字节跳动在 RocksDB 存储引擎上的改进实践

原创 基础架构团队 字节跳动技术团队 2020-03-23

收录于话题
#数据 15 #存储 8 #基础架构 16 #RocksDB 1

> 本文选自“字节跳动基础架构实践”系列文章。
>
> “字节跳动基础架构实践”系列文章是由字节跳动基础架构部门各技术团队及专家倾力打造的技术干货内容，和大家分享团队在基础架构发展和演进过程中的实践经验与教训，与各位技术同学一起交流成长。
>
> RocksDB 是世界上最被广泛使用的存储引擎之一，字节跳动内部大量的数据库产品（如图数据库、NewSQL 等）都构建在 RocksDB 之上，对存储引擎的研发投入将会持续加大，为更多业务赋能。
>
> 本文将介绍字节跳动对 RocksDB 存储引擎的几方面改进，其中参考了大量社区贡献的经验，也有部分独创的技术，希望能为大家带来更多的优化思路。

1. 背景

RocksDB 作为最著名的 LSM 类存储引擎之一，在字节跳动内部占据非常重要的位置，大量的数据库、存储系统都在基于 RocksDB 进行构建或改进，但 LSM 系列众所周知的一些问题同样困扰着字节跳动的业务，包括性能问题、成本问题、功能问题等等。

本文首先尝试梳理和介绍我们对 RocksDB 在五个方面的改进（在内部存储引擎 TerarkDB 的基础上开发），希望能给社区带来一些参考价值，也欢迎对存储引擎感兴趣的技术专家加入我们一起为字节跳动构建更强大的底层支撑。

2. RocksDB 的不足

- 读写放大严重
- 应对突发流量的时候削峰能力不足
- 压缩率有限
- 索引效率较低
- 等等

3. 我们的改进

3.1 LazyBuffer

RocksDB 之前的某个版本引入了 PinnableSlice 作为数据在引擎内的传输载体，它的主要作用是可以**减少数据复制**，即当用户所要查找的数据在 BlockCache 中的时候，只返回其引用。

但是 PinnableSlice 在一些场景下无法发挥价值，比如用户的某个操作需要触碰 **大量不需要的 Value 时**（如 Value 有很多版本或者有大量的 tombstone），PinnableSlice 依然会对这些无用的 Value 产生 I/O 操作，这部分开销是完全可以避免的。

我们为此构建了 LazyBuffer 替换 PinnableSlice，当用户获得 Value 的时候，并不真正进行磁盘 I/O，只有用户真正需要取值的时候才进行真正的 fetch 操作进行 I/O。

Lazy Buffer 是对 PinnableSlice 的增强，从减少数据复制更进一步减少不必要的 IO，对于大量的扫描、SeekRandom 等场景有很大好处。

3.2 Lazy Compaction

自从 LSM 推广开来，针对 LSM Compaction 的各种策略优化层出不穷，其中主流的 Compaction 策略有以下几种：

- Leveled Compaction
  - 全部层级都按照标准的从上到下进行层级合并
  - 读写放大都比较严重，但是空间放大较低
  - 在这篇论文（*Towards Accurate and Fast Evaluation of Multi-Stage Log-Structured Designs*）中有详细的阐述
- Tiered Compaction
  - **即 RocksDB 中的 Universal Compaction**
  - 空间放大和读放大严重，但是写放大最小
  - 在这篇论文（*Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging*）有详细的阐述
- Tiered+Leveled Compaction
  - **即 RocksDB 中的 Level Compaction**

