Efficient Range-Based Storage Management for Scalable Datastores

Giorgos Margaritis, and Stergios V. Anastasiadis

Abstract—Scalable datastores are distributed storage systems capable of managing enormous amounts of structured data for online serving and analytics applications. Across different workloads, they weaken the relational and transactional assumptions of traditional databases to achieve horizontal scalability and availability, and meet demanding throughput and latency requirements. Efficiency tradeoffs at each storage server often lead to design decisions that sacrifice query responsiveness for higher insertion throughput. In order to address this limitation, we introduce the novel *Rangetable* storage structure and *Rangemerge* method so that we efficiently manage structured data in granularity of key ranges. We develop a general prototype framework and implement several representative methods as plugins to experimentally evaluate their performance under common operating conditions. We experimentally conclude that our approach incurs range-query latency that is minimal and has low sensitivity to concurrent insertions, achieves insertion performance that approximates that of write-optimized methods under modest query load, and reduces down to half the reserved disk space.

Index Terms—Distributed systems, Storage Management, Performance, Measurements

1 Introduction

Scalable datastores (or simply datastores) are distributed storage systems that scale to thousands of commodity servers and manage petabytes of structured data. Today, they are routinely used by online serving, analytics and bulk processing applications, such as web indexing, social media, electronic commerce, and scientific analysis [1], [2], [3], [4], [5], [6], [7]. Datastores differ from traditional databases because they: (i) Horizontally partition and replicate the indexed data across many servers, (ii) Provide weaker concurrency model and simpler call interface, and (iii) Allow dynamic expansion of records with new attributes. Depending on the application needs, they organize data as collections of key-value pairs, multidimensional maps or relational tables.

System scalability across multiple servers is necessitated by the enormous amount of handled data and the stringent quality-of-service requirements [1], [2], [8]. Production systems keep the high percentiles of serving latency within tens or hundreds of milliseconds [2], [8]. General-purpose datastores target good performance on both read-intensive and write-intensive applications [1], [3]. Furthermore, applications that ingest and mine event logs accelerate the shift from reads to writes [9].

The data is dynamically partitioned across the available servers to handle failures and limit the consumed resources. To a large extent, the actual capacity, functionality and complexity of a datastore is determined by the architecture and performance of the constituent servers [10], [11], [12]. For instance, resource management efficiency at each storage server translates into fewer hardware components and lower maintenance cost for power consumption, redundancy and administration time. Also, support of a missing feature (e.g., range queries) in the storage server may require substantial reorganization with overall effectiveness that is potentially suboptimal [13], [14].

A storage layer at each server manages the memory and disks to persistently maintain the stored items [12]. Across diverse batch and online applications, the stored data is typically arranged on disk as a dynamic collection of immutable, sorted files (e.g., Bigtable, HBase, Azure, Cassandra in Section 7, Hypertable [15]). Generally a query should reach all item files to return the eligible entries (e.g., in a range). As the number of files on disk increases, it is necessary to merge them so that query time remains under control. Datastores use a variety of file merging methods but without rigorous justification. For instance, Bigtable keeps bounded the number of files on disk by periodically merging them through *compactions* [1] (also HBase, Cassandra, LazyBase in Section 7, Anvil in Section 15.1). In the rest of the document we interchangeably use the terms merging and compaction.

Despite the prior indexing research (e.g., in relational databases, text search), datastores suffer from several weaknesses. Periodic compactions in the background may last for hours and interfere with regular query handling leading to latency spikes [12], [16], [17], [18], [19]. To avoid this problem, production environments schedule compactions on a daily basis, thus leaving fragmented the data for several hours [12]. Frequent updates in distinct columns of a table row further fragment the data (e.g., HBase) [6], [20]. When several files on a server store data with overlapping key ranges, query handling generally involves multiple I/Os to access all files that contain a key. Bloom filters are only applicable to single-

• *The authors are with the Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece*
E-mail: {gmargari,stergios}@cs.uoi.gr

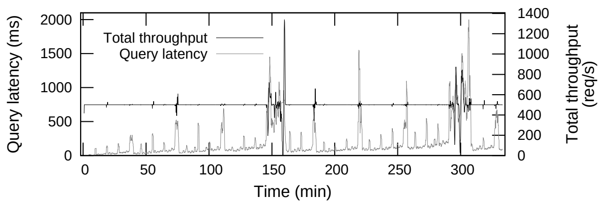


Fig. 1. The query latency at the Cassandra client varies according to a quasi-periodic pattern. The total throughput of queries and inserts also varies significantly.

