Deferred secondary index update with *LSM-trees*

Vladislav Shpilevoy
Lomonosov Moscow State University
v.shpilevoy@tarantool.org

Konstantin Osipov
Tarantool
kostja@tarantool.org

Dmitry Volkanov
Lomonosov Moscow State University
volkanov@lvk.cs.msu.su

ABSTRACT

Log-Structured merge-trees (LSM-trees) are becoming more and more widely adopted as a staple data structure in secondary storage database management systems, ending the half-a-century-long dominance of B-trees. The major LSM-tree's advantage is that it writes new data and old data updates on disk always sequentially. It is possible due to LSM-tree's ability to store many versions of the same key - it allows to LSM-based tables do not read and delete old data explicitly from primary index on such operations as deletion or replace to delete old data. Data is deleted by new data during compaction instead. But this advantage is nullifed by secondary indexes, because replace/delete-like operations require read and delete old data from each secondary index explicitly. This paper presents a modification for LSM-tree data structure, which allows to do not read any index on replace/delete operations even if there are non-secondary indexes in the table. Experimental research of the modified LSM-tree shows write speed growth 1.5 to 10 times against original LSM-tree on tables with 2-4 non-unique secondary indexes. And the more secondary indexes there are, the faster the new LSM-tree works on write operations.

PVLDB Reference Format:
Vladislav Shpilevoy, Konstantin Osipov, Dmitry Volkanov. Deferred secondary index update with LSM-trees. *PVLDB*, 11 (5): xxxx-yyyy, 2018.
DOI: https://doi.org/TBD

1. INTRODUCTION

Log-Structured Merge trees were developed for write-intensive tasks in 1990th and were used in file systems and for reserve copying. But their prevalence was restricted by hidden reads.

These are the reads which accompany writes: e.g. when its necessary to delete old data from a secondary index after an update, or check an unique or a foreign key constraint. A significant amount of reads in a modern database is hidden. Such reads, when followed by a write, cost nearly nothing in a B-tree, but can penalize performance of an LSM-tree based table to an extent when its write advantage becomes completely diluted.

Proceedings of the VLDB Endowment, Vol. 11, No. 5

DOI: https://doi.org/TBD

With Solid-State Drive (SSD) appearance cost of random writes had started contribute to overall performance way more heavily than cost of random reads, and nowadays the LSM-tree is consideted to be a standard data structure for databases. For example, it is used in LevelDB, RocksDB, Cassandra, Tarantool, BigTable, Hbase, Riak, MySQL (MyRocks).

However SSD does not help in a special case, which is very popular and dilutes LSM-tree's ability to store multiple versions of the same keys. It is linked with LSM-tree compaction algorithm, and appears on tables with multiple indexes.

In a primary LSM-tree based index any replace or delete operation is executed with no hidden reads because of LSM-tree ability to store multiple key versions. A new tuple is just inserted into memory and after some compactions discards old tuples. But it is not possible if there are secondary indexes. If a table contains secondary indexes, any update operation must read an old tuple from a primary index, extract from it secondary keys and delete them from each secondary index explictly. That is any update of such table produces hidden reads from a primary index.

Primary index read and explicit old tuple deletion can not be avoided in classic LSM-tree, because it deletes old versions of the key only if these versions are equal by the key. If a request changes a secondary key in an exising tuple and does not deletes old one, then it is not deleted from a secondary index. LSM-tree considers new and old tuple not to be different versions of the same key, but to be different keys.

The paper describes a new LSM-tree compaction algorithm, a new LSM-tree based table update and read algorithms. Replace and delete operations on the new LSM-tree do not any reads regardless of secondary index number, if all of them are not unique. Experiments show exponential requests number per second (RPS) grow on some loading types and some schemas. For example, on a table with 3 non-unique secondary indexes and replace/delete batched requests RPS increased up to 10 times.

2. BACKGROUND AND MOTIVATION

This section defines some LSM-tree basics: how it is can be stored on disk and in memory, how compaction, update and read works. What LSM-tree parameters are typically configurable.