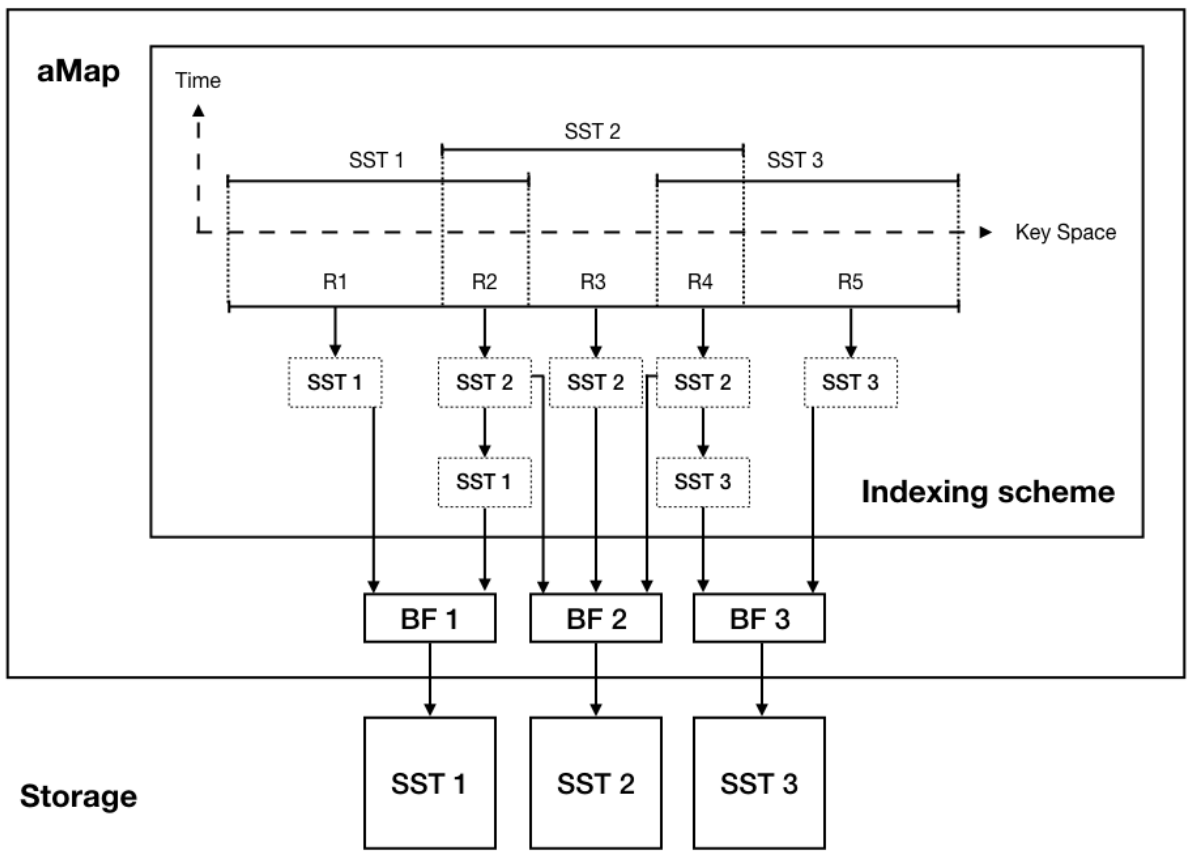
- 是一个混合的 Compaction 策略，空间放大比 Tiered Compaction 更低，写放大也比 Leveled 低

- Leveled-N Compaction
  - 比 Leveled Compaction 写放大更低，读放大更高
  - 一次合并 N - 1 层到第 N 层

从上面的分类我们可以看到，主流的 Compaction 策略主要 **在不同的合并时机之间进行权衡和选择**，字节跳动在这里使用了稍微改进一点的方式。

首先，我们要理解如果能够允许 SST 可以不必保持强有序，那么就可以让我们收集到更多的统计信息后再真正执行外排序（Compaction），但缺点是会增加一定程度的读放大，对读延迟会有影响，那么有没有办法让增加的读放大可控，甚至几乎不增加读放大呢?

我们尝试构建了一种新的 SST 数据结构（Adaptive Map，简称 aMap），区别于 RocksDB 默认的结构，aMap 是一个逻辑上的虚拟 SST，如下图所示：



图：aMap 结构示意图

图中 SST 1、SST 2、SST 3 是三个物理上的 SST 文件，当需要对他们进行合并的时候，我们先构建虚拟遮罩，对上层暴露逻辑上的合并好的 SST（逻辑上是一个大 SST），同时记录和标记其中的覆盖度、Key Space 等统计信息。

