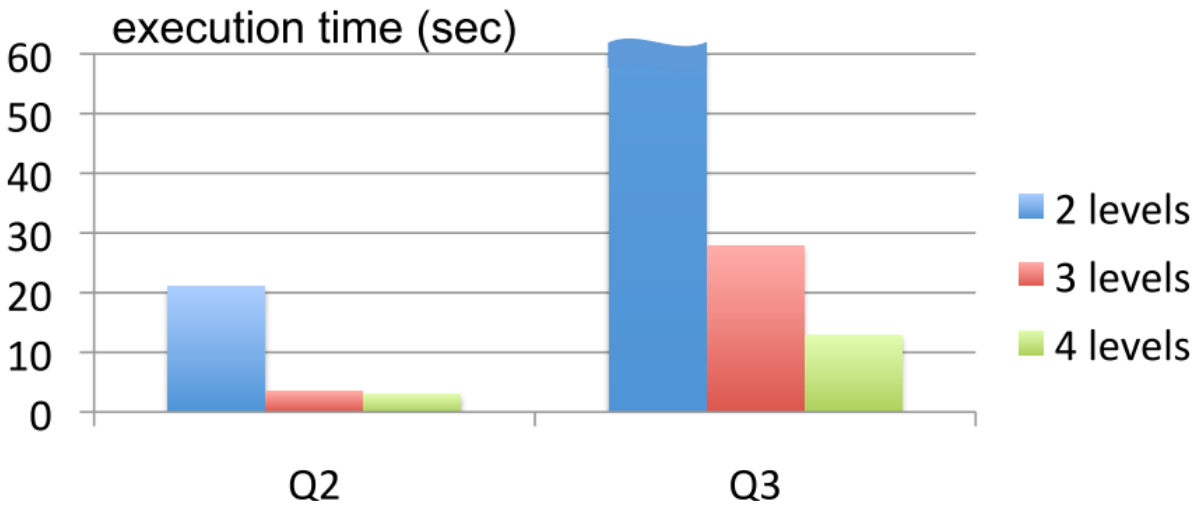
而如果我们在中间加入中间层的服务器，比如，我们有 100 个中间层的服务器，每个服务器下面聚合 30 个叶子服务器。那么中间层服务器就只需要 600 毫秒完成中间层的聚合，中间层的结果到根服务器也只需要 2 秒，我们可以在 3 秒内完成两层的聚合工作。

当然，在实际的 SQL 执行过程中，我们还有叶子节点扫描数据，以及数据在叶子节点和中间层，还有中间层和根服务器之间的网络传输开销，实际花费的时间会比这个多一些。但是**中间层，帮助我们把数据归并的工作并行化了。我们归并工作需要的 CPU 时间越多，这个并行化就更容易缩短整个查询的响应时间。**

我们的叶子节点越多，叶子节点返回的数据记录越多，增加中间层就越划算。论文里的实验部分针对不同的 SQL 和不同层数的中间服务器做了各种实验，你可以去仔细看一看。

这里，我们可以来对照着看看实验部分里，两个 SQL 中的 Q2 和 Q3：

> Q2：SELECT country, SUM(item.amount) FROM T2 GROUP BY country
> Q3：SELECT domain, SUM(item.amount) FROM T2 WHERE domain CONTAINS ‘ .net’ GROUP BY domain



来自论文中的图11
通过增加中间层，我们的SQL扫描的数据没有发生变化，但是整个任务的延时大大缩短了
这个对于返回的数据量越大，聚合工作越多的情况下越适用

其中，Q2 是按照国家进行数据聚合，因为国家的数量很少，所以每一个叶子节点返回的数据量也很小。但是即使这样，在没有中间节点的情况下，因为根服务器要和 3000 个叶子服务器一一通信、聚合数据，花费的时间也要 20 秒。而我们只要加上一个中间层，所花费的时间立刻缩短到了 3 秒，但是要注意，这个时候即使我们再增加中间层，时间也无法缩短了。

而里面的 Q3，是按照域名进行数据聚合。我们知道互联网上的域名数量特别多，在这个 SQL 中，最终一共会有 110 万个域名。没有中间层的时候，执行时间需要超过一分钟。增加了 100 个节点的中间层之后，时间就缩短了一半以上，而当我们在中间层再加一层，把整个服务器的树形结构变成 1:10:100:2900 的时候，执行时间能够再缩短一半，到 15 秒之内。

其实，这个树形垂直扩展的架构，也是搜索引擎能从无穷无尽的网页中，快速在几百毫秒之内给到你结果的核心所在。

来自 MapReduce 的“容错”方案

除了 MPP 数据库和搜索引擎之外，Dremel 也没有忘了向自家的前辈 MapReduce 借鉴经验。我们刚才看到，Dremel 的整个服务器集群也不小，实验里就动用了 3000 台服务器。那么一旦遇到这种情况，我们一样要面临“容错”的问题。

而 Dremel 和 MapReduce 一样，会遇到网络问题、硬件故障。乃至于个别叶子节点因为硬盘可能将坏未坏，虽然仍然能够读取数据，但是就是特别慢，这些它会遇到的问题，其

