海量数据存储最佳实践

一、存储系统的技术选型

存储系统的一个特点是繁杂，可供选择的产品非常多。排除那些小众或目前还不太成熟的产品，真正广泛应用于生产系统的、可供选择的存储系统，仍然非常多。每种存储产品都有其擅长之处，有其适用的业务场景，当然也有各自的短板。

如果所选择的存储系统不能很好地匹配业务，那么不仅是开发的时候会很别扭、不顺畅，数据量稍大时，还可能会出现性能严重下降的问题，甚至出现存储慢到卡死以致不可用的情况。反过来，如果选择的是合适的存储系统，就会让你在构建和运营系统的时候感觉顺畅很多。

选择什么样的存储系统来保存数据，对系统的性能和稳定性来说都是非常重要的。要牢记一句话，在实际的业务开发中，我们所做的所有事情总结起来两件：存储和计算。选定了存储系统，往往决定了系统的上限；计算方式决定了能否发挥这个上限以及系统的下限。比如选择了MySQL数据库，决定了单机能达到的QPS在千级，业务中不适当的表的设计和SQL的编写，却只能达到一、两百甚至几十的QPS。

那么，我们应该如何根据业务的特点，选择合适的存储来构建系统呢？我们把互联网大厂存储方面相关的内容做一个提炼，总结一下，看看我们如何做存储系统的技术选型。

技术选型时应该考虑哪些因素

我们需要根据业务的特点来选择合适的存储系统，是否有一些具体的、可操作的方法，让我们做参照呢？肯定是有的，我们来看看大厂多年的实践中总结出来的经验。

首先需要确定的是系统的类型

是一个在线业务系统，还是一个分析系统？在线业务系统对应的术语是OLTP (On-line Transaction Processing，在线事务处理)，分析系统对应的术语是OLAP(Off-line Analytical Processing，离线分析处理)。

由于这两种不同类型的系统对存储系统的要求完全不一样，因此在做存储技术选型的时候，需要先确定到底是哪种系统。

但现实情况是，大部分系统很难明确地归类为在线业务系统或是分析系统。比如，电商系统既包括在线业务部分，又要满足做报表分析的需求，像这种情况又该如何划分呢?

答案是主要取决于系统的规模，如果系统的规模不大，那么我们需要确认的是，系统的主要业务是在线业务还是分析业务，然后以这个主要业务作为划分的依据。

比如，创业公司的电商系统，其主要业务一定是在线交易部分，那就按照在线业务系统来处理。如果系统的规模足够大，那就把系统划分为在线业务系统和分析系统两个部分，每个部分分别选择合适的存储，当然这样的架构成本会比较高，只有规模足够大才值得这样拆分。

第二个需要考量的维度是数据量

系统需要处理的数据在什么量级？这里的数据量不需要特别精确，能估计到量级就可以了。在估算系统数据量级的时候，需要考虑存量数据和增量数据两个部分，简单地说就是，现在有多少数据,未来还会新增多少数据。在估计系统数据量的时候，不必对未来做过多的预留，一般来说按照未来两年，最多三年来估计就足够了。

不用担心因为预估不足，而无法支撑两年之后业务的问题。一般来讲，很少有新系统在上线之后两三年内，业务没有发生重大改变的。既然系统在不到两三年的时间内就要进行重构，那么存储只需要在重构时，也跟着进行相应的调整即可。

退一步来讲，即使系统在两三年内没有进行重构，那么之前预估的两三年的数据量再撑个三四年问题也不大。因为大部分人在对未来业务和数据量做预估的时候，往往都会过于乐观。也就是说，两年后系统的实际数据量大概率要远少于两年前预估的数据量。

自然我们考量系统的数据量，就是系统现在的数据存量加上未来二到三年的数据量。然后再来看一下，我们预估的数据量级在下面哪个范围内。

1） 1GB以下量级，或者数据的条数在千万以下。对于这个量级内的数据来说，几乎所有的存储产品其性能都还不错，因此不需要过多考虑数据量和性能，重点考虑其他维度即可。