它的主要功能有：

- 大的 Compaction 策略上，继承了 RocksDB 的 Level Compaction（Universal Compaction 也可以支持，看场景需要，默认是 Level Compaction）
- 当需要进行 Compaction 的时候，会首选构建 Adaptive Map，将候选的几个 SST 构成一个逻辑上的新 SST（如上图所示）
- Adaptive Map 中会切分出多个不同的重叠段，R1、R2、R3、R4、R5，这些重叠段的重叠度会被追踪记录
- 后台的 GC 线程会优先选择那些重叠度更好的层进行 GC，通过这种手段，我们可以让 GC 更有效率，即写放大远低于默认的情况

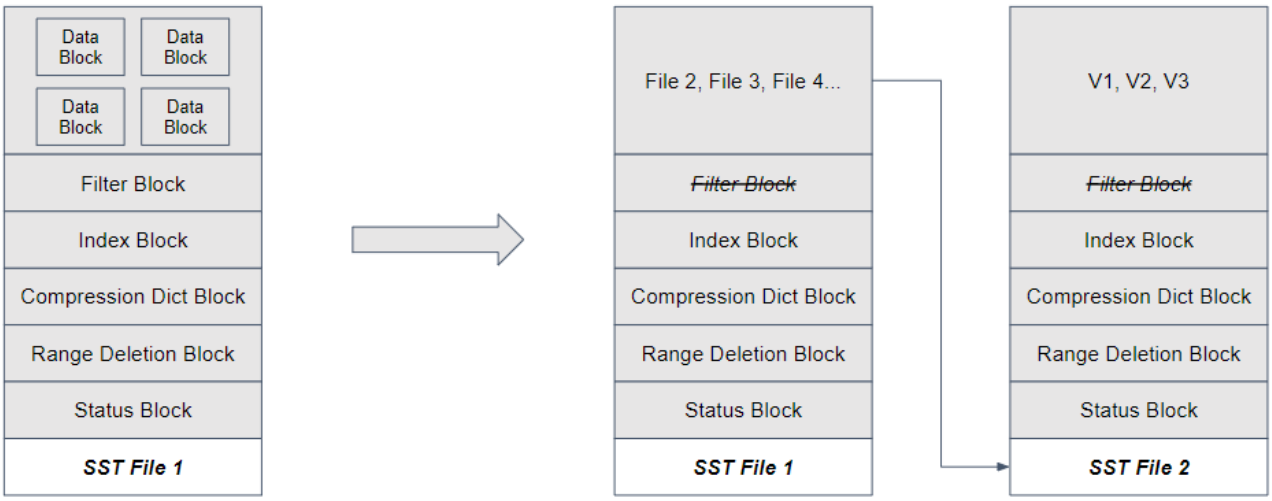
读写放大和原版 RocksDB 对比，理论分析上是有优势的：

I/O 开销	原版 RocksDB	改进后	说明
读放大	O(log N)	O(1)	N 是总数据量
写放大	O(log N)	O(log log N)	N 是总数据量
计算开销	**原版 RocksDB**	**改进后**	
读消耗	O(p)	O(p)	P 是 BloomFilter 的 False Positive Rate
写消耗	O(log N)	O(log log N)	N 是总数据量

表：复杂度分析对比（读放大和写放大）

3.3 KV 分离

在论文 ***WiscKey: Separating Keys from Values in SSD-conscious Storage**** 介绍了一种 KV 分离的 SST 设计，它的主要方式是构建一个 Value Log 文件不断的在上面追加 Value 数据，同时原始的 SST 中 Value 只记录数据真实存在的位置即可。



图：KV 分离的基本原理

其实 KV 分离的思路比较直接和简单，把符合阈值的 value 从直接存储在 SST 中，改为存储文件指针，降低 Compaction、Seek 等操作的开销。

RocksDB 社区有一个 KV 分离的 BlobDB 功能，但是目前功能还不完善，还有大量的工作需要继续做，这里就暂不做对比。另一个 TitanDB 是一个实现上相对完整的 KV 分离存储引擎（以 RocksDB 插件的形式构建），我们在这里对他们进行一个简单的对比：