Latency (ms) of Range Queries on Cassandra

#	Client			Server Storage Mgm		
Servers	Avg	90th	99th	Avg	90th	99th
1	204.4	420	2282	178.8	382	1906
4	157.6	313	1601	130.8	269	1066
8	132.2	235	1166	111.7	218	802

TABLE 1
Storage management on the server occupies more than 80% of the average query latency measured at the client.

key (but not range) queries, and have diminishing benefit at large number of files (e.g. 40) [12]. Finally, several merge-based methods require roughly half of the storage space to remain free during merging for the creation of new files [12].

In this work we study the storage management of online datastores that concurrently support both range queries and dynamic updates. Over inexpensive hardware we reduce the data serving latency through higher storage contiguity; improve the performance predictability with limited query-update interference and configurable compaction intensity; and decrease the storage space required for file maintenance through incremental compactions. Our main insight is to keep the data of the memory and disk sorted and partitioned across disjoint key ranges. In contrast to existing methods (e.g., Section 3), when incoming data fills up the available memory of the server, we only flush to disk the range that occupies the most memory space. We store the data of each range in a single file on disk, and split a range to keep bounded the size of the respective file as new data arrives at the server.

Our contributions can be summarized as follows: (i) Unified consideration of known solutions for datastore storage management across different research fields. (ii) Identification of several limitations in existing systems and introduction of the Rangetable structure and Rangemerge method to address them. (iii) Prototype development of a general framework to support alternative management methods as plugins. (iv) Comprehensive experimental study of existing methods and demonstrated superior performance of our solution over different storage devices and workload conditions.

In Sections 2 and 3, we experimentally motivate our study and outline previous known solutions. In Section 4 we specify our system assumptions and goals, while in Section 5 we introduce our solution and describe our prototype framework. In Section 6 we present our experimentation environment and results across different workloads. In Section 7 we compare our work with related literature, and in Section 8 we summarize our results and future work. In Sections 9-15 we provide additional assumptions, analytical and experimental evaluations, and differentiation from related research.

2 Motivation

In a distributed system, variability in the latency distribution of individual components is magnified at the service level; effective caching cannot directly address tail latency unless the entire working set of an application resides in the cache [21]. In this section, over a distributed datastore we experimentally demonstrate the query latency to vary substantially over time with a high percentage of it to be spent in the storage layer.

We use a cluster of 9 machines with the hardware configuration described in Section 6. We apply the Apache Cassandra version 1.1.0 as datastore with the default Size-Tiered compaction and the Yahoo! YCSB version 0.1.4 as workload generator [4], [5]. An item has 100B key length and 1KB value size. A range query requests a random number of consecutive items that is drawn uniformly from the interval [1,100]. Initially we run Cassandra on a single node. On a different machine, we use YCSB with 8 threads to generate a total of 500req/s out of which 99% are inserts and 1% are range queries. We disregarded much higher loads (e.g., 1000req/s) because we found them to saturate the server. The experiment terminates when a total of 10GB is inserted into the server concurrently with the queries.

For average size of queried range at 50 items, the generated read load is 250items/s, i.e., almost half the write load of 495items/s. An I/O on our hard disk takes on average 8.5-10ms for seek and 4.16ms for rotation. Accordingly the time to serve 5 range queries is 67.2ms, while the time to sequentially write 495 items is 21.9ms. Although the read time appears 3 times higher than that of the writes, the actual write load is practically higher as a result of the compactions involved.

In Fig. 1 we show the query latency measured every 5s and smoothed with a window of size 12 for clarity. The query latency varies substantially over time following some quasi-periodic pattern which is independent of the random query size. In fact, the latency variation approximates the periodicity at which the server flushes from memory to disk the incoming data and merges the created files. In the same figure, we additionally show the measured throughput of queries and inserts to also vary considerably over time, and actually drop to zero for 90 consecutive seconds at minutes 157 and 295.

We repeat the above experiment with Cassandra over 1, 4 and 8 server machines. We linearly scale the generated request rate up to 4000req/s and the inserted

The I/O Complexity of Datastore Storage Structures

Dynamic Data Structure	Insertion Cost	Query Cost	System Example
B-tree	$O(\log_B \frac{N}{M})$	$O(\log_B \frac{N}{M} + \frac{Z}{B})$	PNUTS [3], Dynamo [2]
Log-structured File System (LFS)	$O(\frac{1}{B})$	N/A	RAMCloud [22], FAWN [23]
Log-structured Merge Tree (LSM-tree), Geometric, r-COLA	$O(\frac{r}{B} \log_r \frac{N}{M})$	$O(\log_r \frac{N}{M} + \frac{Z}{B})$	HBase [12], Anvil [16], Azure [24], Bigtable [1], bLSM [9]
Geometric with p partitions, Remerge (special case for p=1)	$O(\frac{1}{B} \sqrt[p]{\frac{N}{M}})$	$O(p + \frac{Z}{B})$	bottom layer of SILT [25]
Stepped-Merge Algorithm (SMA), Sorted Array Merge Tree (SAMT), Nomerge (special case for $k = N/M$)	$O(\frac{1}{B} \log_k \frac{N}{M})$	$O(k \log_k \frac{N}{M} + \frac{Z}{B})$	Cassandra [4], GTSSL [12], Lucene [26]

