12 引擎拓展：解读当前流行的分布式存储引擎

这一讲是存储引擎模块的最后一讲，通过这一个模块的学习，相信你已经对存储引擎的概念、使用方法与技术细节有了全方位的认识。本讲我们先总结一下模块二的主要内容，并回答大家提到的一些典型问题；而后我会介绍评估存储引擎的三个重要元素；最后为你介绍目前比较流行的面向分布式数据库的存储引擎。

让我们先进行本模块的内容回顾。

存储引擎回顾

存储引擎是数据库的核心组件，起到了物理模型与逻辑模型之间的沟通作用，是数据库重要功能，是数据写入、查询执行、高可用和事务等操作的主要承担者。可谓**理解存储引擎也就掌握了数据库的主要功能。**

在这个模块里，我首先向你介绍了存储引擎在整个数据库中的定位，点明了它其实是本地执行模块的组成部分；而后通过内存与磁盘、行式与列式等几组概念的对比，介绍了不同种类的存储引擎的实现差异；并最终说明了分布式数据库存储引擎的特点，即面向内存、列式和易于散列。

在第 8 讲中，我介绍了分布式数据库的索引。着重说明了存储引擎中大部分数据文件其实都是索引结构；而后带着你一起探讨了典型分布式数据库存储引擎的读取路径，并介绍了该路径上的一些典型技术，如索引数据表、内存跳表、布隆过滤和二级索引等。

接着我介绍了一个在分布式数据库领域内非常流行的存储引擎：LSM 树。介绍了其具体的结构、读写修改等操作流程；重点说明了合并操作，它是 LSM 树的核心操作，直接影响其性能；最后介绍了 RUM 假说，它是数据库优化的一个经典取舍定律。

最后，我们探讨了存储引擎最精华的概念，就是事务。我用了两讲的篇幅，详细为你阐述事务的方方面面。总结一下，事务其实是数据库给使用者的一个承诺，即 ACID。为了完成这个承诺，数据库动用了存储引擎中众多的功能模块。其中最重要的事务管理器，同时还需要页缓存、提交日志和锁管理器等组件来进行配合。故在实现层面上，事务的面貌是很模糊的，它同时具备故障恢复和并发控制等特性，这是由其概念是建立在最终使用侧而造成的。

事务部分我们主要抓住两点：故障恢复+隔离级别。前者保证了数据库存储数据不会丢失，后者保证并发读写数据时的完整性；同时我们要将事务与模块一中的分布式一致性做区别，详细内容请你回顾第 5 讲。

在事务部分，有同学提到了下面这个问题，现在我来为你解答。

当内存数据刷入磁盘后，同时需要对日志做“截取”操作，这个截取的值是什么？

这个“截取”是一个形象的说法，也就是可以理解为截取点之前的数据已经在输入磁盘中。当进行数据库恢复的时候，只要从截取点开始恢复数据库即可，这样大大加快了恢复速度，同时也释放了日志的空间。这个截取点，一般被称为检查点。相关细节，你可以自行学习。

以上我们简要回顾了本模块的基本知识。接下来，我将带你领略当代分布式数据库存储引擎的一些风采。但是开始介绍之前，我们需要使用一个模型来评估它们的特点。

评估存储引擎的黄金三角

存储引擎的特点千差万别，各具特色。但总体上我们可以通过三个变量来描述它们的行为：缓存的使用方式，数据是可变的还是不可变的，存储的数据是有顺序的还是没有顺序的。

缓存形式

缓存是说存储引擎在数据写入的时候，首先将它们写入到内存的一个片段，目的是进行数据汇聚，而后再写入磁盘中。这个小片段由一系列块组成，块是写入磁盘的最小单位。理想状态是写入磁盘的块是满块，这样的效率最高。

大部分存储引擎都会使用到缓存。但使用它的方式却很不相同，比如我将要介绍的 WiredTiger 缓存 B 树节点，用内存来抵消随机读写的性能问题。而我们介绍的 LSM 树是用缓存构建一个有顺序的不可变结构。故**使用缓存的模式是衡量存储引擎的一**