	WiscKey	TitanDB	TerarkDB (我们的改进)
什么情况下分离	总是	超过阈值	超过阈值
Value 保存方式	VLOG	专门设计的 Blob 文件	原生 SST 文件
Get 流程	1) Key → VLOG Position2) VLOG Position → Value	1) Key → FileNumber + Handle 2) FileNumber → Blob 3) Blob + Handle → Value	1) Key → FileNumber2) FileNumber → SST3) SST + Key → Value
Scan 成本	比 LevelDB 慢支持 Prefetch	比 RocksDB 慢，IO 利用率相对低暂未支持 Prefetch	比 RocksDB 慢，IO 利用率相对低暂未支持 Prefetch
GC 流程	1) 反查并 pop VLOG 头部的 KV2) 有效数据重新写入 VLOG 末尾并重新插入 LSM 树	1) 根据事件监听器挑选垃圾最多的 Blob 发起 GC，反查 Blob 中 KV2) 生成新的 Blob，并将触碰到的数据重新写入 LSM 树	1) 根据事件监听器挑选垃圾最多的 SST 发起 GC，反查 SST 中 KV2) 生成新的 SST
GC 效率	滚动 VLOG 实现 GC 滚动一次完整一次 GC 周期很长，热数据不友好	总是对垃圾最多的 Blob 发起 GC，热数据友好，改写 LSM 树	总是对垃圾最多的 SST 发起 GC，热数据友好，不改写 LSM 树

综合来看，在和社区的对比中，我们实现 KV 分离的大体思路是类似的，但由于我们**有 Adaptive Map 的存在，可以对真正的 GC 操作进行延迟到负载较低的时候进行**，对于应对突发流量尖峰会有相当不错的效果。

但 KV 分离也带来了一些损失，最重要的就是对于范围查询造成了损害，后续可以通过逻辑层进行 Prefetch 来降低这部分的损耗。

3.4 多种索引支持

对于原生的 RocksDB，其 SST 格式大致如下图所示：

```
[data block 1]
[data block 2]
...
[data block N]
[meta block 1: filter block]          (see section: "filter" Meta Block)
[meta block 2: index block]
[meta block 3: compression dictionary block]  (see section: "compression dictionary" Meta B
[meta block 4: range deletion block]      (see section: "range deletion" Meta Block)
[meta block 5: stats block]          (see section: "properties" Meta Block)
...
[meta block K: future extended block]  (we may add more meta blocks in the future)
[metaindex block]
[Footer]               (fixed size; starts at file_size - sizeof(Footer))
```

其中，index block 和 filter block 帮助用户快速定位目标 key 所在的 block。RocksDB 默认的 index 并没有考虑不同数据类型之间的差异，无法根据不同数据类型选择压缩效率和查询效率最高的索引结构，针对这个问题，我们构建了一种自适应的索引选择和构建机制。

- 对于输入的数据，我们会对其进行分段式探测，确定最高效的索引算法
- 对于这批数据进行单独索引并把索引放在 index block 中的

目前已经支持的额外索引算法有：

- 压缩 Trie 索引，针对字符串类型进行通用压缩
- 非降序整数索引，通过 bitmap 构建高度压缩的索引
- ……

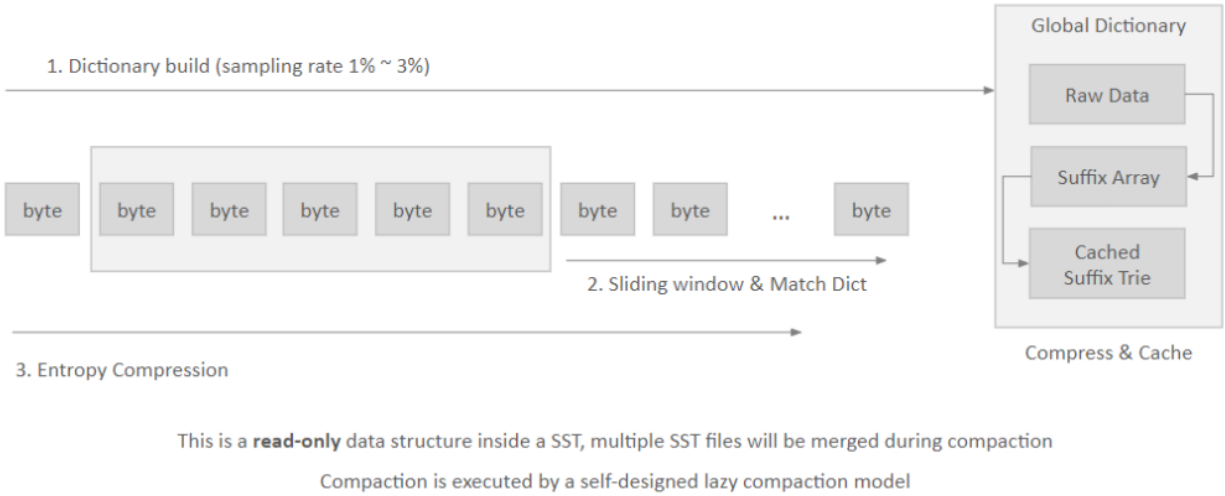
通过多种索引结构的支持，为以后的长期优化提供了更多可能，甚至在 SST 中内嵌 B+ 树索引 data block 等等，更加灵活、模块化的结构让引擎的适应能力更强，面对不同的存储介质、访问模式可以有更佳的综合表现。

