基于更新热点感知的LSM-Tree查询优化

林清音，陈志广
中山大学计算机学院，广东 广州 510006

摘要
基于LSM-Tree的键值存储已经得到广泛使用。LSM-Tree通过将更新的数据缓存在内存中、随后批量写入磁盘的优化措施取得极高的写性能。然而，在基于LSM-Tree的键值存储中，被更新键值对的旧数据不会立即从存储系统中清除，导致整个存储系统中积累大量的无效数据，最终会显著降低键值存储的读性能。针对以上问题，提出一种更积极的压缩（compaction）方法，通过记录键值对更新的历史信息，识别出更新热点，在整个LSM-Tree存储系统中寻找无效数据大量聚集的SSTable，尽早实施压缩，清除无效数据，缓解写放大效应，从而提升读性能。实验表明，该方法能够降低LevelDB 65.2%的平均读时延、69.4%的99%读尾时延以及71.4%的写放大。

关键词
键值存储；日志结构合并树；读性能优化；写放大

中图分类号：TP392 文献标志码：A doi: 10.11959/j.issn.2096-0271.2022049

A hot-update-aware optimization to the query of LSM-Tree

LIN Qingyin, CHEN Zhiguang
School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou 510006, China

Abstract
Key-value stores based on LSM-Tree have been widely used. LSM-Tree gains excellent write performance by collecting updated data in memory and then flushing data into storage in batches. However, in LSM-Tree-based key-value stores, old data generated by update operations will not be eliminated immediately from the storage system, resulting in a large amount of invalid data accumulated in the entire storage system, which will eventually significantly reduce the read performance of key-value stores. For the above problems, an active compaction method was proposed. By recording the history information of updated key-value pairs, recognizing hot-updated keys, finding SSTables that contain a large amount

of invalid data in the storage system, and triggering compaction as soon as possible to clear much more invalid data, the proposed method could reduce write amplification and improve the read performance of LSM-Tree based key-value stores. Experiments showed that this method could reduce the average read latency of LevelDB by 65.2%, 99% read tail latency by 69.4%, and write amplification by 71.4%.

Key words
key-value stores, log-structured merge tree, read performance optimization, write amplification

0 引言

随着互联网的发展，数据规模与日俱增，大数据时代已经来临。在大数据时代背景下，有海量数据需要存储，如在电子商务、社交网络、网页搜索等场景，每天都需要存储和读取大量的数据。这就要求用一种高性能的数据索引结构来组织海量数据，以便快速存储和读取。日志结构合并树（log-structured merge tree, LSM-Tree）[1]是一种被广泛使用的索引结构，它是一种分层、有序、面向持久化存储的数据索引结构，其核心思想是将数据缓存在内存中再批量写入磁盘。现已有许多成熟的键值存储系统基于LSM-Tree实现，如Google开发的LevelDB[2]、Facebook开发的RocksDB，以及eBay开发的Cassandra[3]等。

LSM-Tree具有高写入性能，但其层次化的数据组织结构使其读性能较差。在各种应用场景下，在关注数据的写入速度时也应该关注数据的读取速度。LSM-Tree的层次结构一方面导致查询过程产生读放大，另一方面也带来非常严重的写放大。由于查询时需要逐层查找，在此过程中需要经过多次磁盘访问才能最终查找到目标数据，因此存在读放大。LSM-Tree通过压缩（compaction）操作来维护其层次结构，需要不断地读出数据进行重新排序和合并，之后再次写入磁盘，因此导致了写放大。写放大不仅占用了存储介质的写带宽，影响了写性能，而且对写放大较为敏感的介质，例如固态硬盘（solid state disk, SSD），使用寿命将被大大缩短。LSM-Tree采用异地更新的模式，即通过追加新版本数据来完成对数据的更新而不是直接修改数据，因此LSM-Tree中包含了较多的旧数据，这些旧数据不会产生读命中，但加重了LSM-Tree的读放大和写放大。因此，关注更新操作对LSM-Tree产生的影响对于提升读性能和减小写放大具有重要意义。

现已有很多针对LSM-Tree的写放大问题的研究。Yao T等人[4]提出了MatrixKV，通过减少LSM-Tree的总层次来减小写放大；Lu L Y等人[5]提出了WiscKey，采用键值分离的方法，使得每次压缩时只需要对键进行维护，从而减小了写放大；Raju P等人[6]提出了PebblesDB，设计了“守卫”来维护每一层的部分有序，降低压缩次数，从而减小写放大；Yao T等人[7]提出了LWC-Tree，通过对有序字符串表（sorted string table, SSTable）追加数据，只对其元数据进行压缩，大大减小了写放大。这些工作为了减小写放大在一定程度上牺牲了读性能，键值分离使得需要二次查找才能读取值，而破坏LSM-Tree原本的完全有序性，不利于数据的查找。另外，虽然针对LSM-Tree的读性能展开优化的相关工作很多[8-10]，但是针对频繁更新场景下的读性能优化的工作却较少。Shin J等人[11]提出了LSM

RUM-Tree，通过在内存中添加更新备忘录来加速R-Tree[12]的查询，进而加速整个LSM-Tree的查询；Chandramouli B等人[13]提出了Faster，设计了一种跨内存和存储介质的混合日志，能够实现高并发的数据读取和数据就地更新。但LSM RUM-Tree没有解决更新带来的旧数据导致的写放大问题，且该工作是针对LSM R-Tree[14]进行优化的；Faster为了实现数据在内存中的就地更新，引入了过多的内存开销。

本文通过实验证明更新操作会显著降低LSM-Tree的读性能，并进一步证明了减少LSM-Tree中因为更新而引入的旧数据对于提升读性能非常重要。因此，本文针对此问题展开优化，提出了一种积极的压缩方法，该方法有3个优点：①能够在压缩过程中彻底清除旧数据，减小压缩带来的写放大；②能够检测存储了大量旧数据的SSTable，并提前对其进行旧数据清除；③加速查询过程。

本文首先介绍LSM-Tree的基本结构及其写入和查询的执行过程，并详细阐述更新操作对其产生的影响；之后对本文为解决旧数据带来的负面影响问题而提出的积极的压缩方法的设计进行详细介绍；最后介绍使用本文方法对键值存储系统进行改进后的优化效果。

1 日志结构合并树

LSM-Tree的主要特点是具有良好的写性能，但其读性能较差。基本的LSM-Tree由两部分组成：一部分是内存中的有序表（memory table，MemTable），另一部分是磁盘中的SSTable。LSM-Tree的核心思想是将写入的数据缓存在内存中，随后批量刷入磁盘，利用磁盘的批量顺序写性能远高于多次随机写性能的特点，达到极高的写入速度。如图1所示，在写入数据时，数据首先写入位于内存的MemTable中，当MemTable写满时，会转化为不可变MemTable。不可变MemTable会被写入磁盘的L_0层生成SSTable，这个过程被称为刷写（flush）。磁盘中的SSTable以层次结构存储，每一层的容量随着层次的加深呈指数级增长。为了使刷写快速完成，L_0层中的不同SSTable是部分有序的，即单个SSTable内部的数据有序，但不同的SSTable之间可能存在数据范围的重叠。除L_0层之外，其余层次中的SSTable都是完全有序的。