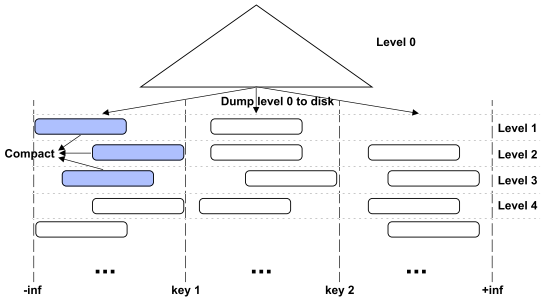

Figure 1: LSM-tree ranges dump and compaction

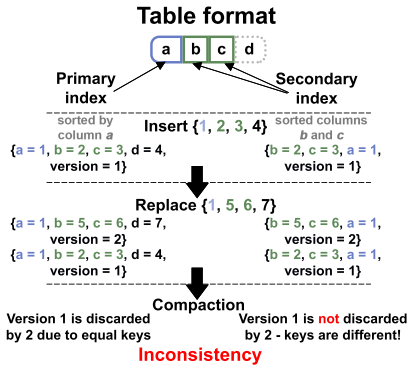

Figure 2: LSM-tree based table inconsistency example

LSM-tree is a data structure optimized for write-intensive tasks. It has multiple levels: zero-level is stored in a memory and other on a disk. Zero-level amasses updates and periodically is dumped on to a disk to become first level. Zero-level after dump are cleaned up and collects new updates. And so on.

Zero-level usage allows to write updates to a disk in batches regardless of their order, keys, types. Even delete is put in a zero-level as a tuple of a special type.

When level number became too big, the LSM-tree is compacted - several levels are merged in a new one. During compaction new tuples discard old tuples of the same key. For example, delete discards all older data and sometimes itself; replace discards all older data, but not itself.

LSM-tree can store levels in multiple ways. For example, memory level can be B-tree or Red-Black tree. Disk levels can be organized as B-trees or sorted arrays. In the paper sorted arrays on disk and B+ tree in memory are considered.

2.1 Compaction

Each level consists of multiple sublevels. Each sublevel is an array sorted by a key (in a case of LSM-tree index the key consists of index parts) and containig a specific key values range. During compaction some sublevels of each levels are merge-sorted into a new sorted array. It is known, that if one key version is on level i and another on $j < i$, then the key version on level j is newer and another one is discarded when compacted.

Compaction procecure is needed to reduce level count and to discard old tuples to make room for new ones. Compaction frequency is regulated by LSM-tree parameters level size ratio and maximal sublevels count. In each pair of neighbour levels i and $i + 1$ size of a level $i + 1$ must be greater than level i size $*$ level size ratio at least. If maximal sublevels count per level is exceeded or level size ratio is violated, then compaction procedure merges as much sublevels as needed to satisfy both limitations.

In practice, sublevels are produced by memory level dump and by spliting index key values into ranges, when each range stores and maintains its own sublevels independently of other ranges (see Figure 1).

2.2 LSM-tree based table structure

In LSM-tree based table each index is an LSM-tree, sorted by the index parts. It is common pattern, that a table has one primary index and a number of secondary indexes. A primary index always is unique, and stores full tuples, including all indexed and not indexed fields, as they were inserted by an user. A secondary index stores only its own key parts and primary index parts to save memory. Such indexes are called non-covering. Primary parts are used as link to a full tuple in a primary index, when a search uses a secondary index only.

The described pattern is not unique feature of LSM-tree based tables. For example, SQLite secondary indexes based on B-tree are not covering by default too. And each SQLite table has unique primary index even if an user did not specified it explicitly.

Since secondary indexes stores primary index parts, they are linked with primary indexes. And if a tuple does not exist in a primary index, the same tuple can not exist in a secondary index. This is the reason, why hidden reads are indispensible - if a tuple is replaced with another secondary key or deleted from a primary index, then its old version must be deleted from each secondary index by old secondary key, else a table indexes are not consistent.

2.3 Multiple indexes update problem

LSM-tree ability to manage multiple versions of a key allows to avoid hidden reads on such operations as replace or delete which can not break consistency if there is only a primary index in a table and no foreign key constraints. Such operations are known as **blind-writes** [5]. Blind-writes are much faster than non-blind insert or update operations, which read an old tuple in a worst case from a disk to check for duplicate or apply update operations.

The big problem of LSM-tree based tables is that in presense of secondary indexes all writes become non-blind. See Figure 2 for example of inconsistency if replace is blind and a table has secondary index. Here a table is defined with 4 columns: first is part of a primary index, 2 and 3 are parts of a secondary index, and 4 is not indexed. *Replace*$\{1, 5, 6, 7\}$ before executed must read old tuple from a primary index by the key $\{1\}$, extract the secondary key $\{2, 3\}$, delete it from the secondary index, and only then insert the new tuple. If the old tuple is not deleted, then during compaction it becames garbage with no link to a full tuple in the primary index.