2） 1GB 以上、10GB 以下量级，或者数据的条数在一亿以内。这个量级基本上是单机存储系统能够处理的上限。

3）超过10GB量级，或者数据的条数超过一亿。这个量级的数据必须使用分布式存储，只有将数据分片，才能获得可以接受的性能。

第三个需要考虑的维度，非常重要，但也最容易被忽略，那就是总体拥有成本(Total Cost of Ownership，TOC)。

总体拥有成本是指，选择该存储产品，所需要付出的成本。虽然现在大部分存储系统都是开源免费的，但是无论使用哪一款产品，都是有成本的。成本主要来自如下三个方面。

第一，也是最重要的，团队是否熟悉该产品？如果不熟悉则意味着使用过程中可能要踩坑，然后填坑。踩坑和填坑的代价可能是系统宕机、丢数据或者开发进度延期。

第二，需要考虑该产品是否简单、易于学习和使用。

第三，需要考虑系统上线后的运维成本，比如，Hadoop生态的一系列产品,维护工作相对来说就比较困难，要想让它们持续正常地运转，一般都需要一个有经验的运维人员专门负责维护。

在线业务系统如何选择存储产品

首先来看一下在线业务系统如何选择存储产品。在线业务系统是指为在线业务提供服务的系统，比如，电商系统的交易部分，或者手机上使用的绝大多数App，直接支撑这些App的后端系统，都是在线业务系统。通俗地说就是，那些主要对数据库执行增删改查操作的系统，都是在线业务系统。

那么在线业务系统对存储产品有什么样的要求呢?

1）由于需要频繁地对数据进行增删改的操作，因此存储产品需要有较好的写性能。

2）由于在线业务直接服务于前端，需要快速响应，因此每次存储访问必须要快，至少要达到毫秒级的响应。

3）另外，存储产品需要能够支撑足够多的并发请求，以满足大量用户同时访问的需求。

4）最后，很重要的一点是，由于在线业务系统的需求一直都在不停地变化，因此存储产品需要能够提供相对比较强大的查询能力，以便应对频繁变化的需求。否则，一旦业务需求稍微有一点儿变动，存储结构就不得不随之做出调整，这就非常麻烦了。

那么哪些存储产品可以满足上面列出的这些需求，支撑在线业务呢？答案是，没有这么完美的产品。

但是如果把要求放宽一点，最接近上述要求的就是我们最常用的以MySQL为代表的关系型数据库。除了我们常用的MySOL 之外，Oracle、DB2、SOL Server，以及各大云厂商提供的RDS等都是关系型数据库。

由于各关系型数据库产品的存储结构和实现原理都是类似的，因此它们在功能等方面的差别并不大，是可以相互替代的。考虑到成本等诸方面的因素，MySOL 一般是我们的首选。

此外，一些KV存储也可以用于在线业务，比如，Redis、Memcached等等。Redis这种基于内存的存储，具有非常好的读写性能，能提供有限的查询功能，但是其并不能保证数据的可靠性，一般来说,Redis都是配合MySQL 数据库作为缓存来使用。所以，目前绝大多数的在线业务，仍然使用的是 MySQL(或者其他关系型数据库)加 Redis这对经典组合，暂时还没有更好的选择。

还有一些存储产品也可以用于在线业务，但大多数局限于特定的业务场景中,不具备通用性,比如,用于存储文档型数据的 MongoDB，等等。

在线业务系统需要存储产品能够支持高性能写入、毫秒级响应以及高并发。MvSOL加 Redis的经典组合可以应对大部分的场景需求。而分析系统则需要存储产品能够促在海量数据，并且能够支持在海量数据上快速聚合、分析和查询，而对写入性能、响应时延和并发的要求并不高。

一般来说量级在GB以内的可以使田MySQL ； 量级超过GB的数据并且如果还是需要做实时的分析和查询，则可以优先考虑ES，Hbase、Cassandra和ClickHouse这些列式数据库也可以视情况选择。量级超过 TB的数据，一般只能事先对数据做聚合计算，然后再在聚合计算的结果上进行实时查询，这种情况下，一般选择把数据保存在 HDFS 中。