1.1 LSM-Tree的写入和查询

LSM-Tree的数据写入是由上到下的，即当某一层写满之后，LSM-Tree才会将该层的SSTable压入下一层。具体地，如图1所示，每一层的容量有限，当L_i层的数据写满后，LSM-Tree会选取L_i层中的SSTable（记为S_i）以及L_{i+1}层中与S_i存在数据范围重叠的SSTable，将这些SSTable中的数据读取到内存中，进行排序重新合并之后写入新的SSTable，统一保存到L_{i+1}层中，此过程被称为压缩。写入的数据随着时间的推移会逐渐被压入底层，而最近写入的数据则保存在较浅的层次。

LSM-Tree的数据查询也是由上到下的。首先在内存里的MemTable和不可变MemTable中查找，然后再查找磁盘中的SSTable。如前面所说，磁盘中的SSTable以层次结构存储，因此需要逐层对SSTable进行查找。L_0层中的SSTable是部分有序的，因此需要遍历该层所有的SSTable，而这种查找逻辑也导致了L_0层不能存放过多的SSTable，否则会大大降

低查找速度。若在L_0层的SSTable中没有查找成功，则需要从上到下逐层查找其他层次的SSTable。其他层次的SSTable是完全有序的，因此只需要进行二分查找即可。总体来说，越底层的数据，其查询速度越慢。

1.2 更新对LSM-Tree的影响

LSM-Tree的数据更新是异地更新。SSTable一旦写入磁盘后就不可更改，因此对数据的更新只能通过在新的SSTable中写入新版本数据来完成。而新版本的数据通常存在于较浅的层次，也会在查找过程中首先被找到，因此其正确性能够得到保证，数据更新也与数据写入一样具有极高的速度。然而，随着数据不断更新，保存在SSTable中的旧数据不再被访问，却占用了存储空间。随着压缩的不断进行，这些旧数据可能被不断读取和重新写入，带来了额外的读写放大。尽管压缩过程可以清除一部分旧数据，但这种清除是不彻底的，因为LSM-Tree不知道每个键的最新版本保存在哪个SSTable中，所以只能清除参与到某一次压缩中的重复键，而被保留下来的键并不一定是最新的。

总而言之，LSM-Tree的压缩过程带来严重的写放大，在查询时需要访问多个SSTable才能查询成功，因此存在读放大，而LSM-Tree中由更新带来的旧数据加重了这些影响。

本文通过实验证明了更新操作确实降低了LSM-Tree的读性能。本文选择LevelDB进行测试。LevelDB是非常经典的基于LSM-Tree的键值存储系统。本文对基于固态硬盘的LevelDB进行了测试，并修改了雅虎云服务测试基准（Yahoo! cloud serving benchmark，YCSB）[15]中工作负载类型的读写比，工作负载U1~U9中更新操作总数呈递增趋势。YCSB工作负载配置见**表1**。

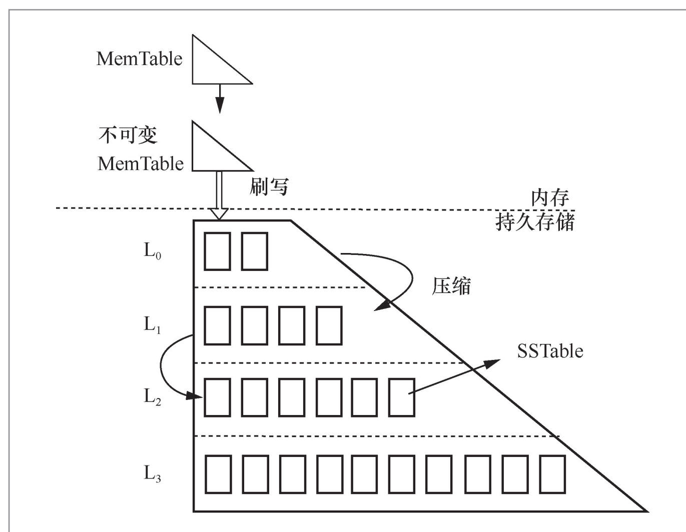


图1 LSM-Tree 基本结构

从**图2**可以看出，对于基于固态硬盘的LevelDB，当更新操作逐渐增加时，平均读时延和99%读尾时延都呈现出了增加趋势。在执行工作负载U9时，LevelDB的平均读时延和99%读尾时延比执行工作负载U1时分别增加了约100%和约309%。该实验结果证明了更新操作对LevelDB读性能的影响，即更新操作引入大量旧数据，占用了存储空间，加重了查询操作的读放大，最终导致读性能下降。因此笔者认为，研究如何减少读写混合型负载下更新操作引入的旧数据对于提高LSM-Tree的读性能具有重要意义。

表1 YCSB 工作负载配置

负载类型	U1	U3	U5	U7	U9
更新	10%	30%	50%	70%	90%
查询	90%	70%	50%	30%	10%

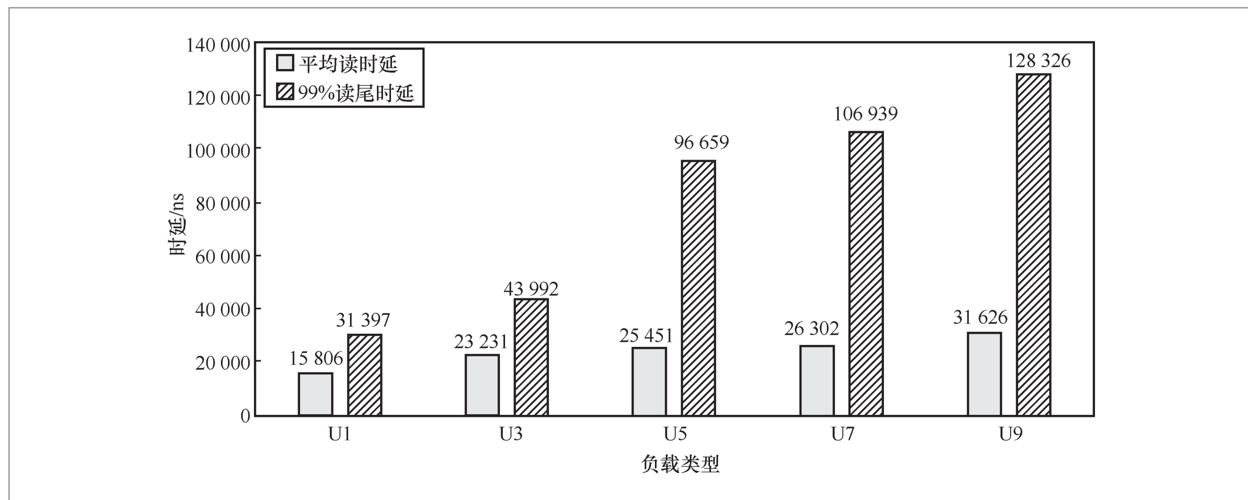


图 2　基于固态硬盘的 LevelDB 读时延变化

2 积极的压缩

更新操作产生的旧数据加重了LSM-Tree 压缩过程的写放大和查找过程的读放大，从而降低了读性能。针对这个问题，笔者提出了一种积极的压缩方法，能够有效降低LSM-Tree的写放大，并有效提升LSM-Tree的读性能。该方法的总体架构如图3所示，笔者为LSM-Tree添加了位于内存中的更新表和积分表，增加了压缩的触发条件，并优化了压缩过程，使其能够彻底清除旧数据，同时优化了LSM-Tree的查询方法，使其能够直接找到目标键所在的SSTable。此外，笔者设计的积极的压缩方法适用于所有非原地更新的基于LSM-Tree实现的数据存储系统，因为该方法是针对LSM-Tree的压缩操作对清除旧数据具有延后性和不彻底性的问题进行优化的。原地更新的LSM-Tree键值存储系统（如Faster[13]）不适合使用该方法进行优化，而非原地更新且具有比较严重的写放大的键值存储系统（如LevelDB、RocksDB等）则较为适用。