Since disk access is much slower than memory access, secondary index presence destroys LSM-tree versions advantage. In the next section a new LSM-tree update algorithm is presented reviving versions and blind-writes.

3. DESIGN AND IMPLEMENTATION

Below the basics of a new algorithm are described, followed by three sections, each of which unfolds one of enclosed part of the algorithm.

The key idea of the algorithm is that deletion of old data can be deferred until primary index compaction stage. It allows to do not hidden reads during replace and delete by primary key operations since they use hidden reads only to learn and delete old secondary key versions, not to check consistency. It works on tables with non-unique secondary indexes only, because it is impossible to break consistency of non-unique secondary index using delete or replace operations.

Obviously, it is not true if there are unique secondary indexes. For example, consider a table with columns $\{a, b\}$, unique primary index on $\{a\}$ and unique secondary index on $\{b\}$. A table contains two records: $\{1, 1\}$ and $\{2, 2\}$. It is impossible now to do replace $\{1, 2\}$, because it leads to duplicate in the secondary index.

Of course, in MySQL, for example, replace would delete both old records, but such kind of replace can not be deferred because during the primary index compaction there is no way to learn that a new tuple with one primary key discards an old tuple with another primary key without explicitly inserted deletion. This multi-replace must delete old records with another primary keys, and it is not considered under the paper.

The algorithm of deferred update consists of three enclosed parts:

1. Update of a zero-level of all indexes. This is where replace and delete by a primary key are inserted into memory and makes secondary and primary indexes mal-synchronized.

2. Compaction, which differs in primary and secondary indexes. A primary index during compaction detects tuples to discard and sends them to a secondary indexes as a new LSM-tree sublevel. A secondary index on compaction takes into account sublevels received from a primary index.

3. Secondary index reading, that now can see tuples already deleted from a primary index. Existance of a concrete version of a key read from a secondary index must be check via primary index lookup.

3.1 Deferred update

Only two operations can be deferred: replace and delete by a primary key. At first algorithm of replace execution is presented, and then the delete's one which is very similar.

Consider a table with one primary index and some non-unique secondary indexes. Assume an user executes replace. A tuple specified in the replace is inserted into each indexe's memory level as is, with no preliminary reading and deletion of an old tuple from secondary indexes.

When this algorithm is used, it is possible and valid, that some secondary indexes consider discarded tuples as not discarded, like in example on Figure 3. In the example there is a tuple with version 3, which is discarded in a primary index by a newer tuple with the same key and version 5. But secondary keys of these tuples are different, and a secondary index sees the tuple with version 3 as a separate non-discarded tuple. Such tuples, visible in a secondary index and not visible in a primary one are called **dirty**.

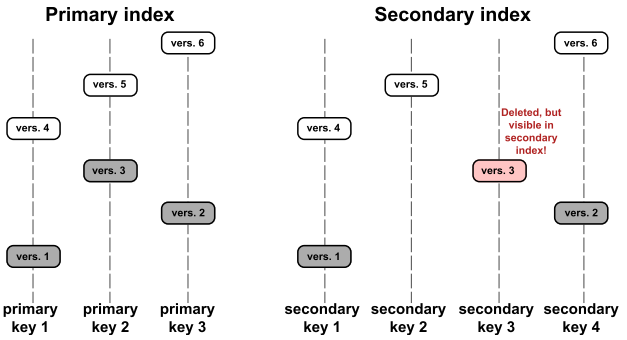

Figure 3: LSM-tree based table if update is deferred

A delete by a primary key is inserted only in a primary index memory. It can not be inserted in a secondary index, because a corresponding secondary key is unknown with no hidden reads. In such a case a tuple, discarded by the delete, becomes dirty in each secondary index.

In the next section the algorithm of deletion of dirty tuples is presented as a part of new LSM-tree compaction procedure.

3.2 Compaction

3.2.1 Primary index

Compaction is triggered by the same signals, as in the classic LSM-tree - by too big number of sublevels, or too big difference in neighbour levels size.

Consider the basic compaction, when there are no deferred updates and dirty tuples. Assume the levels consist of sublevels belong to key ranges, and a sublevel is sorted array of versioned keys.

Sublevels are compacted in merge-sorting procedure discarding tuples having older versions among compacting sublevels. That is if a tuple has newer version out of compaction participants, then it is not discarded. Two tuples are considered to be verions of the same key, if they are equal by parts of compacting LSM-tree index. If compaction participants have several versions of a key then during merge-sort they met during one of merge-steps, and only one tuple stays. After merge-sort is finished, a new single sublevel replaces old ones.