常见问题：如何存储前端埋点之类的海量数据

对于大部分互联网公司来说，数据量最大的几类数据是：前端埋点数据、监控数据和日志数据。“前端埋点数据”也称为“点击流”，是指在App、小程序和 Web页面上的埋点数据，这些埋点数据主要用于记录用户的行为，比如打开了哪个页面，点击了哪个按钮，在哪个商品上停留了多久等信息。

这种记录用户的行为数据，为了从统计学上分析群体用户的行为，从而改进产品和运营。比如，浏览某件商品的人很多、在其上停留的时间也很长，最后下单购买的人却很少，那么采销人员就要考虑这件商品的定价是不是太高了。

除了点击流数据之外，监控和日志也是大家常用的数据。

上述三种数据都是真正的“海量”数据，相比于订单、商品之类的业务数据，点击流的数据量要多出2～3个数量级。在头部大厂，这类数据每天产生的数据量很有可能会超过 TB(1 TB = 1024 GB)级别，经过一段时间的累积，有些数据会达到PB(1 PB =1024 TB）级别。

早期对于海量原始数据的存储方案，都倾向于先计算再存储。也就是说，在接收原始数据的服务中，先对数据进行过滤和聚合等初步的计算，将数据压缩收敛之后再进行存储。这样可以降低存储系统的写入压力，同时还能节省磁盘空间。

随着存储设备的成本越来越低，以及数据的价值被不断地重新挖掘，很多大型企业都倾向于先存储再计算。即直接保存海量的原始数据，再对数据进行实时或批量计算。这种方案成本较高但是优点很多，比如不需要二次分发，就可以同时为多个流和批计算任务提供数据；如果计算任务出现错误，则可以随时回滚，重新计算；如果对数据有新的分析需求，则上线之后，可以直接用历史数据计算出结果，而不用等待收集新的数据。

不过先存储再计算的方式，对保存原始数据的存储系统提出了更高的要求：不仅要有足够大的容量，能够水平扩容，而且要求读写速度足够快，要能跟得上数据生产的速度，同时环要为下游计算提供低延迟的读服务。一般都会选择Kafka/RocketMQ来存储。

这类产品能够提供“无限”的消息堆积能力，具有招高的吞吐量，与大部分大数据生态圈的开源软件都有非常好的兼容性和集成度。

如果是需要长时间(几个月到几年)保存的海量数据，适合用HDFS之类的分布式文件系统来存储。

还有一类是时序数据库(Time Series Databases)，比如InfluxDB，不仅具有非常好的读写性能,还能提供简便的查询和聚合数据的能力。但是它们并不能存储所有类型的数据，而是专用于存储类似于监控之类的有时间特征，并且数据内容都是数值的数据。

二、面对海量数据,如何才能查得更快

前面说明了如何保存海量的原始数据，因为原始数据的数据量实在是太大了,能够存储下来已属不易，这个数据量是无法直接提供给业务系统进行查询和分析的。

其中有两个方面的原因：一是数据量太大了，二是目前没有很好的数据结构和查询能力可以支持业务系统的查询。

所以，目前的一般做法是，通过流计算或批计算（也就是 MapReduce)对原始数批进行二次或多次讨滤、汇聚和计算的处理,然后把计算结果保存到另外一个存储系统中由该存储系统为业务系统提供查询支持。

有的业务计算后的数据变得非常少，比如，一些按天进行汇总的数据，或者排行榜类的数据，无论使用哪种存储，都能满足要求。还有一些业务，无法通过事先计算的方式解决所有的问题。比如，像点击流、监控和日志之类的原始数据，就属于“海量数据中的海量数据”，这些原始数据经过过滤汇总和计算之后，在大多数情况下，数据量会出现数量级的下降，比如，从TB级别的数据量，下降到GB级别，但仍然属于海量数据。除此之外，我们还要对这个海量数据，提供性能可以接受的查询服务。

面对这种数量级的海量数据，如何才能让查询变得更快一些?

