23 | 数据库查询串讲：重难点回顾+思考题答疑+知识全景图

2020-09-30 王磊

分布式数据库30讲 进入课程 >

讲述：王磊
时长 19:17 大小 17.67M

你好，我是王磊。

这一讲是我们课程的第三个答疑篇，我会和你一起回顾第 16 讲到 22 讲的主要内容，这部分内容是围绕着数据库的“查询”展开的。同时，我也会集中解答留给大家思考题，并回复一些大家关注的热点内容。

第 16 讲：慎用存储过程

第 16 讲，我首先分享了自己职业生涯中的一个小故事，说的是如何用 Oracle 的存程将程序执行速度提升了 60 倍。这是个值得骄傲的事情，但后来我也发现了存储过程的局限性，就是难以移植、难以调试。所以，我个人的建议是不使用或者少使用存储过程。

然后，我们对分布式数据库的支持现状做了介绍，只有 OceanBase 2.0 版本支持 Oracle 存储过程，但官方并不建议在生产环境中使用。Google 的 F1 中通过引入独立的 UDF Server 来支持存储过程，可以使用 Java 等多种高级语言进行开发，这样调试和迁移会更加方便，但是普通企业的网络带宽能否支撑这种架构还不好说。

最后，我们重点介绍了 VoltDB。它是一款内存分布式数据库，大量使用内存、单线程、主要使用存储过程封装 SQL 是它的三个特点。VoltDB 的存储过程也支持 Java 开发，更加开放。

这一讲的是思考题是“VoltDB 的设计思路很特别，在数据的复制上的设计也是别出心裁，既不是 NewSQL 的 Paxos 协议也不是 PGXC 的主从复制，你能想到是如何设计的吗？”

VoltDB 数据复制的方式是 K-safety，也叫做同步多主复制，其中 K 是指分区副本的数量。这种模式下，当前分区上的任何操作都会发送给所有副本去执行，以此来保证数据一致性。也就是说，VoltDB 是将执行逻辑复制到了多个分区上，来得到同样的结果，并不是复制数据本身。

对这个问题，“佳佳的爸”在留言区给出了一个标准答案。同时，针对存储过程的使用，“Jxin”和“佳佳的爸”两位同学都谈了自己的经验和体会，都讲得非常好，点赞。

同时，我也看到有的同学表达了不同的观点，这个我还是蛮能理解的。调试难、移植难这都是主观判断，没有统一的标准。难还是易，说到底是和个体能力有关系。但是，今天已经是软件工程化的时代，越来越重视协作，孤胆英雄的开发模式在多数情况下已经不适用了。

如果你的整个团队都能低成本地使用这项技能，确实可以考虑继续使用，但这样也不是完全没有风险。作为管理者，你肯定还要考虑团队技术栈和市场上多数程序员的技术栈是否一致，来降低人员变动带来的风险。我碰到过不少项目，开发语言选 C++ 或 Java 都可以，但就是因为 C++ 程序员太少，所以选择了 Java。

第 17 讲：自增主键 VS 随机主键

第 17 讲的核心内容是自增主键的使用，它是一个具体的特性，因为要依赖全局性约束，所以很有代表性。我们首先介绍了 MySQL 的自增主键，很多同学会认为它能够保证单调递增，但如果发生事务冲突时，自增主键是会留下空洞的。

而且，当并发很大时，也不能直接使用 MySQL 的自增列，因为会成为性能瓶颈。然后，我介绍了如何使用 Oracle 的 Sequence 配合应用系统共同支持海量并发。

在分布式数据库下，自增主键与 Range 分片共用会引发“尾部热点”问题，我们用 CockroachDB 与 YugabyteDB 的性能测试数据来验证这个判断。因为 Range 分片是一种普遍的选择，所以通常是舍弃自增主键转而用随机主键替换，仅保证 ID 的唯一性。具体技术方案分为数据库内置和应用系统外置两种，内置方案包括 UUID 和 Random，外置方案包括 Snowflake 算法。

这一讲的是思考题是“使用 Range 分片加单调递增主键会引发‘尾部热点’问题，但是使用随机主键是不是一定能避免出现‘热点’问题？”

