# 掌握了LSM架构，你就掌握了90%的分布式数据库 已付费

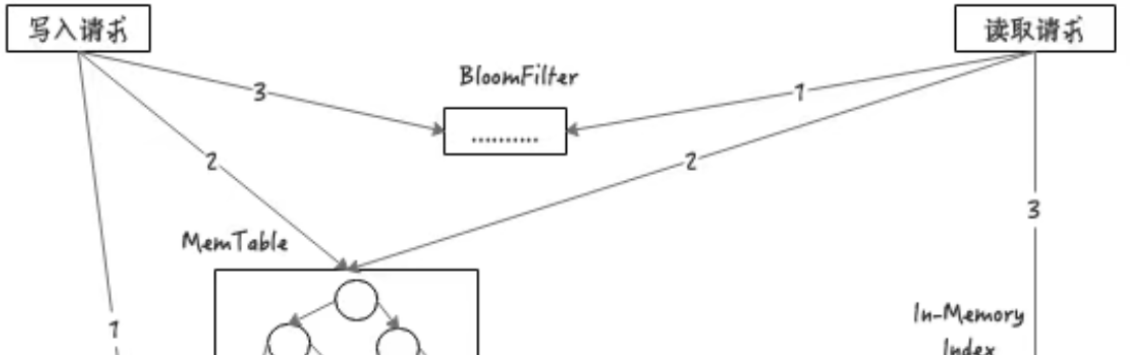
原创 贺卓凡 ImportSource 2020-11-01

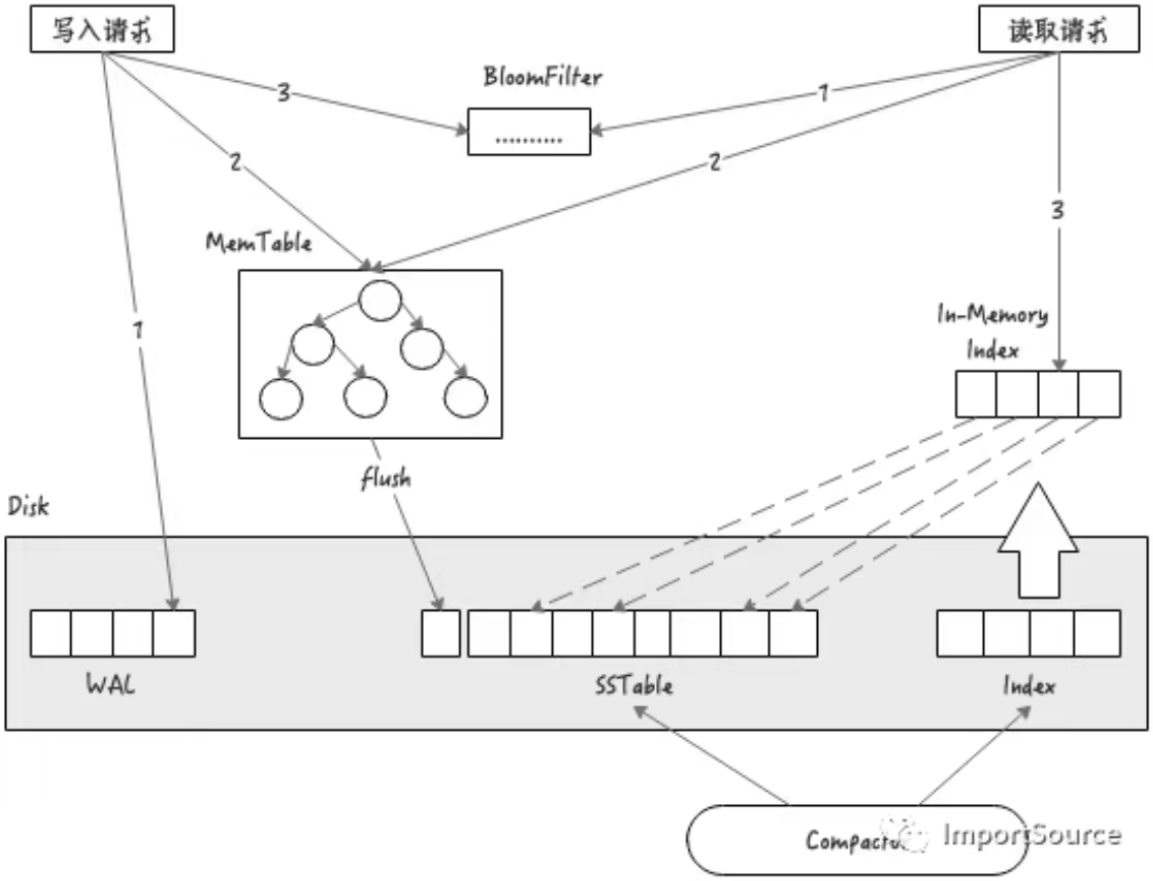
收录于话题

#LSM 1 #分布式数据库 1 #Bloom Filter 1

#布隆过滤器 1 #高吞吐 1

很多的数据库现在都在使用LSM tree作为其核心结构，因为它可以提供非常高的写入吞吐量。一些分布式数据库比如Bigtable、HBase、LevelDB、SQLite4、Tarantool、RocksDB、WiredTiger（MongoDB新一代的引擎）、Apache Cassandra、InfluxDB、ScyllaDB、OceanBase，甚至一些MQ都在使用这个结构，比如pulsar。既然LSM tree这么厉害，我们就有必要深入学习一下。一个典型的基于LSM tree的系统应该包括如下组件。没错，通过本文你将了解到一个典型的LSM系统应该有的五脏六腑。






六大组件

WAL

MemTable

SSTable (Sorted Strings Table)