常用的分析类系统应该如何选择存储

查询海量数据的系统大多是离线分析类系统，可以简单地将其理解为类似于做报表的系统，也就是那些主要功能是对数据做统计分析的系统。这类系统大多是重度依赖于存储的。选择什么样的存储系统、使用什么样的数据结构来存储数据，将直接决定数据查询、聚合和分析的性能。

分析类系统对存储的需求一般包含如下四点。

1）用于分析的数据量一般会比在线业务的数据量高出几个数量级，这就要求存储系统能够保存海量数据。

2）并且还要能在海量数据上快速进行聚合、分析和查询的操作。注意，这里所说的“快速”，前提是处理GB、TB甚至PB级别的海量数据，在这么大的数据量上做分析，几十秒甚至几分钟都算是快速的了,这一点与在线业务要求的毫秒级速度是不一样的。

3）由于在大多数情况下，数据都是异步写入，因此系统对于写入性能和响应时延，要求一般不高。

4）由于分析类系统不用直接支撑前端业务，因此也不要求高并发。接下来我们看一下，可供选择的存储产品有哪些。如果系统的数据量在GB量级以下，那么MySOL依然是可以考虑的，因为它的查询能力足以应付大部分分析系统的业务需求。而且可以与在线业务系统合用一个数据库，不用做ETL(数据抽取),更简便而且实时性更好。当然最好能为分析系统配置单独的MySQL 实例，以避免影响在线业务。

如果数据量级已经超过了MySQL的极限，则还可以选择一些列式数据库，比如 Hbase、Cassandra、ClickHouse 等。这些产品对海量数据都有非常好的查询性能，在正确使用的前提下，10GB量级的数据查询基本上可以做到秒级返回。不过，这些数据库对数据的组织方式会有一些限制，在查询方式上也没有MySQL那么灵活。

还可以考虑Elasticsearch (ES)，ES本来是一个为了搜索而生的存储产品，，但是其也支持结构化数据的存储和查询，也支持分布式并行查询，因此其在海量结构化数据查询方面的性能也非常好。最重要的是ES对数据组织方式和查询方式的限制,不像其他列式数据库那么死板。也就是说，ES的查询能力和灵活性是要强于上述这些列式数据库的。不,ES也有一个缺点，那就是需要具有大内存的服务器，硬件成本比较高。

当数据量级超过TB 级的时候，对这么大量级的数据做统计分析,无论使用哪种存储系统，速度都快不了，这里的性能瓶颈主要在于磁盘IO和网络带宽，这种情况下肯定做不了实时的查询和分析，这里可以采用的解决方案具定期对数据讲行聚合和计算，然后把结果保存起来，在需要时再对结果做一次查询。这么大量级的数据，一般是选择存储在HDFS中，配合Spark、 Hive等大数据生态圈产品，对数据进行聚合和计算。

转变思想：多层缓存共同构建存储体系

我们虽然有众多的存储系统可以选择，但是这些不同的存储系统之间其实并没有本质的差别。他们最核心的区别只在于，存储引擎的数据结构、存储集群的构建方式，以及提供的查询能力等这些方面的差异。这些差异，使得不同的存储系统只有在他所擅长的那些领域或场景下，才会有很好的性能表现。

也就是说，没有哪种存储能够在所有情况下，都具有明显的性能优势。所以说，存储系统没有银弹，不要指望简单地更换一种数据库就可以同时解决数据量大、查询慢的问题。

但是我们其实可以换一种思路，利用不同存储的特点，各取所长，将多个存储引擎组合到一起，共同构建存储体系。例如，在电商项目中，对于并发要求更高的秒杀场景，会在传统存储系统的基础上，引入RocksDB组件，用来在集中存储之外，另外构建一套服务本地的缓存数据，用来加速数据存储读取速度。后续我们就可以将RocksDB和ES、MySQL这类集中存储一起，共同构建复杂的存储系统体系。