2.1 位于内存的更新表和积分表

LSM-Tree无法彻底清除旧数据的主要原因在于无法获知某个键的最新版本所在的SSTable。因此，笔者在内存中引入了一个更新表，用于记录最近插入的键所在的SSTable。具体来说，更新表中记录的是一个二元组<KEY,SST>，即键到SSTable的映射。由此，可以快速获得某个键的最新版本所在的位置。积分表则用于统计每个SSTable包含的旧数据数量。具体地，积分表中记录的是一个二元组<SST,Score>，即SSTable及其对应的积分。SSTable中的旧数据越多，其积分也就越高。

为了使更新表能够准确记录键的最新版本所在的SSTable，其工作机制如下。首先，当键即将被写入磁盘中的SSTable（SST_{new}）时，更新表会进行记录。若 KEY_0 此前在更新表中没有对应的条目，则插入一个新的 $<KEY_0, SST_{new}>$ 条目，若 KEY_0 此前已有记录 $<KEY_0, SST_{old}>$，说明 KEY_0 更新了，需要将此条记录修改为 $<KEY_0, SST_{new}>$，而 SST_{old} 中存储的 KEY_0

已经变为旧数据。

积分表对每个SSTable的积分更新是通过捕捉键的更新来完成的，这需要依赖于更新表。当更新表的某个条目发生修改（而不是插入）时，例如由 <KEY_0, SST_{old}> 修改为 <KEY_0, SST_{new}>， SST_{old} 中的旧数据就产生了，其对应的积分应该加1。

更新表的大小是一个可变的参数。记录所有插入LSM-Tree中的键值对的更新情况需要非常大的内存开销。为了节省内存开销，更新表只记录最近的键值对更新情况，因此使用最近最少使用（least recently used，LRU）缓存机制来实现更新表。更新表越大，能够记录的历史记录越多，对旧数据的清除越有利；更新表越小，所产生的内存开销也越小。积分表采用的数据结构是哈希表，积分表的大小不设上限，因为其记录的只是SSTable的ID以及其对应的积分，且积分表中的条目会随着SSTable的删除而删除，不会处于无限增长状态，因此积分表产生的内存开销非常小。目前笔者采用了单线程的LRU缓存机制来实现更新表，在高并发情况下，笔者将考虑采用多线程LRU缓存机制来提高更新表的并发访问性能。

2.2 压缩触发条件

在保留LSM-Tree原本的压缩触发条件的同时，笔者为LSM-Tree加入了额外的压缩触发条件，目的是提前让包含大量旧数据的SSTable触发压缩。具体地，当积分表中某个条目的积分超过一定阈值（old_thres）时，表明该条目对应的SSTable中包含大量的旧数据，可以直接对其触发压缩。原本需要等到该SSTable所在的层次容量不足时才能触发压缩，这样导致了旧数据可能在该层次驻留较长的时间，新的触发策略则能够提前清除这些旧数据。

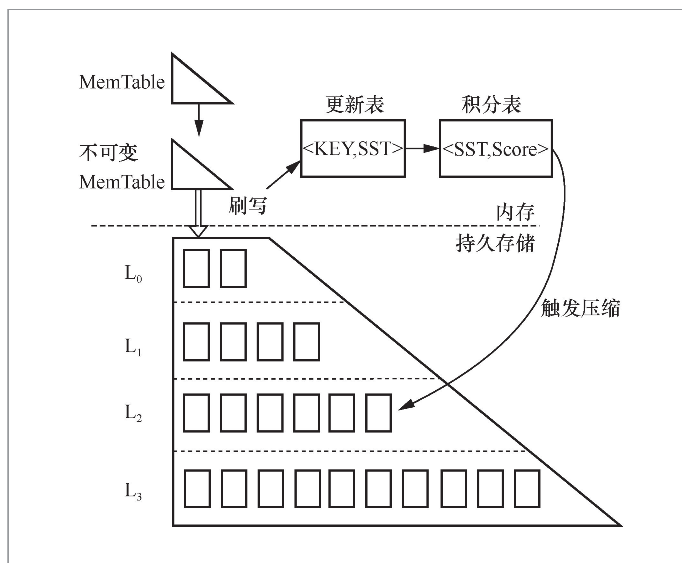


图 3 积极的压缩方法总体架构

2.3 压缩过程

依据内存中的更新表，能够快速获得某个键的最新版本所在的SSTable，因此在压缩过程中能够区分某个键是否为最新版本，并且访问内存中的更新表带来的开销极小。在压缩过程中只需要加入相当简单的判断即可达到清除旧数据的目的。具体的压缩过程如下。

如图4所示，在遍历参与压缩的每个键时，对于键，访问更新表是否存在KEY_i对应的条目，若更新表中没有，则此键最近没有发生更新，将其视为最新版本并保留；若更新表中存在对应条目<KEY_i,SST_i>，则对比SST_i与当前参与压缩的SSTable是否一致，若一致，则KEY_i是最新版本，继续保留；若不一致，则此SSTable中的KEY_i已是旧数据，可以清除，不再写入新的SSTable中。

由于在压缩过程中，参与的SSTable会被删除，保留下来的数据会被写入新的SSTable并保存到磁盘中，因此需要对更新表和积分表进行相应的维护。对于更新

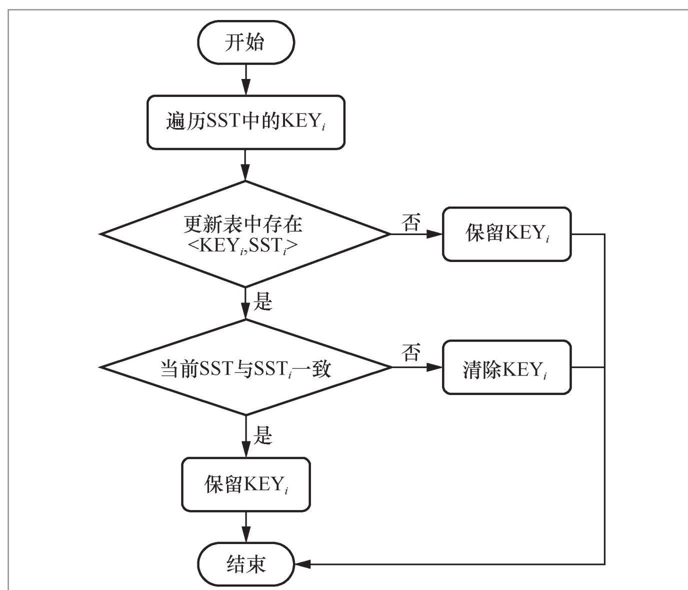


图 4 积极的 Compaction 判别旧数据流程

表来说，其每个条目记录的是键以及其最新版本所在的SSTable，若该SSTable参与了压缩，则表明该键的最新版本所在的SSTable发生了变化，因此需要更新对应的条目。例如，更新表中记录了$<KEY_i,SST_i>$，且SST_i参与了某次压缩，KEY_i被保存到SST_j中，则更新该条目为$<KEY_i,SST_j>$。同时，某个SSTable参与了压缩，说明其中的旧数据已经被彻底清除，无须在积分表中继续记录该SSTable的积分，因此可以直接从积分表中移除该SSTable对应的条目。