个重要指标。

可变/不可变数据

存储的数据是可变的还是不可变的，这是判断存储引擎特点的另一个维度。不可变性一般都是以追加日志的形式存在的，其特点是写入高效；而可变数据，以经典 B 树为代表，强调的是读取性能。故一般认为可变性是区分 B 树与 LSM 树的重要指标。但 BW-Tree 这种 B 树的变种结构虽然结构上吸收了 B 树的特点，但数据文件是不可变的。

当然不可变数据并不是说数据一直是不变的，而是强调了是否在最影响性能的写入场景中是否可变。LSM 树的合并操作，就是在不阻塞读写的情况下，进行数据文件的合并与分割操作，在此过程中一些数据会被删除。

排序

最后一个变量就是数据存储的时候是否进行排序。排序的好处是对范围扫描非常友好，可以实现 between 类的数据操作。同时范围扫描也是实现二级索引、数据分类等特性的有效武器。如本模块介绍的 LSM 树和 B+ 树都是支持数据排序的。

而不排序一般是一种对于写入的优化。可以想到，如果数据是按照写入的顺序直接存储在磁盘上，不需要进行重排序，那么其写入性能会很好，下面我们要介绍的 WiscKey 和 Bitcask 的写入都是直接追加到文件末尾，而不进行排序的。

以上就是评估存储引擎特点的三个变量，我这里将它们称为**黄金三角**。因为它们是互相独立的，彼此并不重叠，故可以方便地评估存储引擎的特点。下面我们就试着使用这组黄金三角来评估目前流行的存储引擎的特点。

B 树类

上文我们提到过评估存储引擎的一个重要指标就是数据是否可以被修改，而 B 树就是可以修改类存储引擎比较典型的一个代表。它是目前的分布式数据库，乃至于一般数据库最常采用的数据结构。它是为了解决搜索树（BST）等结构在 HDD 磁盘上性能差而产生的，结构特点是高度很低，宽度很宽。检索的时候从上到下查找次数较少，甚至如 B+ 树那样，可以完全把非叶子节点加载到内存中，从而使查找最多只进行一次磁盘操作。

下面让我介绍几种典型的 B 树结构的存储引擎。

InnoDB

InnoDB 是目前 MySQL 的默认存储引擎，同时也是 MariaDB 10.2 之后的默认存储引擎。

根据上文的评估指标看，它的 B+ 树节点是可变的，且叶子节点保存的数据是经过排序的。同时由于数据的持续写入，在高度不变的情况下，这个 B+ 树一定会横向发展，从而使原有的一个节点分裂为多个节点。而 InnoDB 使用缓存的模式就是：为这种分裂预留一部分内存页面，用来容纳可能的节点分裂。

这种预留的空间其实就是一种浪费，是空间放大的一种表现。用 RUM 假设来解释，InnoDB 这种结构是牺牲了空间来获取对于读写的优化。

在事务层面，InnoDB 实现了完整的隔离级别，通过 MVCC 机制配合各种悲观锁机制来实现不同级别的隔离性。

WiredTiger

WiredTiger 是 MongoDB 默认的存储引擎。它解决了原有 MongoDB 必须将大部分数据放在内存中，当内存出现压力后，数据库性能急剧下降的问题。

它采用的是 B 树结构，而不是 InnoDB 的 B+ 树结构。这个原因主要是 MongoDB 是文档型数据库，采用内聚的形式存储数据（你可以理解为在关系型数据库上增加了扩展列）。故这种数据库很少进行 join 操作，不需要范围扫描且一次访问就可以获得全部数据。而 B 树每个层级上都有数据，虽然查询性能不稳定，但总体平均性能是要好于 B+ 树的。

故 WiredTiger 首先是可变数据结构，同时由于不进行顺序扫描操作，数据也不是排序的。那么它是如何运用缓存的呢？这个部分与 InnoDB 就有区别了。

在缓存中每个树节点上，都配合一个更新缓冲，是用跳表实现的。当进行插入和更新操作时，这些数据写入缓冲内，而不直接修改节点。这样做的好处是，跳表这种结构不需要预留额外的空间，且并发性能较好。在刷盘时，跳表内的数据和节点页面一起被合并到磁盘上。

由此可见，**WiredTiger 牺牲了一定的查询性能来换取空间利用率和写入性能**。因为查询的时候出来读取页面数据外，还要合并跳表内的数据后才能获取最新的数据。