对于秒杀系统的具体实现，我们会在后续的章节进行整体的设计，但是在这里，我们可以先了解一下RocksDB存储的特性。

三、RocksDB 详解

RocksDB是Facebook开源的一个高性能、持久化的KV存储引擎，最初是Facebook的数据库工程师团队基于Google LevelDB开发。一般来说我们很少见到过哪个项目直接使用RocksDB来保存数据，即使未来大概也不会像Redis那样被业务系统直接使用。

但是越来越多的新生代数据库都选择RocksDB作为它们的存储引擎。比如：

CockroachDB（中文名蟑螂，一个开源、可伸缩、跨地域复制且兼容事务的 ACID 特性的分布式数据库，思路源自 Google 的全球性分布式数据库 Spanner。其理念是将数据分布在多数据中心的多台服务器上）

YugabyteDB，Tidb作为CockroachDB的竞争产品，底层也是RocksDB

开源项目MyRocks使用RocksDB给MySQL做存储引擎，目的是取代现有的InnoDB存储引擎。MySQL的亲兄弟MariaDB也已经接纳了MyRocks，作为它的存储引擎。

实时计算引擎Flink，其State就是一个KV存储,它使用的也是RocksDB。

此外包括MongoDB、Cassandra、Hbase等在内的很多数据库,都在开发基于RocksDB的存储引擎。

这其中的主要原因就是RocksDB高性能、支持事务。按照RocksDB主页https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks的说法：

随机读最高可以达到19W/S，平均水平13W/S，覆盖操作可以达到9W/S，多读单写的情况下在10W/S左右。

根据InfluxDB的测试：https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/

批量写入5000万数据，RocksDB只花了1m26.9s，在几个对照引擎中（LevelDB与RocksDB与HyperLevelDB与LMDB），RocksDB读取和删除方面表现最好，InfluxDB也认为根据测试用例，RocksDB作为存储引擎是个非常好的选择

> Overall it looks like RocksDB might be the best choice for our use case. However, there are lies, damn lies, and benchmarks. Things can change drastically based on hardware configuration and settings on the storage engines.

在后续秒杀系统中，我们也会借鉴这些存储产品的思想，引入RocksDB，构建一套定制化的相对比较全面的存储体系。

RocksDB VS Redis

说到KV存储,我们最熟悉的就是Redis了。RocksDB和 Redis都是KV存储。其实Redis和 RocksDB之间并没有可比性，一个是缓存，一个是数据库存储引擎。

Redis是一个内存数据库，它的性能非常好，主要原因之一是它的数据全都保存在内存中。按照 Redis官方网站提供的测试数据来看，它的随机读写性能大约为50万次/秒，当然我们普遍的认可的说法是10W/S。从我们上面的文字可以看到RocksDB相应的随机读写性能大约为20万次/秒，虽然其性能还不如 Redis，但是已经可以算是在同一个量级水平了，毕竟一个是内存操作，另一个是磁盘IO操作。

但是我们也知道，Redis只是一个内存数据库，并不是一个可靠的存储引擎。在 Redis 中，数据写到内存中就算成功了，其并不能保证将数据安全地保存到磁盘上。而 RocksDB则是一个持久化的KV存储引擎，它需要保证每条数据都已安全地写到磁盘上。这种情况下，RocksDB的优势就很明显，磁盘的读写性能与内存的读写性能本就相差了一两个数量级，读写磁盘的RocksDB能达到与读写内存的Redis相近的性能这就是RocksDB的价值所在了。

一个存储系统的读写性能主要取决于它的存储结构，也就是数据是如何组织的。RocksDB采用了一个非常复杂的数据存储结构，并且这个存储结构采用了内存和磁盘混合存储的方式，它使用磁盘来保证数据的可靠存储的,并且会利用速度更快的内存来提升读写性能。

RocksDB为什么能实现这么高的写入性能呢？大多数存储系统为了能够实现快速查找都会采用树或哈希表之类的存储结构。数据在写入的时候必须写到特定的位置上。比如,我们在向B+树中写入一条数据时，必须按照B+树的排序方式,写到某个固定的节点下面。哈希表也与之类似，必须要写到特定的哈希槽中。