2.4 加速查询

本文的研究目的是清除旧数据。一方面减少囤积在LSM-Tree中的旧数据以弱化在查询过程中产生的读放大效应；另一方面优化查询路径，最终达到加速查询的目的。

LSM-Tree在磁盘中查询数据时，首先要遍历L_0层的所有SSTable，对于L_0层以外的其他层次，先比较SSTable的键范围，若所查询的键落在某个SSTable的键范围中，就需要读取该SSTable进行二分查找，然后依次逐层查找。这样的查询效率较低，需要经过多次磁盘访问之后才能找到目标键，造成读放大效应。而旧数据对读过程的影响则体现在其占用了SSTable，若将囤积在LSM-Tree中的旧数据清除，一方面可减少占用的磁盘空间，在写入数据量相同的情况下占用的层次数更少，有效数据位于更浅的层次，查询速度也就越快；另一方面可减少无效数据占用的SSTable数量，减少查询过程中对SSTable的无效访问。

本文设计的更新表记录了每个键的最新版本所在的SSTable，而查询请求的目标是找到该键的最新版本，因此，若查找时在内存中的MemTable和不可变MemTable中都不命中，说明该键保存在磁盘的SSTable中，可以通过查询更新表，获得该键的最新版本所在的SSTable，直接对其进行数据读取，而不需要经历原来的遍历L_0层以及对其他层次进行二分查找的过程。

更新表的作用与缓存不同，它记录的是最近更新的数据，而缓存记录的是最近读取的数据。在实际应用场景中，读操作与更新操作常常是混合的，且最近更新的键也很有可能马上被读取，因此更新表在读性能方面也能实现有效提升。此外，缓存的目标是存储频繁访问的数据，但是最近读取的数据也会被写入缓存中，因此缓存容易受到范围查询的影响，使得非频繁访问的数据代替了频繁访问的数据，造成缓存命中率低下。另外，缓存无法与数据同步更新，一旦SSTable进行了压缩，缓存的数据就会失效，需要等待下一次从SSTable中读取后才能再次生效。而我们通过更新表加速查找可以弥补缓存在这两

方面存在的问题。

3 性能优化效果对比

本文基于LevelDB实现了上述积极的压缩方法，本节将展示该方法为LevelDB带来的优化效果，包括在读写混合负载基准测试和YCSB[11]基准测试下的读性能提升效果以及写放大优化效果。实验运行在装有CentOS 7.6、Linux 3.10内核、英特尔Xeon 2.3 GHz处理器、16 GB内存的机器上，对基于固态硬盘的LevelDB进行了测试。首先，介绍本文所提方法涉及的参数对优化效果的影响；其次，设计自定义的读写混合负载，并将本文所提方法与原来的LevelDB和PebblesDB进行对比；最后，使用YCSB基准测试，将优化后的LevelDB与原来的LevelDB和PebblesDB进行对比，观察优化效果。本文通过平均读时延和99%读尾时延来反映读性能，通过平均写时延来反映写性能，时延越低，则读、写性能越高；通过写入硬盘的总数据量来反映写放大，写入的总数据量越大，写放大越严重。本文针对4个不同的指标——平均读时延、99%读尾时延、写入数据量、平均写时延，用式（1）量化所提方案的优化效果。其中，T_{active}表示优化后的LevelDB（active）的指标数据，T_{origin}表示原来的LevelDB的指标数据，P为优化后的LevelDB（active）相比原来的LevelDB在某个指标上减少的百分比，P值越大，优化效果越好。

$$P=\left(1-\frac{T_{\text{active}}}{T_{\text{origin}}}\right)\times 100\% \tag{1}$$

PebblesDB通过降低LSM-Tree中每一层的有序度，减少压缩的次数以及每次压缩涉及的SSTable数量，从而缓解写放大，代价是牺牲了部分读性能，尽管PebblesDB也对读操作进行了优化，以对每个SSTable创建一个布隆过滤器的行为代替对每个块创建布隆过滤器的行为，避免对SSTable的无效读取，但此优化方案仍不可避免从磁盘中读取布隆过滤器块的行为，且增加了对SSTable进行二分查找的时间。此外，PebblesDB也是基于LevelDB实现的键值存储系统，因此本文选择将PebblesDB作为对LevelDB进行写放大优化的另一种方案，与本文所提优化方案进行对比，证明本文所提优化方案在降低写放大和提升读性能两者之间取得了较好的平衡。

3.1 更新表大小与old_thres参数

本文所提方法涉及两个参数：更新表大小与old_thres参数。为了确定合适的参数大小，首先使用不同的参数大小进行了实验。图5（a）展示了读性能在不同更新表大小下的变化情况，采用的负载是YCSB 工作负载U9（负载配置见表1），总键值对数量为10 MB，键大小为8 byte，值大小为1 024 byte，横坐标为参数变化情况，纵坐标为读时延。图5（a）对应的old_thres参数固定为10 000，不论是平均读时延还是99%读尾时延，随着更新表的增大都呈下降趋势，随后保持稳定。这是由于更新表缓存了最近更新的键值对，其大小越大，能够缓存的键值对越多，得到的性能提升也就越多，而当更新表为16 MB时已经能够满足该负载的缓存需求，因此更新表继续增大对性能影响不大。更新表的最优大小与负载的总键值对数量相关，总键值对数量越多，设置越大的更新表越能够加速更多的查询请求，也能记录更多的数据更新情况。虽然在实际的场景中总键值对数量往往非常大，但为了不产生过多的内存开销，更新表无须记录所有