Obviously, old tuples are read from disk during compaction. And these reads replace hidden reads of update operations. Actually, the hidden reads are deferred until compaction, where they are done anyway and sequentialy - sequential read is at a far quicker than random both on SSD and HDD.

Old tuples going to be discarded are used to extract dirty secondary keys. Extracted keys are resorted into order of a secondary index, and are written as a new sublevel directly into the secondary index. Secondary keys are written as deletes saving the version of the original tuple. New primary index compaction produces subleves for itself and for each secondary index. When a secondary index compacting, it takes into account sublevels, created for it by a primary index.

Resorting into order of a secondary index is executed to observe LSM-tree indexe's invariant whereby sublevel of an index is an array tuples sorted by key parts of the index. So sublevels, created for secondary indexes during primary one's compaction, must be resorted into secondary index

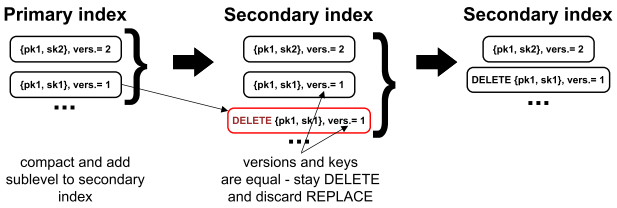

Figure 4: Secondary index compaction example

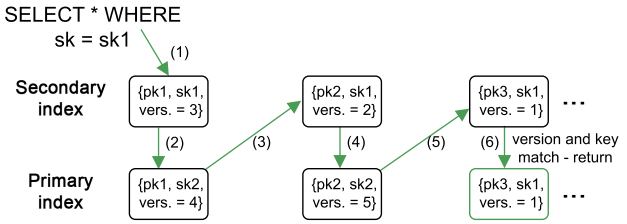

Figure 5: Secondary index read example

order. Resorting method depends on implementation, and can be

- in-memory quick sort after all tuples are read from disk;
- in-memory tree-sort, when for each index a tree is created before compaction and it is filled just as tuples are reading from disk. Tree can be of any type: binary, red-black, B-tree. When a primary compaction is finished, the tree is written into a secondary index sublevel as a sorted array. So such a tree is prefered, which can be iterated fast over all keys (B^+-tree for example - it maintains sorted list of stored values);
- disk merge-sort, which can be used when sublevels compaction discard too many tuples and there are many secondary indexes - they may no fit into memory. The idea is to do in-memory sort of discarded tuples by batches of some fixed size, and dump them as sorted arrays into temporary files. When the primary compaction is finished, the congested temporary sorted arrays are merge sorted into single sublevel intended for a secondary index. Merge-sort of temporary files is executed in the same way as sublevels compaction - read and write files in parts with no reading them all into memory. The described procedure is executed for each secondary index.

3.2.2 Secondary index

Compaction procedure of a secondary index is slightly changed to correctly process the deletes received from a primary index. The point is that the deletes, received from a primary index, have both the same version and the same key as the dirty tuples, for which they are intended, and a compaction procedure must correctly process these versions and keys match.

The only modification of a secondary index compaction is that in a case of versions and keys match of delete and replace tuples the replace tuple must be discarded, and the delete tuple must stay, if it is not **major** compaction. A compaction is major, if all sublevels of all levels are compacted. On the Figure 4 it is presented example of a secondary index compaction with sublevel, received from a primary index. In the example, a primary and secondary indexes stores two tuples: {*pk1, sk1*} and {*pk1, sk2*}. In a primary index it is the different versions of the key *pk1*. When a primary index is compacted, a tuple {*pk1, sk1*} is discarded and sent to a secondary index as a DELETE tuple with the same version. During a secondary index compaction this DELETE {*pk1, sk1*} with version 1 mets the dirty tuple {*pk1, sk1*} with version 1 - their keys and versions are equal. It is assumed, that it is not major compaction, so the tuple {*pk1, sk1*} is discarded and the DELETE tuple stays.

3.3 Read

Read of a primary index is not changed, because dirty tuples can appear in secondary indexes only. When a secondary index is read, the dirty tuples are skipped. To check a tuple if it is dirty, its primary key and version are looked up in a primary index. If a primary index contains the same key with the same version, then the tuple is not dirty and can be returned to an user.