Compactor

Index

Bloom Filter

**WAL**

LSM之所以能提供那么高的写入吞吐量，一个很重要的原因就是它的每次写请求实际上都仅仅是写入在内存的

（in-memory）,而不是像传统的基于B-Tree实现的存储系统那样一次更新到磁盘的动作会触发对索引的更新，从而使得写入成本非常昂贵。那么你也许会问了，LSM究竟是如何完成持久化的呢？这时候就需要介绍一下WAL了。

WAL就是write-ahead log。这个write-ahead log从字面意思就是“写入之前的日志”，就是在内存中的数据落盘之前的操作日志。在正式数据落盘之前，每次写入请求过来都会在WAL文件中先写入一条日志。这个有什么用呢？一个最重要的用途就是当写入的过程中系统挂了，此时你的数据还能恢复，因为你有WAL。你可以通过WAL来恢复那些当时还在内存没来得及写入磁盘的数据。

此时你也许又会问了，那么为什么不直接把数据写入磁盘呢？这不是脱裤子放屁多费手续吗？为什么还要先写入WAL呢？因为WAL的写入成本非常低，它只是在尾部追加（append-only），而不需要对磁盘上的数据做任何的重新组织和调整（reconstruction）。为什么append就成本低，因为不是随机写。

**MemTable**

接下来是MemTable。在LSM系统中一个不可缺少的组

件就是MemTable。MemTable其实就是某种基于内存的数据结构，这里起个名字叫MemTable。MemTable有多种实现，简单起见，这里你就姑且把它想象成一种类似二分查找树的数据结构。

好，现在的每次写请求，数据都是先追加到WAL，然后再把数据更新到MemTable，然后就向client返回一个成功的响应，一次写入结束。这里你可以想象用java如何实现一个MemTable，通常大家的实现都是通过direct memory来实现，而不是直接写入到堆内存（heap memory），因为这样可以避免GC。

**SSTable (Sorted Strings Table)**

现在我们把数据写入到WAL和MemTable了，但你也许发现，这个MemTable不可能无限制地写入啊，一直写下去RAM就被写满了，我们得定期把MemTable的数据刷到磁盘上，这时候就引入了一个新的组件：SSTable。SSTable，见名知意，就是一个被排序了的String Table，这个SSTable是被存储在磁盘上的。为什么它必须是排序的，因为这样读请求就可以轻松的检索了。

文到如此，你其实已经知晓了LSM高吞吐量的原因了，就是因为使用了WAL、MemTable、SSTable。而且，

每次删除请求也都会在MemTable中添加一条带有删除标记的记录，同样的数据也会刷到SSTable。

**Compactor**

由于我们定期会把数据通过SSTable的方式刷到磁盘，那么相同的key就有可能会在多个SSTables中出现，尽管某个key最晚的数据记录大概率会出现在最近的SSTable中，但依然有必要把所有出现在之前的SSTable中的重复数据清理掉。这就是Compactor的作用，它默默地运行在后台，把重复的key以及已被删除的key合并，然后重新创建一个个新的被压缩和合并了的SSTable。Compactor同时还要负责更新索引（典型的就是一个基于B-Tree的索引）以使得你能够定位到key所在的SSTable（注意：定位到SSTable就够了）。

**Index**

索引的目的就是用来定位到key所在的SSTable，一旦定位到SSTable就容易了，此时找到SSTable中实际的key就非常容易了，因为SSTable是排过序的，此时在内存中使用二分查找就可以轻松解决了。同样，通常设置的SSTables的大小与操作系统页面大小相对应（通常是磁盘bock大小的倍数），这样就可以更轻松地将数据更快地加载到内存中。

尽管索引可以帮助我们定位到SSTable，然后快速查到对应的key，但所有的读请求依然是首先去MemTable中的查找，因为最近的更新都在MemTable中，只有当MemTable中没找到，才会使用到索引去定位key所在的SSTable，定位到SSTable后，再在内存中使用二分查找找到对应的key。

由于每次读取请求都会检查MemTable、Index和SSTable，这样就会使得那些不存在的key的查询变得非常地昂贵，你想想，那个key根本就不存在，你还要挨个去MemTable、Index、SSTable中找一遍，多傻。为此，是时候引入一个新的组件解决这些问题了。那就是Bloom Filter，没错，布隆过滤器。

**Bloom Filter**

布隆过滤器被用在了很多的分布式存储系统中，你能想到的分布式存储系统几乎都用到了这个组件。布隆过滤器是一个概率型数据结构。它可以帮助你
在一个更高层次（相对于数据存储来说）就检查到一个key是否存在于系统中，在内存中的时间复杂度是O(1)。我们在之前文章中已经介绍过布隆过滤器，这里就不赘述。它有个核心能力就是它检查到一个key存在，那么这个key有可能存在于系统中，也有可能不存在于系统；但如果它检查到一个key不存在，那么这个

key就肯定不在系统中。正因为它的这个能力，在很多的分布式存储系统中都使用布隆过滤器来过滤key不存在的场景。比如本文描述的场景，当一个key不存在系统中时，
就可以避免上面昂贵的读取路径（MemTable、Index、SSTable）。但这仅仅能解决当key不存在的场景，如果使用布隆过滤器查到某个key存在时，依然要按照上面的路径（MemTable、Index、SSTable）挨个检查去读取数据。

**总结**

通过本文你已经了解LSM架构的各个组件。以后有人向你宣称他们的数据库或某个中间件有很高的写入吞吐，那么你可以断定他大概率使用了LSM架构，或者类LSM的架构，而且你还知道其中的具体场景下读取性能可能一般。因为你曾经通过本文掌握了未来数十年甚至更多年流行的又一个数据存储架构：LSM。

import Src
贺卓凡