3.5 极致压缩

对于数据库应用来说，除了追求高性能外，成本控制也是一个很重要的话题，其中成本控制的重要一环就是数据压缩，对于 LSM 结构来说，默认是通过 block 进行压缩的，压缩率和块的尺寸强相关。

在这里为了解决块尺寸和压缩率的 tradeoff 问题，我们构建了一系列的压缩算法，其中最常用的是可以按记录抽取数据的全局压缩，其基本思路其实并不复杂，通过对 LZ 系列的改进，使用

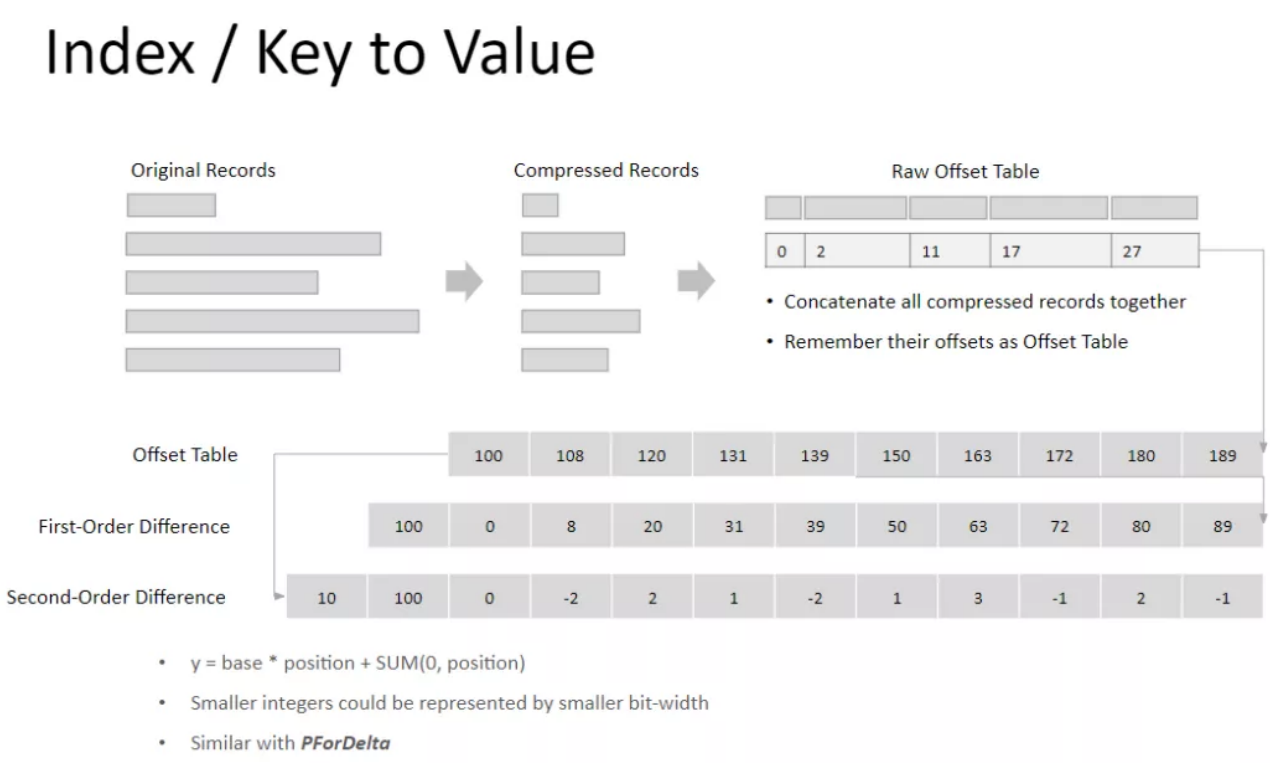
高效的手段保留每一个 Record 的 Offset 即可，而 Offset 表有很多种方法进行压缩存储（显然它是递增的非连续整数序列），利用 pfordelta 等方法进行优化变通存储不难办到。