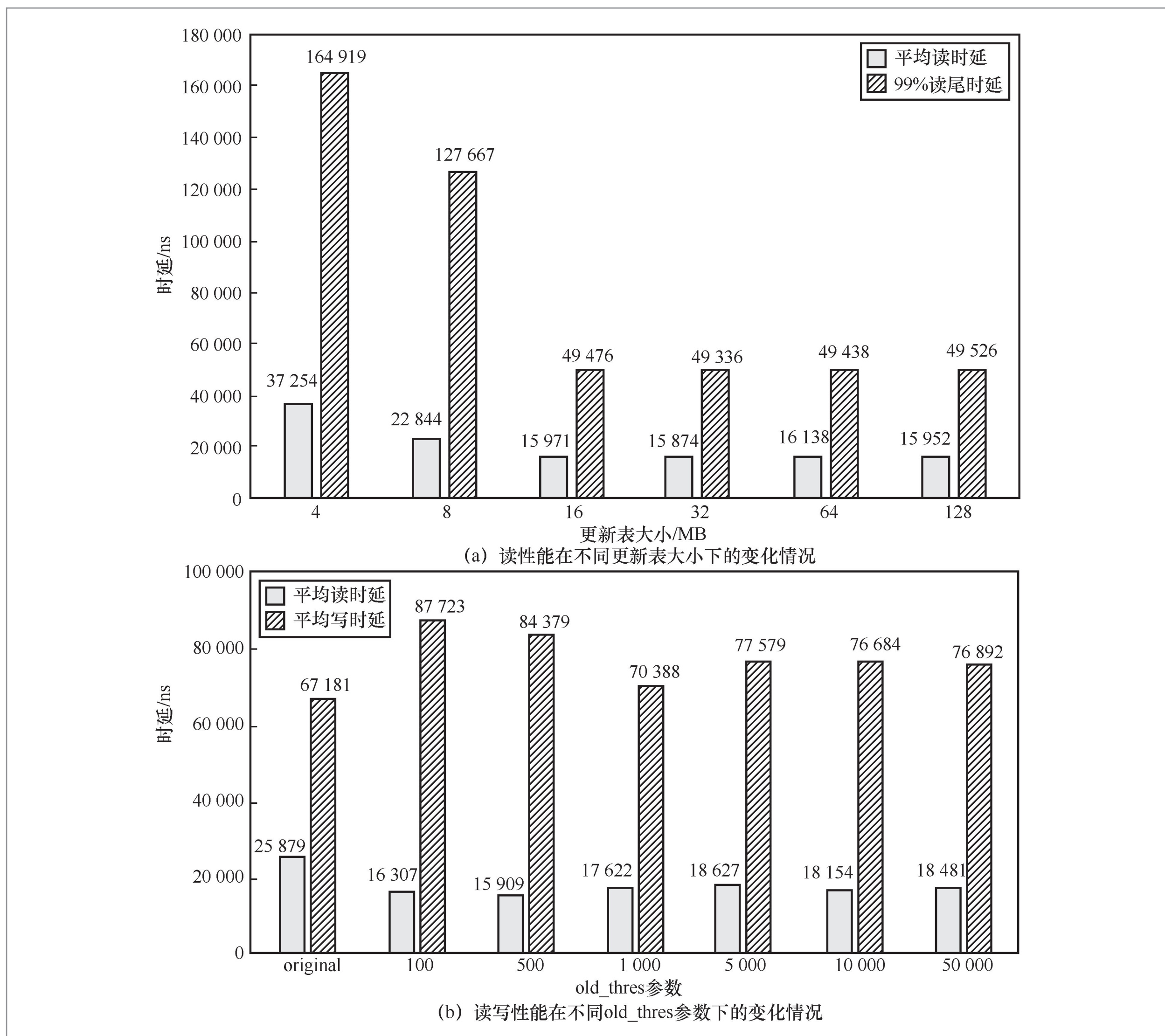


图 5 不同参数对优化效果的影响

键值对的更新情况。即使只设置16 MB大小的更新表也能够加速非常多的查询请求，在实际场景下可以权衡实际的需求和开销，对更新表大小进行设置。图5（b）展示了读性能和写性能在不同old_thres参数下的变化情况，采用的负载是自定义的读写混合负载（负载配置见表2，update_times=1，N=5×107）。此时的更新表大小固定为64 MB，图5（b）中的平均写时延统计的是所有更新操作的写时延。当old_thres参数较小时，额外触发的积极的压缩较多，因此读时延略有降低，写时延则较高。当old_thres参数为1 000时，写性能达到最优，与原来的LevelDB（即original）相差较小，表明额外触发的压缩减少了旧数据，因此减少了由空间不足所触发的压缩，二者达到平衡。当old_thres参数继续增加时，虽然额外触发的积极的压缩会减少，但由于存在较多旧数据，LevelDB会因为空间不足而触发压缩，因此写性能又会降低。

虽然图5展示的是更新表大小与old_

thres参数各自的变化情况，但old_thres参数的作用也与更新表大小相关。若更新表过小，则记录的更新操作也就越少，因此SSTable对应的积分也会减少，此时积分大于old_thres参数的SSTable过少，也就难以及时清除旧数据。因此接下去的实验将更新表大小设置为64 MB，综合读写性能将old_thres参数设置为1 000。

3.2 混合负载基准测试

在实际应用场景下，用户对数据的插入、更新和查询操作常常是混合的，例如一些移动社交应用可能会跟踪用户的定位，并向用户发送特定的广告。在这种场景下，移动的用户会频繁地更新定位，而发送广告的后台则会频繁地查询用户的定位[11]。因此，为了模拟真实应用场景，本文设计了插入、更新和查询操作混合的读写混合负载。读写混合负载配置见表2，本基准测试中键平均大小为8 byte，值平均大小为1 024 byte，此时插入请求数$N=1\times10^7$，并设置了不同数量的更新操作。表2中的update_times参数表示对每个键的更新次数，该参数的值越高，则更新次数越多。

图6所示为使用本文提出的积极的压缩方法进行优化的LevelDB取得的良好的读性能优化。在update_times参数变化的情况下，优化后的LevelDB（图6中的active）的读性能均为最优。当update_times=4时取得的优化效果最好，根据式（1），与原来的LevelDB相比，优化后的LevelDB（active）降低了45.3%的平均读时延，降低了59.7%的99%读尾时延。在此读写混合负载下，随着update_times值增加，原来的LevelDB与优化后的LevelDB（active）的读性能变化幅度较小，PebblesDB则出现较为明显的读性能降低。这是由于触发的压缩策略不同。