这样的数据结构会导致在写入数据的时候,不得不先在磁盘的这里写一部分，再到那里写一部分这样跳来跳去地写，即我们所说的“随机写”。MySQL为了减少随机读写，下了不少功夫。而RocksDB的数据结构，可以保证写入磁盘的绝大多数操作都是顺序写入的。

Kafka所采用的也是顺序读写的方式，所以读写性能也非常好。凡事有利也有弊，这种数据基本上是没法查询的因为数据没有结构,只能采用遍历的方式。

RocksDB究竟如何在保证数据顺序写入的前提下，还能兼顾很好的查询性能呢?其实使用了数据结构LSM-Tree。

LSM-Tree 如何兼顾读写性能

LSM-Tree的全称是The Log-Structured Merge-Tree，是一种非常复杂的复合数据结构。它包含了WAL (Write Ahead Log)、跳表(SkipList)和一个分层的有序表(Sorted String Table, SSTable)，LSM-tree专门为 key-value 存储系统设计的，以牺牲部分读取性能为代价提高写入性能，通常适合于写多读少的场景。

我们借用下论文“An efficient design and implementation of LSM-tree based key-value store on open-channel SSD”（pdf链接https://cadlab.cs.ucla.edu/beta/cadlab/sites/default/files/publications/eurosys2014_final44.pdf，本论文主要讲述在SSD的高并行性下，如何扩展LevelDB以显式利用SSD的多个通道，优化并发I / O请求的调度和调度策略，将常规SSD上运行LevelDB的吞吐量再提高4倍以上）的图

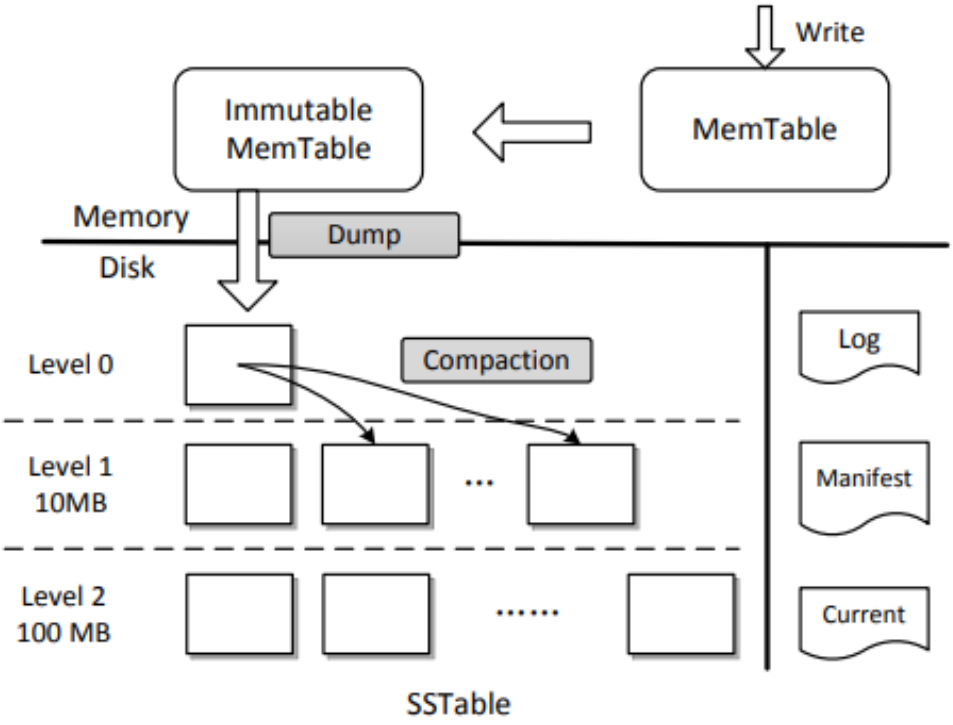

Figure 1. Illustration of the LevelDB architecture.