图：全局压缩算法的概要流程

其大致流程是：

- 先对数据进行扫描采样构建数据字典
  - 所以默认情况下需要对 Compaction SST 进行改造以提供两遍扫描的能力
- 根据数据字典，对原数据进行滑动窗口压缩
- 最后再进行一轮可选的熵压缩（Entropy Compression）
  - 熵压缩业内主流的包括 ANS 和高阶 Huffman 等等，可以根据实际数据分布探测选择
- 在压缩过程中，将会保存每条原始数据的 offset 信息构成偏移表



图：偏移表的构建和保存

偏移表采用常见的类 pfordelta 算法压缩保存即可，需要注意的是因为偏移表会被频繁访问，这里的适宜有一阶差分表和二阶差分表，根据实际情况选择即可。

索引后续可以直接映射到这里具体的 record offset 中来，方便后续的直接按记录寻址。

3.6 新硬件支持

对目前流行的新硬件（如持久化内存、FPGA、QAT 等），我们同样进行了适配和优化，比如在在设备有 QAT 硬件的时候，我们在主机 CPU 负载较高时，选择放弃一部分 CPU 压缩 offload 到 QAT 中进行压缩，再比如我们将一部分数据放在持久化内存上实现真正的 Record 级别数据访问（我们采用的并非常见的块级别的索引结构，而是直接按记录索引）等等。

这里我们以 QAT 压缩为例来说明：

- 我们知道随着 PCIe NVMe SSD 的大范围普及，磁盘的带宽和 IOPS 大幅度提升
- 磁盘带宽的提升进而将系统瓶颈转移到了 CPU
    - CPU 要做的工作包括数据排序（SST 需要维护有序性）、CRC 校验、数据压缩、查询比对等等
- 我们初步目前的计划是把数据压缩和 CRC 校验卸载到专门的 QAT 硬件中进行加速
    - 目前 QAT 硬件的性价比较高，甚至部分主板还自带
- QAT 本身能支持的压缩算法有限，主要以 zlib 的 deflate 为主
- 当我们卸载了数据压缩和 CRC 校验后，就可以分配更多的 CPU 进行 SST 的 GC 和 Compaction，尽快将 SST 形态调整到最佳

目前 QAT 的使用还在测试阶段，还没有正式上线，下一步计划对 FPGA 的应用进行深度的调研。

4. 对比测试

我们使用 RocksDB 的 bench 工具进行了一些简单的测试，对比了 RocksDB、TitanDB 和 TerarkDB 的区别和差异，需要注意是，该工具使用的是随机生成的数据，对于 TerarkDB 的压缩算法不是很友好，**所以压缩率差距并不大**。

这次改进，我们重点关注的是 KV 分离的表现，所以只对比较大的 Value 进行 benchmark 确认其改进效果：

- **测试环境：**
  - 原始测试数据集大小为 256 GB，内存 128 GB
  - CPU : 48 Core
  - RAM : 128 GB
  - Disk : Intel NVMe 3.4T
  - 测试程序为 db_bench
  - Linux version 4.14
  - GCC Version 6.3.0
- **测试内容：**
  1. `fillrandom`： 多线程随机写入, 存在重复 key
  2. `readseq`： 多线程顺序 Scan
  3. `readrandom`： 多线程随机 Get
  4. `seekrandom`： 多线程随机 Seek