BW-Tree

BW-Tree 是微软的 Azure Cosmos DB 背后的主要技术栈。它其实通过软件与硬件结合来实现高性能的类 B 树结构，硬件部分的优化使用 Llama 存储系统，有兴趣的话你可以自行搜索学习。我们重点关注数据结构方面的优化。

BW-Tree 为每个节点配置了一个页面 ID，而后该节点的所有操作被转换为如 LSM 树那样的顺序写过程，也就是写入和删除操作都是通过日志操作来完成的。采用这种结构很好地解决了 B 树的写放大和空间放大问题。同时由于存在多个小的日志，并发性也得到了改善。

刷盘时，从日志刷入磁盘，将随机写变为了顺序写，同样提高了刷盘效率。我们会发现，BW-Tree 也如 LSM 树一样存在读放大问题，即查询时需要将基础数据与日志数据进行合并。而且如果日志太长，会导致读取缓慢。而此时 Cosmos 采用了一种硬件的解决方案，它会感知同一个日志文件中需要进行合并的部分，将它们安排在同一个处理节点，从而加快日志的收敛过程。

以上就是典型的三种 B 树类的存储引擎，它们各具特色，对于同一个问题的优化方式也带给我们很多启发。

LSM 类

这个模块我专门用了一个完整篇章来阐述它的特点，它是典型的不可变数据结构，使用缓存也是通过将随机写转为顺序写来实现的。

我们在说 LSM 树时介绍了它存储的数据是有顺序的，其实目前有两种无顺序的结构也越来越受到重视。

经典存储

经典的 LSM 实现有 LeveledDB，和在其基础之上发展出来的 RocksDB。它们的特点我们之前有介绍过，也就是使用缓存来将随机写转换为顺序写，而后生成排序且不可变的数据。**它对写入和空间友好，但是牺牲了读取性能。**

Bitcask

Bitcask 是分布式键值数据库 Riak 的一种存储引擎，它也是一种典型的无顺序存储结构。与前面介绍的典型 LSM 树有本质上的不同，它没有内存表结构，也就是它根本不进行缓存而是直接将数据写到数据文件之中。

可以看到，其写入是非常高效的，内存占用也很小。但是如何查询这种“堆”结构的数据呢？答案是在内存中有一个叫作 Keydir 的结构保存了指向数据最新版本的引用，旧数据依然在数据文件中，但是没有被 Keydir 引用，最终就会被垃圾收集器删除掉。Keydir 实际上是一个哈希表，在数据库启动时，从数据文件中构建出来。

这种查询很明显改善了 LSM 树的读放大问题，因为每条数据只有一个磁盘文件引用，且没有缓存数据，故只需要查询一个位置就可以将数据查询出来。但其缺陷同样明显：不支持范围查找，且启动时，如果数据量很大，启动时间会比较长。

此种结构优化了写入、空间以及对单条数据的查找，但牺牲了范围查找的功能。

WiscKey

那么有没有一种结构，既能利用无顺序带来的高速写入和空间利用率上的优点，又可以支持非常有用的范围查询呢？WiscKey 结构正是尝试解决这个问题的一个手段。

它的特点是将 Key 和 Value 分别放在两个文件中。Key 还是按照 LSM 树的形式，这样就保证了 Key 是有顺序的，可以进行范围扫描。同时使用 LSM 树，即不需要将所有的 Key 放到内存里，这样也解决了 Bitcask 加载慢的问题。

而 Value 部分称为 vLogs（value Logs），其中的数据是没有顺序的。这种结构适合更新和删除比较少的场景，因为范围扫描会使用随机读，如果更新删除很多，那么其冲突合并的效率很低。同时在合并操作的时候，需要扫描 Key 而后确定合并方案，这个在普通的 LSM 树中也是不存在的。

WiscKey 非常适合在 SSD 进行运行，因为读取 Value 需要进行随机读取。目前 dgraph.io 的 Badger 是该模式比较成熟的实现。

总结

到这里，这一讲内容就说完了。我带你回顾了第二模块的主要内容，这是一个基础知识普及模块，将为接下来的分布式模块打下基础。同时相对于传统关系型数据库，分布式数据库的存储引擎也有其自身特点，如 LSM 树结构，你需要认真掌握这种结构。

上一页 下一页