It means, that even point reading from a secondary index can lead to multiple lookups in a primary index to exclude dirty tuples. On the Figure 5 a SELECT by a secondary key returns a tuple on a third attempt. At first, it reads {*pk1, sk1*} as the newest known version of this secondary key. It appers to be dirty, because in a primary index a tuple with the same primary key has another version and another secondary key. At second {*pk2, sk1*} is checked as the next by version: it is dirty too - in the primary index this tuple is already replaced by {*pk2, sk2*}. At third {*pk3, sk1*} is checked, and in the primary index a tuple is found with the same key and the same version - it means, that this tuple is not dirty and can be returned to an user.

The common algorithm is to read secondary index tuples and lookup each one in a primary index until their versions and keys match.

3.4 Implementation

The developed algorithm is implemented as a patch for Tarantool database's disk engine named Vinyl. Tarantool is DBMS and application server, which supports two storage engines: Memtx in memory and Vinyl on disk. Tables in Tarantool are being called *spaces*. Memtx engine stores data in memory in B^+-trees, and it is not considered here.

Vinyl engine stores indexes as LSM-trees. LSM-tree based index is split into key ranges. LSM-tree level consists of sublevels, produced by zero-level dump and compactions. Sublevels are files. Ranges are compacted independently of each other. Dump and compaction are executed in background threads, while transaction processing is in a single main thread and it does not heavy operations like disk or network access - it delegates these tasks to worker threads.

Sublevels consists of pages. Page knows its minimal and maximal keys and minimal and maximal versions - this information is stored in memory and allows to read just needed pages, not entire sublevels. Page size is configurable, and the greater it is, the smaller meta info stored in memory and the bigger chunks are read when searching for a key.

Each vinyl sublevel has bloom filter with several improvements: less hashing with the same performance [2] and blocked bloom filters [4].

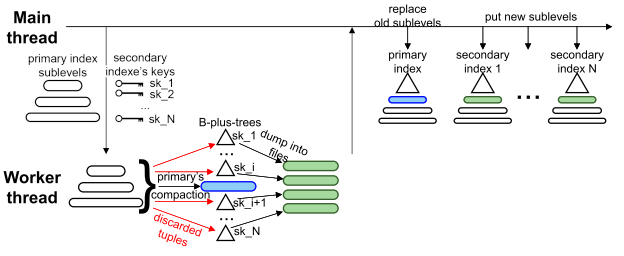

Figure 6: Primary index compaction schema

Main thread consists of coroutins written in C and called *fibers*. When a main thread wants to do long operation like LSM-tree index dump or compaction, access disk or network, it sends a request to a worker thread and switches to another fiber, while the original one waits for response from the worker. The big advantage of such schema, that there are no locks on internals except inter-thread communication queues.

Implementation of the deferred update algorithm consists of three parts like the algorithm itself: new replace and delete execution, compaction and read.

New replace/delete are quire simple: they just do not hidden reads. Replace is inserted in zero-levels of all indexes of Vinyl space, delete is inserted into a primary one only.

The compaction is the most complex part of the implementation. As applied to Vinyl and LSM-trees, the compaction procedure works as follows before the deferred update:

1. Main thread detects that a compaction is necessary (for example, level size ratio is violated or a level consists of too many sublevels), and sends a request to a worker thread with information, which sublevels need to compact, in which files they are stored;

2. Worker thread sees request and starts compaction. Compaction is executed using merge-sort of sublevels files. Processed files are not deleted or changed by a worker thread, because they may be used now in the main thread for reads. When the worker has finished the work, it has one new sublevel file. The main thread is notified, that the work is done;

3. After a while, one of fibers of the main thread sees the finished worker, gets the new sublevel, puts it in the LSM-tree and deletes old sublevels and their files atomically.

After the deferred update is applied, the compaction procedure's second and third steps are completely different. See Figure 6 as illustration. When a primary index is scheduled for compaction, the main thread sends to a worker info not only about a primary indexe's sublevel, but about each secondary index key parts. They are needed to extract secondary keys from primary index tuples.

While the primary index is being merge-sorted, discarded tuples are stored into in-memory B$^+$-trees created for each secondary index. When a primary index compaction is finished and a new sublevel is created, the filled B$^+$-trees are dumped into sublevel files for secondary indexes. The dump is fast, because B$^+$-tree can be fast iterated due to links between neighbour keys. At this step the worker thread has sublevel files for each index of the table, and it notifies the main thread about the finished work.

One of the main thread's fiber replaces compacted sublevels with a new one in a primary index, and puts secondary indexe's sublevels into corresponding indexes at the LSM-tree's first level. Secondary index sublevel received from a primary index is on the LSM-tree's first level, because it can contain any DELETEs from first to last primary index levels and it means, that it can overlap any sublevel stored on disk. These DELETEs must overlap dirty tuples for which intended.