Value size = 4KB

```
fillrandom * value size = 4k
Database    ops/sec p50     p99     p99.9
TerarkDB    106273  91.3    289     1213
TitanDB     36918   469.5   579     867
RocksDB     41425   420.6   578     850
-----------------------------------------
readseq * value size = 4k
Database    ops/sec p50     p99     p99.9
TerarkDB    2685885 4.9     16      29
TitanDB     1667512 5.1     36      546
RocksDB     1638499 5.1     36      549
-----------------------------------------
readrandom * value size = 4k
Database    ops/sec p50     p99     p99.9
TerarkDB    489457  29.9    76      107
TitanDB     292591  28.6    168     238
RocksDB     292466  28.5    168     238
-----------------------------------------
seekrandom * value size = 4k
Database    ops/sec p50     p99     p99.9
TerarkDB    476879  31.4    67      75
TitanDB     270933  31.8    169     240
RocksDB     283244  30.3    169     238
```

Value size = 32KB

```
fillrandom * value size = 32k
Database    ops/sec p50     p99     p99.9
TerarkDB    11516   510.0   7961    84450
TitanDB     12326   1079.1  4328    33392
RocksDB     12603   1083.1  4290    36175
----------------------------------------
readseq * value size = 32k
Database    ops/sec p50     p99     p99.9
TerarkDB    717162  20.2    51      353
TitanDB     375866  17.4    758     965
RocksDB     297811  17.5    793     867
----------------------------------------
readrandom * value size = 32k
Database    ops/sec p50     p99     p99.9
TerarkDB    123295  32.7    573     837
TitanDB     197765  43.5    264     370
RocksDB     196081  44.2    267     371
----------------------------------------
seekrandom * value size = 32k
Database    ops/sec p50     p99     p99.9
TerarkDB    497923  30.2    57      74
TitanDB     194832  45.0    253     369
RocksDB     166346  64.5    282     372
```

5. 后续

字节跳动在单机引擎上的投入会持续加大，同时也会考虑为各类特定业务构建针对性的专用引擎，其目标是在单机内为上层业务提供更强大的性能，更灵活的功能和更可靠的服务。

为了实现这些目标，后续我们还需要做的有很多，包括卸载单机引擎的 CPU 到集群上进行分布式 Compaction、引入 SPDK 相关的技术提升 IO 效率、引入 AI Tuning 针对不同负载做更灵活的 I/O 策略、引入新硬件（如持久化内存和 FPGA）等等。

为了实现字节跳动存储引擎的多样性和走向业界前沿，我们迫切的希望有志者能够加入我们一起做新的探索，我们也希望未来在主流的期刊上、开源社区中能够看到字节跳动的活跃身影，为技术社区贡献自己的力量。

6. 参考文献

1. WiscKey: Separating Keys from Values in SSD-conscious Storage
2. Bitcask A Log-Structured Hash Table for Fast Key/Value Data
3. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data
4. Towards Accurate and Fast Evaluation of Multi-Stage Log-Structured Designs

更多分享

深入理解 Linux 内核--jemalloc 引起的 TLB shootdown 及优化

字节跳动自研万亿级图数据库 & 图计算实践

字节跳动 EB 级 HDFS 实践

字节跳动基础架构团队

字节跳动基础架构团队是支撑字节跳动旗下包括抖音、今日头条、西瓜视频、火山小视频在内的多款亿级规模用户产品平稳运行的重要团队，为字节跳动及旗下业务的快速稳定发展提供了保证和推动力。

公司内，基础架构团队主要负责字节跳动私有云建设，管理数以万计服务器规模的集群，负责数万台计算/存储混合部署和在线/离线混合部署，支持若干 EB 海量数据的稳定存储。

文化上，团队积极拥抱开源和创新的软硬件架构。我们长期招聘基础架构方向的同学，具体可参见 **job.bytedance.com** （点击左下角“阅读原文”直达官网），感兴趣可以联系邮

箱 guoxinyu.0372@bytedance.com 。

欢迎关注「字节跳动技术团队」

点击阅读原文，快来加入我们吧！

收录于话题 #数据·15个

上一篇
字节跳动分布式表格存储系统的演进

下一篇
字节跳动 EB 级 HDFS 实践

阅读原文