答案是，随机主键可能会出现热点问题。因为按照 Range 分片原理，一张数据表初始仅有一个分片，它的 Key 范围是从无穷小到无穷大。随着数据量的增加，这个分片会发生分裂（Split），数据存储才逐渐散开。这意味着，在一段时间内，分片数量会远小于集群节点数量时，所以仍然会出现热点。

解决的方法就是采用预分片机制（Presplit），在没有任何数据的情况下，先初始化若干分片并分配不同的节点。这样在初始阶段，写入负载就可以被分散开，避免了热点问题。目前 Presplit 在分布式键值系统中比较常见，例如 HBase，但不是所有的分布式数据库都支持。

第 18 讲：HTAP VS Kappa

在第 18 讲中，我们讨论了 HTAP 的提出背景、现状和未来。HTAP 是 Gartner 提出的 OLTP 与 OLAP 一体化的解决思路，旨在解决数据分析的时效性。同年，LinkedIn 提出的 Kappa 架构也是针对这个问题。所以，我们将 HTAP 和 Kappa 作为两种互相比较的解决方案。

Kappa 下的主要技术产品包括 Kafka、Flink 等，是大数据生态的一部分，近年来的发展比较迅速。HTAP 的主要推动者是 OLTP 数据库厂商，进展相对缓慢，也没有强大的生态支持。所以，我个人更看好 Kappa 架构这条路线。

要实现 HTAP 就要在计算和存储两个层面支持 OLTP 和 OLAP，其中存储是基础。OLTP 通常使用行式存储，OLAP 则一般使用列式存储，存在明显差异。HTAP 有两种解决思

路。一种是 Spanner 的 PAX，一种新的融合性存储，在行存的基础上融合列存的特点。另外一种是 TiDB 提出的，借助 Raft 协议在 OLTP 与 OLAP 之间异步复制数据，再通过 OLAP 的特殊设计来弥补异步带来的数据不一致问题。

这一讲的思考题是“每次 TiFlash 接到请求后，都会向 TiKV Leader 请求最新的日志增量，本地 replay 日志后再继续处理请求。这种模式虽然能够保证数据一致性，但会增加一次网络通讯。你觉得这个模式还能优化吗？”

问题的答案是，可以利用 Raft 协议的特性进行优化。如果你有点记不清了，可以回到第 7 讲复习一下 Raft 协议。Raft 在同步数据时是不允许出现“日志空洞”的，这意味着如果 Follower 节点收到时间戳为 300 的日志，则代表一定已经收到了小于这个时间戳的所有日志。所以，在 TiFlash 接收到查询请求时，如果查询时间戳小于对应分片的最后写入时间戳，那么本地分片的数据一定是足够新的，不用再与 TiKV 的 Leader 节点通讯。

我在留言区看到“游弋云端”和“tt”同学都给出了自己的设计方案，都很棒。“tt”同学的方案非常接近于我给出的答案，“游弋云端”同学的心跳包方案也值得深入探讨。

第 19 讲：计算下推的各种形式

第 19 讲中，我们谈的核心内容是计算下推，这是计算存储分离架构下主要的优化方法。

计算下推又可以细分为很多场景，比较简单的处理是谓词下推，就是将查询条件推送到数据节点执行。但在不同的架构下实现难度也有差异，比如 TiDB 因为设计了“缓存写提交”所以就会更复杂些。分区键对于处理计算下推是个很好的帮助，在 PGXC 架构中常见，而 NewSQL 架构主要采用 Range 分片所以无法直接使用。

分布式数据库沿用了索引来加速计算。在分布式架构下，按照索引的实现方式可以分为分区索引和全局索引。分区索引可以保证索引与数据的同分布，那么基于索引的查询就可以下推到数据节点执行，速度更快。但是，分区索引无法实现全局性约束，比如就没法实现唯一索引，需要全局索引来实现。

不过，全局索引没有同分布的约束，写入数据会带来分布式事务，查询时也会有两轮通讯处理索引查询和回表，性能会更差。在分布式架构下要慎用全局索引。

这一讲的思考题是讲“将‘单表排序’操作进行下推，该如何设计一种有冗余的下推算法？”

排序是一个全局性的处理，任何全局性的控制对分布式架构来说都是挑战。这个设计的关键是有冗余。假如我们执行下面这一条 SQL，查询账户余额最多的 1,000 条记录。

```
1 select * from balance_info order by balance_num limit 1000;
```