Implementation of reading is changed to take into account dirty tuples existence. There is the same point that during a secondary index compaction - the index contains both dirty tuples and their DELETEs with the same key and version. If an iterator sees this match, it skips these versions and all olders. All other found tuples are looked up in a primary index. If in the primary index the tuple is found with the same secondary key and version, as the tuple from the secondary index, then this tuple is not dirty and can be returned to an user. Else the tuple is dirty - its DELETE while is not generated by primary index compaction. The salient feature of this implementation that sublevels, received from a primary index, are undistinguishable from usual sublevels, and they are beeing read together with others. It allows to avoid lookup in a primary index, if a tuple is dirty and its DELETE already is in the secondary index. But there is overhead for reading these additional sublevels.

There is another way to implement reading, which has its own benefits and drawbacks. The key concept is to distinguish sublevels received from a primary index from others and use them for compaction only, not for reading. It allows to read less sublevel files, but compels to do lookups in a primary index on each tuple even if the dirty tuple's DELETE is already received from the primary index.

4. MATHEMATICAL BASICS

In this section the mathematical complexity of original update algorithm is compared against the deferred one's. There are used the following known values:

N – tuples number on all levels,

b – zero-level B$^+$-tree branching factor,

r – level size ratio,

K – total indexes number.

According to LSM-tree level size ratio definition, level i is r times bigger, than $i-1$, so total count of levels is $O(log_r N)$ [5]. Lets denote it as lc.

Deferred update algorithm allows to do not reads, but it puts new tuples in memory level, and it is the only common part of old and new algorithms. Memory level access complexity depends on tuples number, which can be calculated from N and levels number. Denoting memory (alias zero) level size as m, calculate it via geometrical progression sum formula:

$$N = m + m * r + m * r^2 + ... + m * r^{lc},$$

$$N = \frac{m(1 - r^{lc+1})}{1 - lc},$$

$$m = \frac{N(1 - r)}{1 - r^{lc+1}}.$$

In the scope of the paper, memory level is B$^+$-tree, in which search and insertion complexity is $O(log_b m)$. Total complexity of replace and delete before deferred update includes point search on disk - complexity of this operation is $O(log_r N)$. It is associated with levels number because disk index of levels (pages, their minimal and maximal keys, versions) is stored in memory and needed pages on a disk for a certain key can be found with no disk access. Disk is accessed only to read pages, which can contain a sought key.

Using found parameters, the complexity of update can be calculated:

Complexity of replace:

$$O(log_b m) + O(log_r(N-m)) + O(log_b m) * (2K-1)$$

Do hidden read: scan for an old tuple in memory level ($O(log_b m)$), on disk ($O(log_r(N-m))$). In the worst case an old tuple with the same primary key is found, so insert into memory level ($O(log_b m)$) of each secondary index two tuples ($2(K-1)$): DELETE of the old one and the new tuple. In the primary index the new tuple discards the old one without DELETE, so in the primary index only one tuple is inserted ($2(K-1)+1 = 2K-1$).

Complexity of delete:

$$O(log_b m) + O(log_r(N-m)) + O(log_b m) * K$$

Do hidden read: scan for an old tuple in the primary index disk and memory ($O(log_b m)+O(log_r(N-m))$). In the worst case an old tuple is found - insert into memory level of each index DELETE tuple ($O(log_b m) * K$).

After the deferred update is implemented all hidden reads disappear.

Complexity of **deferred** replace:

$$O(log_b m) * K$$

Simply insert a new tuple into memory levels of all indexes - no DELETEs, no hidden reads. They are deferred until compaction. The deferred replace is at least twice faster than classical one according to the calculations below:

$$\frac{O(log_b m) + O(log_r(N-m)) + O(log_b m) * (2K-1)}{O(log_b m) * K} =$$
$$\frac{O(log_b m * 2K) + O(log_r(N-m))}{O(log_b m) * K} =$$
$$2 + \frac{O(log_r(N-m))}{O(log_b m) * K}.$$

LSM-trees zero level size is bounded above by RAM size, indexes number is set once when a database schema is created and is changed rare, zero-level B$^+$-tree's branching factor is constant, so the nominator $O(log_b m) * K$ can be treatted as constant. Denominator contains N, which in write-intensive tasks can be huge (billions and more) and the bigger it is the bigger is the difference between speed of deferred and simple update. For example, for the following estimation:

$$r = 3,$$
$$b = 10,$$
$$m = 10^5,$$
$$N = 10^8,$$
$$K = 5.$$

the deferred replace is faster in 2.55 times in theory. In the reality the speed growth is much and much greater, because access to disk is much slower than access to memory even for SSD. And exactly on disk speed depends the denominator $O(log_r(N-m))$ - when N is big, most of tuples are stored on disk. It is because it is easy to get speed growth up to 10 times even on small tables.