当然LSM-Tree的结构比上面这个图其实复杂。我们只需要关心核心原理即可。

图中有一个横向的实线，这是内存和磁盘的分界线，实线上面的部分是内存，下面的部分是磁盘。

数据的写入过程。当LSM-Tree收到一个写请求时，比如“PUT foo bar”，即把 Key foo的值设置为bar，这条操作命令会被写入磁盘的 WAL日志中，图中右侧的Log，这是一个顺序写磁盘的操作，性能很好。这个日志的唯一作用就是从故障中恢复系统数据，一旦系统宕机，就可以根据日志把内存中还没有来得及写入磁盘的数据恢复出来。

写完日志之后，数据可靠性的问题就解决了。然后数据会被写入内存的MemTable中，这个 MemTable就是一个按照Key组织的跳表(SkipList)，跳表的查找性能与平衡树类似，但实现起来更简单一些。写MemTable是一项内存操作，速度也非常快。

数据写入MemTable之后就可以返回写入成功的信息了。不过LSM-Tree在处理写入数据的过程中会直接将Key写入 MemTable，而不会预先查看MemTable中是否已经存在该Key。

很明显MemTable不能无限制地写入内存，一是内存的容量毕竟有限，另外，MemTable太大会导致读写性能下降。所以MemTable有一个固定的上限大小,一般是32MB。

MemTable写满之后，就会转换成Immutable(不可变的)MemTable，然后再创建一个空的MemTable继续写。这个Immutable MemTable是只读的MemTable，它与MemTable的数据结构完全一样唯一的区别就是不允许再写入了。

Immutable MemTable也不能在内存中无限地占地方，而是会有一个后台线程，不停地把 Immutable MemTable复制到磁盘文件中，然后释放内存空间。每个Immutable MemTable对应于一个磁盘文件，MemTable的数据结构跳表本身就是一个有序表，写入文件的数据结构也是按照Key来排序的，这些文件就是SSTable。由于把 MemTable写入 SSTable的这个写操作,是把整块内存写入整个文件中，因此该操作同样也是一个顺序写操作。

虽然数据已经保存到磁盘上了，而且这些SSTable文件中的Key是有序的,但是文件之间却是完全无序的，所以还是无法查找。SSTable采用了一个分层合并机制来解决这个问题。SSTable被分为很多层，每一层的容量都有一个固定的上限。

一般来说，下一层的容量是上一层的10倍。当某一层写满时，就会触发后台线程往下一层合并，数据合并到下一层之后，本层的SSTable 文件就可以删除了。合并的过程也是排序的过程，除了Level 0以外，每层中的文件都是有序的,文件内的KV也是有序的，这样就比较便于查找了。

LSM-Tree查找的过程也是分层查找，先在内存中的 MemTable和 Immutable MemTable中查找，然后再按照顺序依次在磁盘的每层SSTable文件中查找，一旦找到了就直接返回。看起来这样的查找方式很低效的，可能需要多次查找内存和多个文件才能找到一个Key，但实际上这样一个分层的结构，它会天然形成一个非常有利于查找的状况，即越是经常被读写到的热数据，它在这个分层结构中就越靠上，对这样的Key查找就越快。

比如最经常读写的Key很大概率会在内存中，这样不用读写磁盘就能完成查找。即使在内存中查不到真正需要穿透很多层SSTable，一直查到最底层的请求,实际也还是很少的。另外在工程实现上，还会做很多优化。

比如RocksDB里，为存储的数据逻辑分族，独特的filter对读优化，使用多个memtable并在immutbale memtable提前进行数据合并的优化，在内存中缓存SSTable文件的Key用布隆过滤器避免无谓的查找等，以加速查找过程；不同的合并算法，SSTFile-Indexing机制，针对sstfile有自己的block cache和table cache。这样综合优化下来最终的性能，尤其是查找会相对较好。

> 有道云笔记链接：https://note.youdao.com/s/ayGDsUbg