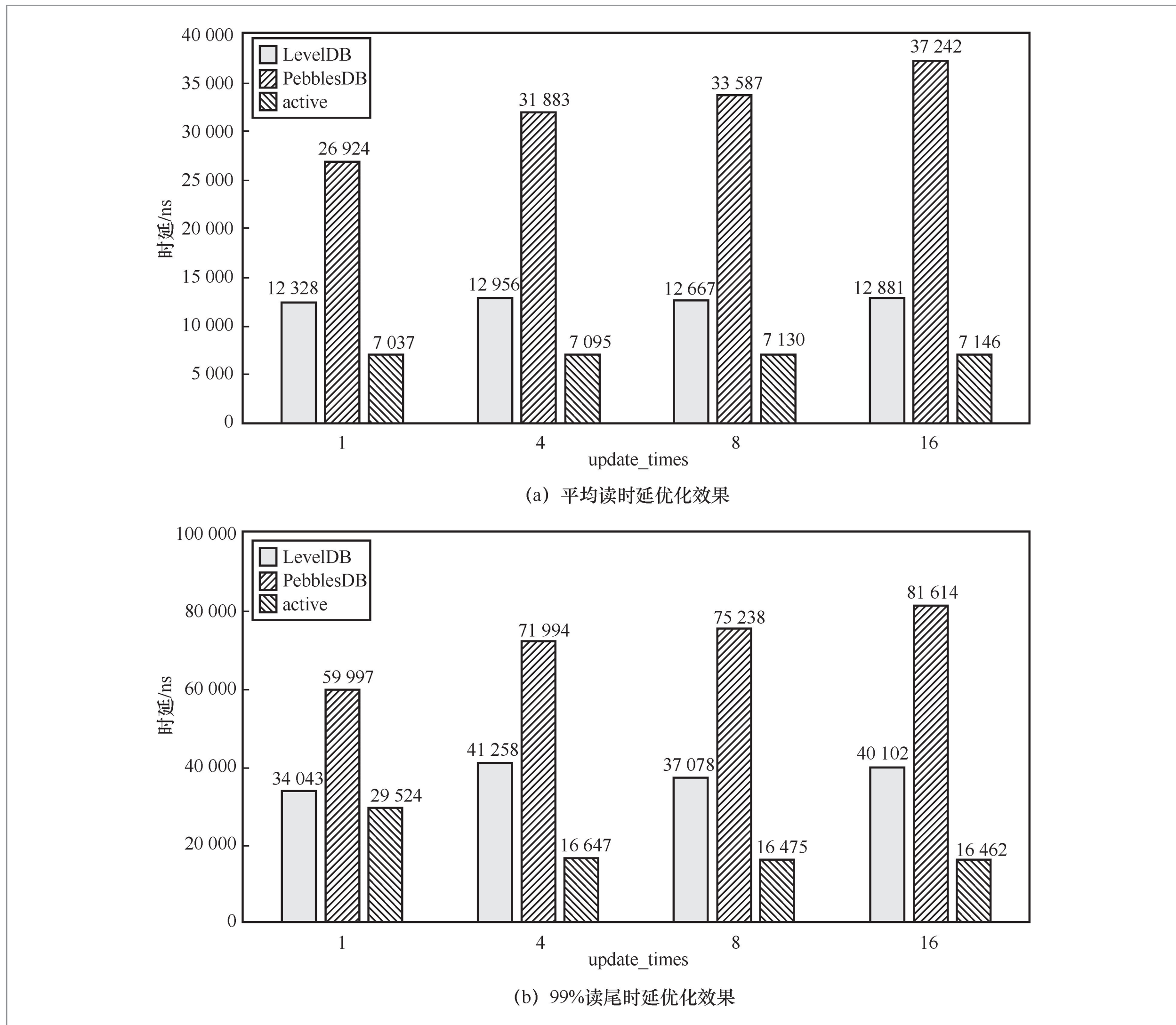
表2 读写混合负载配置

键大小/byte	值大小/byte	插入请求数	更新请求数	读请求数
8	1 024	N	$N\times$update_times	N

PebblesDB对每一层内的SSTable进行了分组，当某个分组内的SSTable数量达到组容量上限后就会进行压缩。而LevelDB则等到每一层存储的SSTable达到层容量上限后才会触发压缩。因此随着update_times值增加，PebblesDB执行的压缩会更频繁，将更多的数据压入更底层，而LevelDB中最近写入的数据能够在较浅的层次停留更长的时间，在此场景下能够及时与更新的数据进行压缩。根据此负载插入、更新、读写混合的特点，最近被更新的数据也倾向于被读取，因此LevelDB的读性能随update_times参数的增加变化不大。此外，在插入、更新、读写混合的情况下，尽管原来的LevelDB也能够较为及时地进行压缩，但优化后的LevelDB（active）有更新表加速查询，因此读性能更优。

3.3 YCSB基准测试

YCSB[15]基准测试是工业界用于评测键值存储系统的标准，它提供了多种负载类型。表1和图2展示了基于固态硬盘的LevelDB随着更新操作的不断增加，其读时延也不断增加的情况。作为对比，笔者也测试了PebblesDB和优化后的LevelDB（active）在更新操作不断增加时的优化效果，即采用表1所示负载类型进行测试，其加载和执行阶段的数据量为10 GB，其请求分布为均匀分布。此外，笔者也选取了YCSB提供的4种经典负载进行测试，见表3。YCSB负载包括加载和执行两个



（a）平均读时延优化效果

（b）99%读尾时延优化效果

图 6 读写混合负载下的读性能优化效果

阶段，加载阶段只包含插入操作，执行阶段则包括多种类型的操作，表3所示是4种经典负载类型在执行阶段包含的不同操作类型的比例，其加载和执行两阶段的数据量均为30 GB，其请求分布为均匀分布。

表 3 4 种 YCSB 经典负载配置

负载类型	a	b	c	d
查询	50%	95%	100%	95%
更新	50%	5%	0	0
插入	0	0	0	5%

如图7所示，随着更新操作的不断增加，本文提出的优化方案与原来的LevelDB相比能够降低65.2%的平均读时延（工作负载U5）以及69.4%的99%读尾时延（工作负载U5）。而PebblesDB的读性能则比LevelDB差，且随着更新操作的增加，其读时延增加的趋势较为明显，而本文提出的优化方案读时延较为稳定，证