复制代码

一个比较简单的思路是计算节点将这个 SQL 直接推送给所有数据节点，每个数据节点返回 top1,000，再由计算节点二次排序选择前 1,000 条记录。

不过，这个方式有点太笨拙。因为当集群规模比较大时，比如有 50 个节点，计算节点会收到 50,000 条记录，其中 49,000 都是无效的，如果 limit 数量再增加，那无效的数据会更多。这种方式在网络传输上不太经济，有一点像读放大情况。

我们可以基于集群节点的数量适当缩小下推的规模，比如只取 top 500，这样能够降低传输成本。但相应地要增加判断逻辑，因为也许数据分布很不均衡，top 1,000 账户都集中在某个节点上，那么就要进行二次通讯。这个方式如果要再做优化，就是在计算节点保留数据统计信息，让数据量的分配符合各节点的情况，这就涉及到 CBO 的概念了。

第 20 讲：关联查询经典算法与分布式实现

在第 20 讲，我们讨论了关联查询（Join）的实现方案。关联查询是数据库中比较复杂的计算，经典算法分为三类，嵌套循环、排序归并和哈希。这三类算法又有一些具体的实现，我们依次做了介绍，其中哈希算法下的 Grace 哈希已经有了分布式执行的特点。

有了算法的基础，我又从分布式架构的角度讨论了关联查询的实现。首先涉及到并行执行框架的问题，多数产品的执行框架比较简单，只能做到计算下推这种比较简单的并行。因为数据节点之间是不通讯的，所以计算节点容易成为瓶颈。另外一些产品，比如 OceanBase 和 CockroachDB 引入了类似 MPP 的机制，允许数据节点之间交换数据。

我们把关联查询这个问题聚焦到大小表关联和大表关联两个场景上。大小表关联的解决方案是复制表方式，具体又包括静态和动态两种模式。大表关联则主要通过数据重分布来实

现，这个过程需要数据节点之间交换数据，和上一段的并行执行框架有很密切的关系。

这一讲的思考题是“当执行 Hash Join 时，在计算逻辑允许的情况下，建立阶段会优先选择数据量较小的表作为 Inner 表，我的问题就是在什么情况下，系统无法根据数据量决定 Inner 表呢？”

选择数据量较小的作为 Inner 表，这是典型的基于代价的优化，也就是 CBO（Cost Based Optimizer），属于物理优化阶段的工作。在这之前还有一个逻辑优化阶段，进行基于关系代数运算的等价转化，有时就是计算逻辑限制了系统不能按照数据量来选择 Inner 表。比如执行左外连接（Left Outer Join），它的语义是包含左表的全部行（不管右表中是否存在与它们匹配的行），以及右表中全部匹配的行。这样就只能使用右表充当 Inner 表并在之上建哈希表，使用左表来当 Outer 表，也就是我们的驱动表。

第 21 讲：查询执行引擎的三个模型

第 21 讲，我们的关键词是查询执行引擎，它的责任是确保查询计划能被快速执行，而执行速度则取决于引擎采用的执行模型。执行模型分为三种，火山模型、向量化模型和代码生成。

火山模型是多数数据库使用的模型，有 20 年的历史，运行非常稳定。火山模型由一组运算符嵌套组成，运算符之间低耦合，通用性高，但它的缺点是无法使用现代 CPU 的特性，尤其是虚函数过多。

向量化模型将运算符的输入从行集合变成向量块，减少了调用虚函数的次数，也提高了 CPU 使用效率。

代码生成的逻辑则是通过编译器生成针对性的代码，从根本上解决虚函数过多的问题。

向量化模型和代码生成是现代高效查询模型的代表，已经获得越来越多认可，在很多数据库中被使用。TiDB 和 CockroachDB 都进行了向量化改造，而 OceanBase 也实现了表达式级别的代码生成。

这一讲的思考题是“基础软件演进中一个普遍规律，每当硬件技术取得突破后就会引发软件的革新。那么，我的问题就是你了解的基础软件中，哪些产品分享了硬件技术变革的红利呢？”