Complexity of **deferred** delete:

$$O(log_b m)$$

Insert DELETE into primary indexe's memory level only. Because only primary index is used, the deferred delete is faster than the simple in at least $K+1$ times, and the more tuples are on disk, the faster deferred delete is:

$$\frac{O(log_b m) + O(log_r(N-m)) + O(log_b m) * K}{O(log_b m)} =$$
$$\frac{O(log_b m) * (K+1) + O(log_r(N-m))}{O(log_b m)} =$$
$$K + 1 + \frac{O(log_r(N-m))}{O(log_b m)}$$

5. EVALUATION

The implemented algorithm is tested on two benchmarks: microbench, on which the difference between usual and deferred update are most significant, and linkbench. Both benchmarks are done on a single machine with Apple SSD SM0512L, Intel i7 2.7Ghz, 4 cores.

5.1 Microbench

The benchmark compares stock Tarantool vs Tarantool with deferred updates. Obviously, the biggest throughput improve can be achieved on workload consisting of many replaces and deletions to a table with non-unique secondary indexes. The database schema consists of one space on Vinyl engine, one primary index on a first field, and 4 secondary indexes each on a one field. In SQL syntax:

```
create table test (field1 unsigned integer primary key, field2
   field3 unsigned integer, field4 unsigned integer,
   field5 unsigned integer);
create not unique index on test(field2);
create not unique index on test(field3);
create not unique index on test(field4);
create not unique index on test(field5);
```

LSM-tree's zero-level size is restricted with 128Mb. Workload is generated and sent via network by 4 clients on the same machine as Tarantool (unix sockets are used - overhead of network is minimal). Each client generates replaces and deletions by primary key in batches with randmo size 1 to 500 tuples. Half of tuples are deletions and half are replaces. A tuple consists of 5 fields with total size about 50 bytes (including tuple meta info). All keys are between 1 and 1000000.

There are two worker threads to dump and compact levels. Level size ratio is 3.5, maximal sublevels count on a level is 2, bloom filter has 5% false positive rate, page size is 8Kb (about 160 tuples capacity).

The request-per-second (RPS) results are on Figure 7.

There are some aggregated RPS values in Table 1. According to values in the table, on such a little table with less than 1000000 tuples RPS growths in 6 times.

Because of much bigger RPS, the table is filled faster, as shown on Figure 8. And memory level is dumped more frequently, as shown on a figure 9.