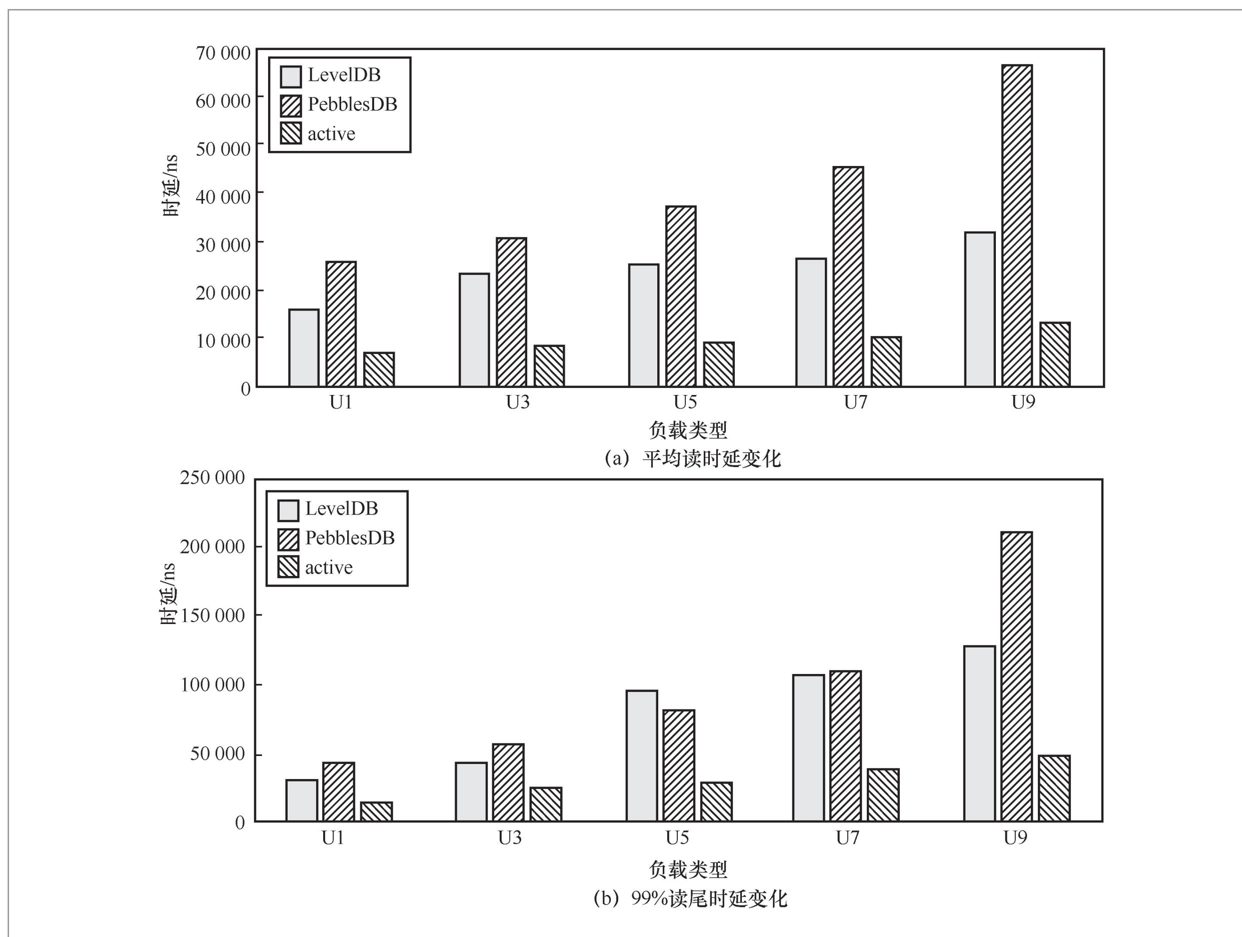


图7 表1所示负载类型下读时延优化效果

明本文提出的优化方案受更新操作的影响较小。

如图8(a)所示，对于表3中的YCSB经典负载类型，优化后的LevelDB(active)与原来的LevelDB相比读性能均有所提升，能够降低33.9%的平均读时延(工作负载a)。而随着更新比例的减小，优化效果逐渐减弱。与PebblesDB相比，本文提出的优化方案在工作负载c测试下的读性能优化效果略显不足，这是由于工作负载c没有更新操作，因此本文提出的优化方案效果微弱，而PebblesDB则依赖于SSTable级别的布隆过滤器得到较好的读性能。如图8(b)所示，优化后的LevelDB(active)与原来的LevelDB相比能够降低71.4%的写放大(工作负载b)，而PebblesDB则在各种负载类型下都维持了最低的写放大。尽管本文对写放大的优化强度不如PebblesDB，但读性能的提升优于PebblesDB。工作负载b的更新操作比例非常低，但工作负载b负载测试的写放大优化效果却优于工作负载a，这是由于LevelDB内部设置了额外的压缩策略，即当对某个SSTable的无效访问次数超过一定阈值时会触发压缩，这也是LevelDB加大清除旧数据力度的方法之一。采用了积极的压缩之后，更多旧数据会被及时清除，且更新表具有直接找到目标SSTable的

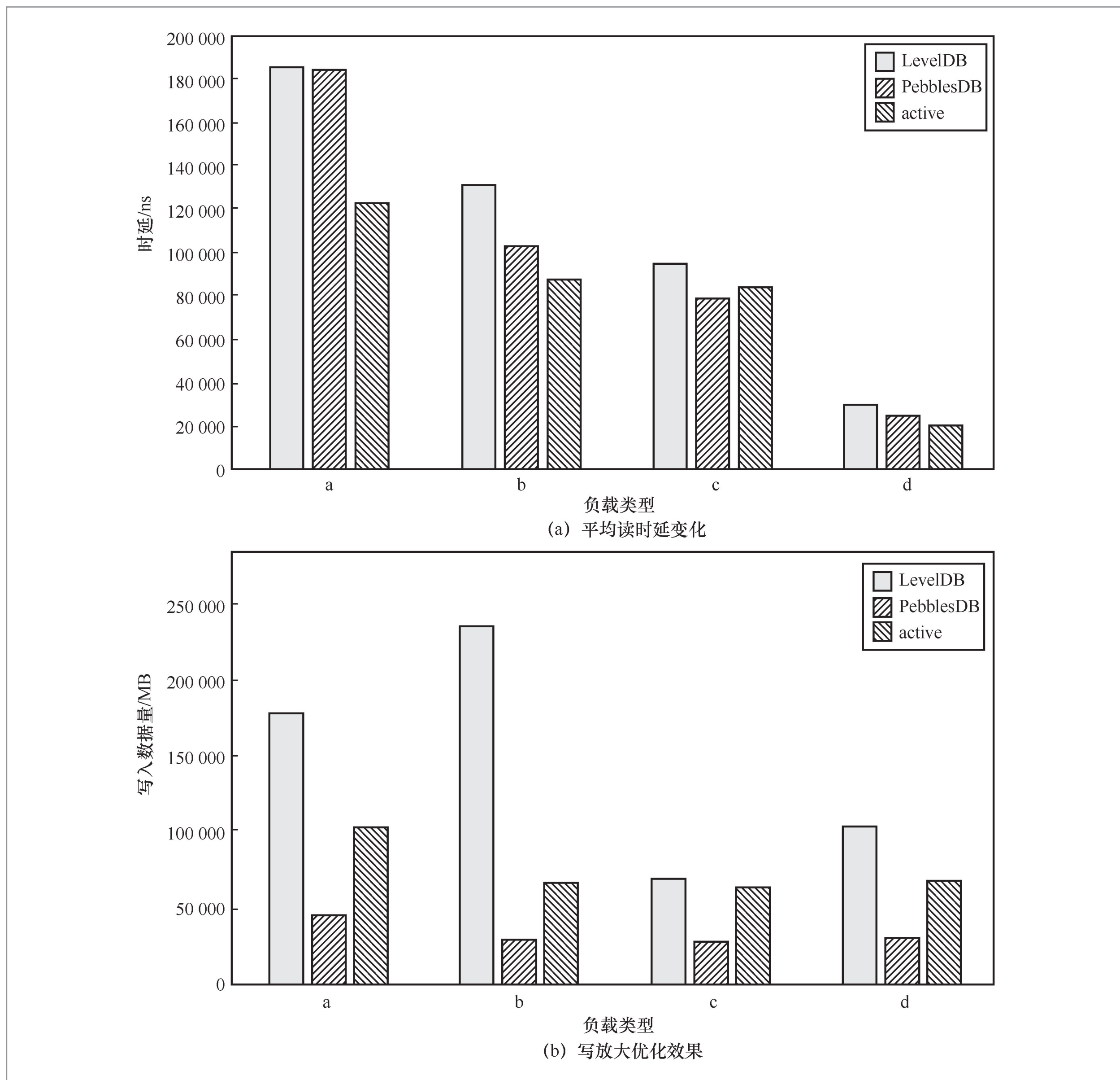


图 8 表 3 所示负载类型下的平均读时延和写放大优化效果

功能，因此对SSTable的无效访问次数会减少。

积极的压缩方法在压缩过程中会额外访问更新表判定旧数据，且该方法还设置了主动触发压缩的策略，可能会导致压缩次数过多，占用过多带宽和CPU资源。为了探究积极的压缩所带来的额外开销，笔者分析了表3所示负载类型测试下执行阶段的平均写时延（更新或插入操作）。在工作负载a测试下，优化后的LevelDB（active）与原来的LevelDB相比有9.3%的写性能下降，说明总压缩次数以及每次执行压缩的时间能够达到较为合理的平衡。在读密集型负载（如工作负载b和工作负载d）测试下，其平均写时延与LevelDB和PebblesDB相比反而有所降低，这是由于省去了由无