就像问题中所说的，每次硬件技术的突破都会引发软件的革新，比如 VoltDB 出现的背景就是内存技术成熟，价格日益降低，即使使用内存作为主存储设备也有足够的商业竞争力。

通过这一讲，你应该已经了解到，查询引擎的优化就是围绕着现代 CPU 特性展开的。而在第 22 讲存储引擎部分，我介绍了 WiscKey 模型，它是伴随着硬盘技术的发展而提出的，具体来说就是 SSD 的技术。

SSD 的物理结构与传统的 HDD 非常不同，没有物理磁头，所以寻址成本更低对于随机写支持等更好，但是反复擦写却更影响 SSD 的使用寿命。WiscKey 模型就是基于适合这两个特性提出的。随着 SSD 价格逐步降低，未来很可能成为服务器的标准配置。

第 22 讲：存储引擎的优化选择

第 22 讲，我们主要谈的是存储引擎，也就是数据落盘的最后一步。

在开始的部分，我们先引入 RUM 猜想这个框架，指出任何数据结构，只能在读放大、写放大和空间放大三者之间优化两项。

然后，我们又回到数据库架构下，分析了 B+ Tree 与 LSM 的区别。它们并不是简单地读优化和写优化。LSM 的两种策略 Tiered 和 Leveled 也会带来不同的效果，其中 Leveled 是 RocksDB 采用的模型，适用范围更广。

因为 RocksDB 是一个优秀的单机存储引擎，所以 TiDB 和 CockroachDB 最初都直接引入了它。但是随着产品的演进，TiDB 和 CockroachDB 分别推出了自己的存储引擎 TiTan 和 Pebble。

Titan 是借鉴了新的存储模型 WiscKey，与 LSM 最大的差异是将 Value 分离出来单独存储，这样的好处是在 Compact 环节减少了写放大。

选择 Pebble 的不是为了优化模型，而是出于工程上的考虑。一方面是 Go 语言调用 C++ 编写 RocksDB 是有额外的延迟，另一方面是 RocksDB 的不断膨胀引入了更多的变更风险。而 Pebble 使用 Go 语言开发，更加小巧，满足了工程方面的要求。

这一讲的思考题是“Scan 操作是否可以使用 Bloom Filter 来加速，如果可以又该如何设计呢？”

Bloom Filter 是很有意思的数据结构，通过多个 Hash 函数将一个数值映射到某几个字节上。这样用少量的字节就可以存储大量的数值，同时能快速地判断某个数值是否存在。虽然没有做映射的数值会有一定概率的误报，但可以保证“数值不存在”是绝对准确的，这就是假阳性。

这种模式显然是不能直接支持 Scan 操作的，这是需要将数值做一定的转化。这个方法在 RocksDB 中称为“Prefix Bloom Filter”，也就是取 Key 的左前缀（Prefix）进行判断。因为 K/V 系统是按照 Key 字典序排列的，那就是说相邻的 Key 通常具有相同的 Prefix，这种匹配方式相当于对一组 Key 做了检验，可以更好地适应 Scan 的特点。