实 MapReduce 里都遇到过。

所以，Dremel 自然也就大大方方地借鉴了 MapReduce 和 GFS 里，已经用过的几个办法。

首先是虽然数据存储到了本地硬盘，也会有 **3 份副本**。这样，当我们有个别节点出现故障的时候，就可以把计算请求调度到另外一套有副本数据的节点上。

其次，是借鉴了 MapReduce 的**“推测执行”功能**，Dremel 也会监测叶子节点运行任务的进度。在 3000 个节点里，我们总会遇到一些节点跑起来特别慢，拖慢了整个系统返回一个查询结果的时间。往往 99% 的节点都算完了，大家等这几个节点要等上个三五分钟。这些节点无论是在 MapReduce 还是 Dremel 中都会存在，我们一般称它们为“掉队者”（Stragglers）。

而 Dremel 和 MapReduce 一样，一旦监测到掉队者的出现，它就会把任务再发给另外一个节点，避免因为单个节点的问题，导致整个任务的延时特别长。

另外，在 MapReduce 里，我们最终还是要等待所有的 Map 和 Reduce 函数执行完才给出结果。而在 Dremel 里，我们可以设置**扫描了 98% 或者 99% 的数据就返回一个近似的结果**。

一方面，从 Dremel 的实验数据来看，通常 99% 的到叶子节点处理的数据是低于 5 秒的，但是另外的少部分数据往往花费了非常长的时间，甚至会到几分钟。另一方面，Dremel 是一个交互式的分析系统，更多是给分析师分析数据给出结论，而不是生成一个用来财务记账的报表，数据差上个 1%~2% 并不重要。

小结

好了，到这里，Dremel 的架构我们就学习完了，那我们就一起来总结一下吧。

可以看到，Dremel 对于大数据集下的 OLAP 系统的设计，并没有止步于我们上节课所说的列存储。

通过**借鉴 MPP 数据库**，把计算和存储节点放在一起，以及通过**行列混合**的方式，Dremel 完成了数据的并行运算，而且缩减了需要扫描的数据。通过**借鉴搜索引擎的分布式索引系**

统，Dremel 搭建了一个树形多层的服务器架构，通过中间层服务器进行数据聚合，减少了整个系统计算和返回结果的延时。

而通过**借鉴 MapReduce 的容错机制**，Dremel 会把太慢的任务调度到其他拥有数据副本的节点里去，并且更激进地抛弃那些“掉队者”节点的数据，在只扫描了 98%~99% 的数据的时候，就返回结果，尽可能让每个 SQL 都能快速看到结果。

其实，从硬件层面的参数来看，Dremel 能够在几秒乃至几十秒内，扫描 240 亿条数据中的几列数据进行分析，的确是做得到的。**Dremel 本身也没有发明什么新算法、新架构，而是通过借鉴现有各类成熟的并行数据库、搜索引擎、MapReduce 搭建起了一个漂亮的框架，把大部分人眼里的不可能变成了可能。**相信这一点，对于所有想要做架构设计的同学来说，都会有所启发。

推荐阅读

过去的那么多节课里，我们读的都是至少十年前的“老”论文了。其实所有的这些系统都在不停地进化。Dremel 论文的几位作者，在 2020 年的 VLDB 里，就发表了一篇新论文叫做《Dremel: A Decade of Interactive SQL Analysis at Web Scale》。这篇论文讲述了 Dremel 系统后续的迭代更新，其中包括数据存储如何迁移到共享的 GFS 上、如何通过内存 Shuffle 架构提升 Dremel 的性能等等，很值得一读。从十年之后回顾看一个老系统，我们会看到技术架构是如何在不断的权衡、优化中进步的。

思考题

最后，按照惯例，还是给你留一道思考题。

Dremel 从 2009 年开始把数据存储，从叶子节点的本地硬盘迁移到了 GFS 上。那么，为什么一开始 Dremel 没有把数据存储就放在 GFS 上呢？放在 GFS 上，和放在本地硬盘上，分别会有什么好处和坏处呢？以及对于这些好处和坏处，我们又有什么应对方案呢？

欢迎你在留言区分享你的思考和答案，和其他同学一起交流，共同进步。

赞 1　提建议

上一篇　16 | 从Dremel到Parquet（一）：深入剖析列式存储

下一篇　18 | Spark：别忘了内存比磁盘快多少

精选留言 (4)

写留言

在路上
2021-11-02

徐老师好，我想Dremel一开始把数据放在硬盘上，是因为当时"计算和存储分离"还不是大数据领域的主流思想，MPP数据库把计算和存储放在一起的思路，在过去证明是有效的，Dremel借鉴过去的成功经验是理所当然的。在Dremel 2020年的论文的第3.1节提到"At the time, it seemed the best way to squeeze out maximal performance from an analytical system was by using dedicated hardware and direct-attached disks. As Dremel's workload g...

展开

2

峰

2021-11-01

好处：不用管存储的高可用，解决struggle的问题。
坏处：打破了数据计算在同节点的设计，造成一定网络开销，解决方法：gfs能够提供固定block位置的api。
问题：开源OLAP系统中，有像dremel这样可以加入中间层（层数> 1）的OLAP引擎吗? 以及如何确定中间层数。

展开

2

陈迪
2021-11-05

尝试回答思考题：采用GFS最明显的好处是，存储扩展容易！
分片存储存本地硬盘，不可避免的、由于本地硬盘存不下了，要人肉做数据搬迁 或者 加一个元数据层进行管理，这不就是GFS么
另外，Dremel这个多层树状汇聚，很拉风！！

展开

1

斜面镜子 Bill
2021-11-01

好处理解是本地访问性能和数据质量相对好保证，处理逻辑也相对简单。坏处就是弹性和IO的吞吐会比较限制。当然也想听听作者的解答。

展开

1