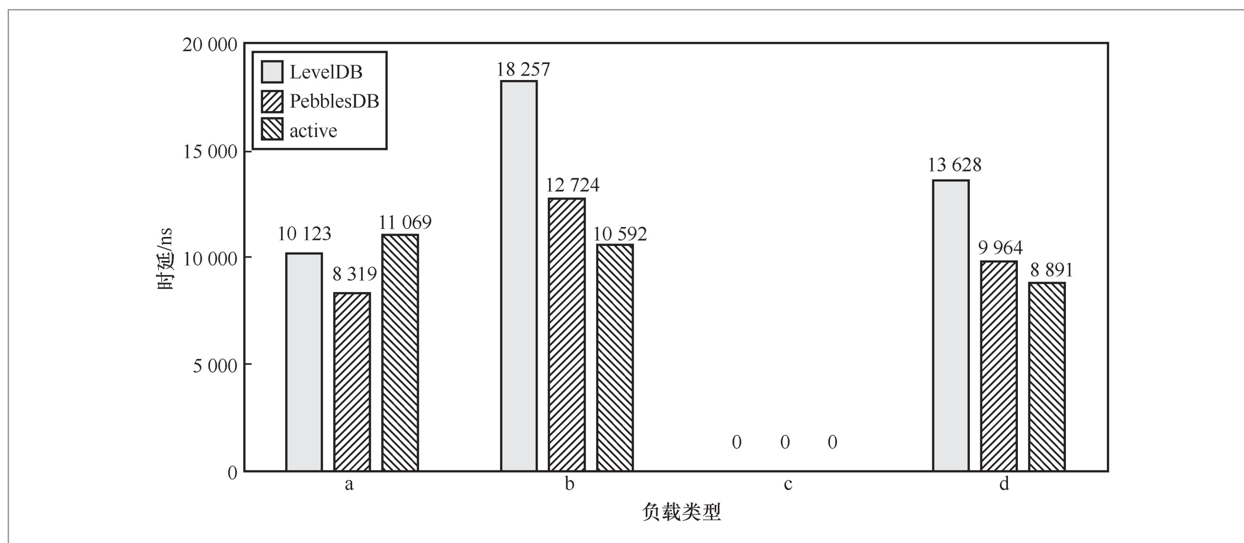


图 9 表 3 所示负载类型下的平均写时延对比

效访问次数触发的压缩操作，由读操作引发的压缩减少了，进而减少了占用的带宽，因此提升了写入速度。

4 结束语

LSM-Tree是一种广泛使用的大数据索引结构，在实现极高写入性能的同时，给数据的更新带来了大量的旧数据，降低了其读性能。本文提出了一种积极的压缩方法，能够提前触发压缩，并在压缩过程中彻底清除旧数据，同时优化了LSM-Tree的查询逻辑，减少了旧数据带来的负面影响，提升了读性能。实验表明，本文提出的优化方案能够降低LevelDB 65.2%的平均读时延、69.4%的99%读尾时延以及71.4%的写放大。

参考文献:

[1] O'NEIL P, CHENG E, GAWLICK D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[2] GHEMAWAT S, DEAN J. LevelDB[Z]. 2016.

[3] LAKSHMAN A, MALIK P. Cassandra[J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.

[4] YAO T, ZHANG Y W, WAN J G, et al. MatrixKV: reducing write stalls and write amplification in LSM-tree based KVStores with matrix container in NVM[C]//Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference. Berkeley: USENIX Association, 2020: 17-31.

[5] LU L Y, PILLAI T S, GOPALAKRISHNAN H, et al. WiscKey[J]. ACM Transactions on Storage, 2017, 13(1): 1-28.

[6] RAJU P, KADEKODI R, CHIDAMBARAM V, et al. PebblesDB: building key-value stores using fragmented log-structured merge trees[C]//Proceedings of the 26th Symposium on Operating Systems Principles. New York: ACM Press, 2017: 497-514.

[7] YAO T, WAN J G, HUANG P, et al. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores[C]//Proceedings of 33rd International Conference on Massive Storage Systems and Technology. [S.l.:s.n.], 2017.

[8] LI Y K, TIAN C J, GUO F, et al.

Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores[C]//Proceedings of the 2019 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2019: 739-752.
[9] WU F G, YANG M H, ZHANG B Q, et al. AC-key: adaptive caching for LSM-based key-value stores[C]//Proceedings of the 2020 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2020: 603-615.
[10] WU Y H, XU S, JIANG Z L, et al. LSM-trie: an LSM-tree-based ultra-large key-value store for small data items[C]// Proceedings of the 2015 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2015: 71-82.
[11] SHIN J, WANG J G, AREF W G. The LSM RUM-tree: a log structured merge R-tree for update-intensive spatial workloads[C]//Proceedings of 2021 IEEE 37th International Conference on Data Engineering. Piscataway: IEEE Press, 2021: 2285-2290.
[12] XIONG X P, AREF W G. R-trees with update memos[C]//Proceedings of 22nd International Conference on Data Engineering. Piscataway: IEEE Press, 2006: 22.
[13] CHANDRAMOULI B, PRASAAD G, KOSSMANN D, et al. FASTER: a concurrent key-value store with in-place updates[C]//Proceedings of the 2018 International Conference on Management of Data. New York: ACM Press, 2018: 275-290.
[14] ALSUBAIEE S, BEHM A, BORKAR V, et al. Storage management in AsterixDB[J]. Proceedings of the VLDB Endowment, 2014, 7(10): 841-852.
[15] COOPER B F, SILBERSTEIN A, TAM E, et al. Benchmarking cloud serving systems with YCSB[C]//Proceedings of the 1st ACM symposium on Cloud computing. New York: ACM Press, 2010: 143-154.

作者简介

林清音（1999- ），女，中山大学计算机学院硕士生，主要研究方向为存储系统。

陈志广（1984- ），男，博士，中山大学计算机学院副教授，主要研究方向为大数据存储与处理、并行与分布式计算、高性能计算与超级计算机。

收稿日期：2022-01-26
基金项目：国家重点研发计划基金资助项目（No.2021YFB0300103）；国家自然科学基金资助项目（No.61872392，No.61832020，No. U1911401）；广东省自然科学基金资助项目（No.2018B030312002）
Foundation Items: The National Key Research and Development Program of China (No. 2021YFB0300103), The National Natural Science Foundation of China (No.61872392, No.61832020, No.U1911401), The Natural Science Foundation of Guangdong Province (No.2018B030312002)