对这个问题，“扩散性百万咸面包”和“可怜大灰狼”两位同学都给出了很准确的答案，点赞。

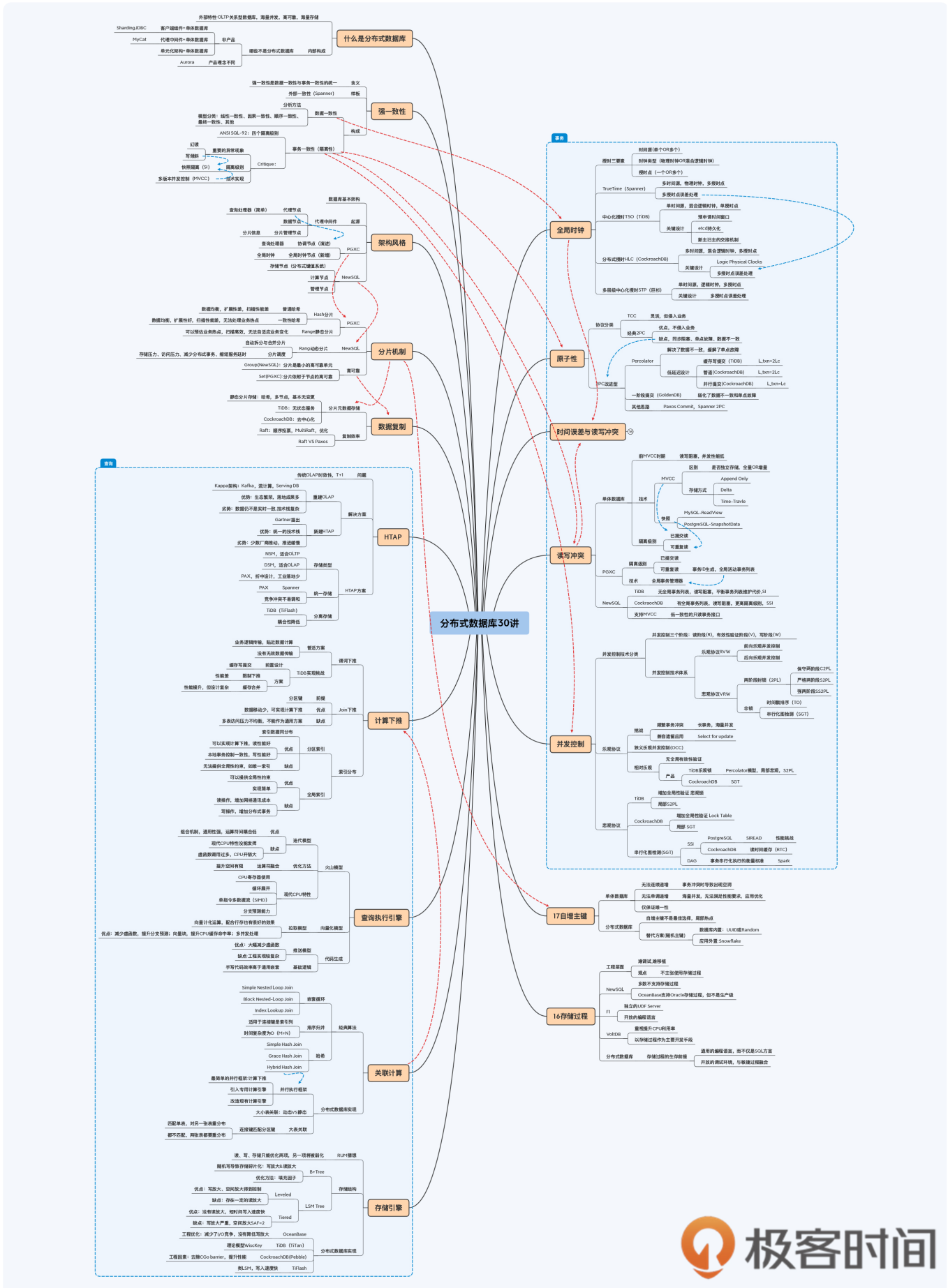
小结

第 16 到第 22 这七讲大多是围绕着查询这个主题展开的。之所以安排这么大的篇幅，是因为我认为对数据库来说，查询是除了事务以外最重要的功能。

当然 OLTP 的查询功能和 OLAP 还有很大的区别，也不能满足所有的查询需求。但了解了这些，可以让我们更准确地管理对分布式数据库的预期。如果你对查询场景更有兴趣，希望这些内容能够为你奠定一个基础，未来更高效地学习 OLAP 数据库的相关内容。

如果你对今天的内容有任何疑问，欢迎在评论区留言和我一起讨论。要是你身边的朋友也对数据库的查询执行过程感兴趣，你也可以把今天这一讲分享给他，我们一起讨论。

分布式数据全景图 3/4


分布式数据库30讲
什么是分布式数据库
强一致性
架构风格
分片机制
数据复制
事务
全局时钟
原子性
时间误差与读写冲突
读写冲突
并发控制
17自增主键
16存储过程
查询
HTAP
计算下推
查询执行引擎
关联计算
存储引擎

提建议

上一篇 22 | RUM猜想：想要读写快还是存储省？又是三选二

下一篇 24 | 全球化部署：如何打造近在咫尺且永不宕机的数据库？

精选留言 (1)

写留言

Geek_2e6a7e
2020-09-30

百亿级的查询复杂查询为啥不用ES,而是自研Pharos?

展开

作者回复: 因为更快呀:)

1