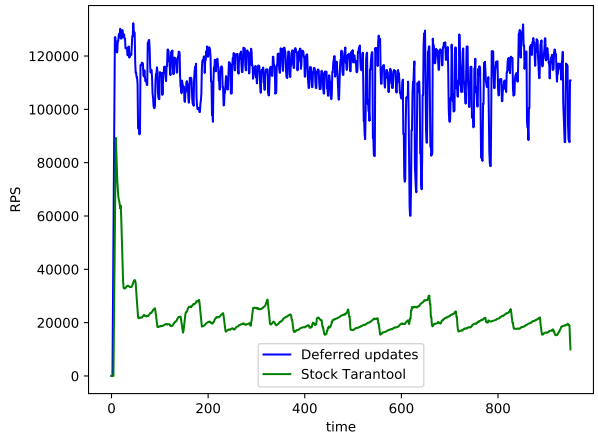

Figure 7: Microbenchmark RPS

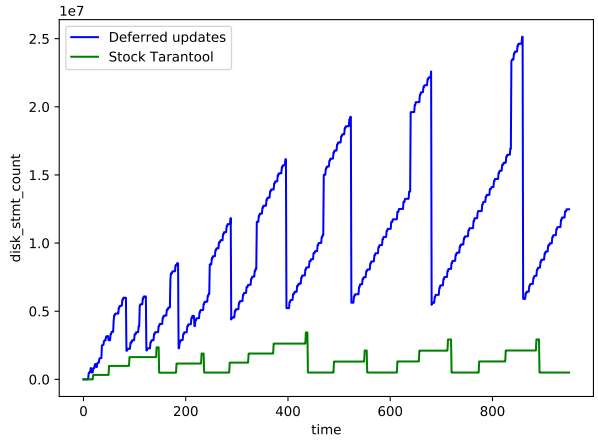

Figure 8: Microbenchmark disk statement number

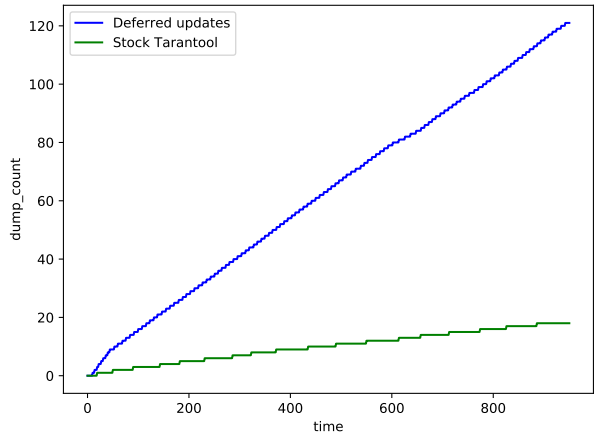

Figure 9: Microbenchmark dump number

Table 1: Microbenchmark RPS aggregated

	Deferred update	Simple update
Average	112343 r/s	21681 r/s
Maximal	132316 r/s	89292 r/s
Median	114772 r/s	20097 r/s

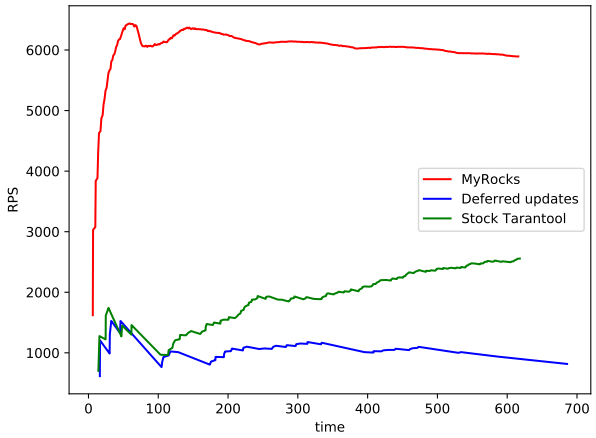

Figure 10: Linkbench RPS

5.2 Linkbench

Deferring of updates slows down readings, because any key found in a secondary index must be checked to be not dirty in a primary index. The grade of slowing depends on loading type - the more updates againts new data insertions, the more tuples become dirty, and the slower readings become. On Figure 10 Linkbench [1] results are showed. Most of requests on this Linkbench run were readings (more than 70%), and on such a loading type read requests are slower than without deferred updates in about 2 times.

6. RELATED WORK

Necessity to speed updates up is not a new task. There are some existing works on this theme: some of them try to reduce hidden reads number, another try to defer some computations until reads. Various decisions are developed not for LSM-trees only for for B-trees too.

The problem of too long updates appeared earlier than SSD disks, when B-trees was the very widespread data structure for HDDs. In the work [3] authors propose a method of deferring updates of B-tree until updated keys are read. Updates are stored in *differential files*, which are used firtly to lookup each key. Differential files were first proposed in 1976 [7] as a method of B-tree multiversion support. It is curious that LSM-tree investigated much later actually completely consists of differential files. Differential files allowed to store multiple versions, but they could not reduce cost of random read-writes in B-trees.

In 2014 another optimization of LSM-tree performance was proposed [8] on a hardware level. According to this paper, SSDs do not allow fully utilize I/O performance because of providing only one channel to an operating system even if SSD has multiple channels, which can be accessed simultaneously. On the base of custom LevelDB, it was showed, that usage of open-channel SSD (SDF) allows to improve

throughput in more than 4 times. But this optimization is actually hardware, and the LSM-tree is not changed, so on simple SSDs it does not work. But this method can be combined with deferred updates.

7. REFERENCES

[1] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[2] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006.

[3] E. Omiecinski, W. Liu, and I. Akyildiz. Analysis of a deferred and incremental update strategy for secondary indexes. *Informafion Systems*, 16(3):345–356, February 1991.

[4] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[5] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. In *Proceedings of the VLDB Endowment*, volume 10. VLDB Endowment, August 2017.

[6] S. Salas and E. Hille. *Calculus: One and Several Variable*. John Wiley and Sons, New York, 1978.

[7] D. G. Severance and G. M. Lohman. Differential files: their application to the maintenance of large databases. *ACM Transactions on Database Systems (TODS)*, 1(3):256–267, January 1976.

[8] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.