TABLE 2
Summary of storage structures typically used in datastores. We include their I/O complexities for insertion and range query in one-dimensional search over single-key items.

dataset size up to 80GB, while we fix to 8 the number of YCSB threads at the client. We instrument the latency to handle the incoming query requests at each server. Table 1 shows the query latency respectively measured at the YCSB client and the storage layer of all the Cassandra servers. The difference mainly arises from time spent on network transfer, request redirection among the servers, and RPC handling. As we increase the number of servers, the query latency drops because the constant (8) number of YCSB threads results into reduced concurrency (and contention) per server in the cluster. Across different system sizes, the storage management accounts for more than 80% of the average latency and the 90th percentile, and more than 65% of the 99th percentile. Overall, compactions cause substantial latency variations, and storage management is dominant in the online performance of Cassandra-like datastores (Section 7). In the following sections we introduce a new storage structure and method to effectively control the compaction impact, and improve the datastore performance.

3 Background

Next we outline representative known methods for the problem of write-optimized data storage. We only consider external-memory data structures that handle one-dimensional range queries to report the points contained in a single-key interval. Thus we do not examine spatial access methods (e.g., R-tree, k-d-B-tree) that directly store multidimensional objects (e.g., lines) or natively handle multidimensional queries (e.g., rectangles). Spatial structures have not been typically used in datastores until recently [27]; also, at worst case, the lower-bound cost of orthogonal *search* in d dimensions (d>1) is fractional-power I/O for linear space and logarithmic I/O for nonlinear storage space [28].

A data structure is *static* if it remains searchable and immutable after it is built; it is *dynamic* if it supports both mutations and searches throughout its lifetime. The processing cost of a *static* structure refers to the total complexity to insert an entire dataset, and the insertion cost of a *dynamic* structure refers to the amortized complexity to insert a single item [29]. In a datastore, multiple static structures are often combined to achieve persistent data storage because filesystems over disk or flash devices are more efficient with appends rather than in-place writes [1], [22], [25].

Some datastores rely on the storage engine of a relational database at each server. For instance, PNUTS [3] uses the InnoDB storage engine of MySQL, and Dynamo [2] the BDB Transactional Data Store. In a relational database, data is typically stored on a B-tree structure. Let N be the total number of inserted items, B items the disk block size, and M items the memory size for caching the top levels of the tree. We assume unary cost for each block I/O transfer. One B-tree insertion costs $O(\log_B \frac{N}{M})$ and a range query of output size Z items costs $O(\log_B \frac{N}{M} + \frac{Z}{B})$ [30]. In contrast, the *Log-structured File System (LFS)* accumulates incoming writes into a memory-based buffer [31]. When the buffer fills up, data is transferred to disk in a single large I/O and deleted from memory. RAMCloud and FAWN use a logging approach for persistent data storage [22], [23].

Inspired from LFS, the *Log-Structured Merge-Tree (LSM-tree)* is a multi-level disk-based structure optimized for high rate of inserts/deletes over an extended period [32]. In a configuration with ℓ components, the first component is a memory-resident indexed structure (e.g., AVL tree), and the remaining components are modified B-trees that reside on disk. Component size is the storage space occupied by the leaf level. The memory and disk cost is minimized if the maximum size of consecutive components increases by a fixed factor r. When the size of a component C_i reaches a threshold, the leaves of C_i and C_{i+1} are merged into a new C_{i+1} component. The LSM-tree achieves higher insertion performance than a B-tree due to increased I/O efficiency from batching incoming updates into large buffers and sequential disk access during merges. The insertion cost of the LSM-tree is $O(\frac{r}{B} \log_r \frac{N}{M})$, where $\ell = \log_r \frac{N}{M}$ is the number of components. However, a range query generally requires to access all the components of an LSM-tree. Thus, a range query costs $O(\log_r \frac{N}{M} + \frac{Z}{B})$, if search is facilitated by a general technique called *fractional cascading* [30].

Bigtable and Azure rely on LSM-trees to manage persistent data [1], [12], [24].

The *Stepped-Merge Algorithm (SMA)* is an optimization of the LSM-tree for update-heavy workloads [33]. SMA maintains $\ell+1$ levels with up to k B-trees, called *runs*, at each level $i = 0, \ldots, \ell-1$, and 1 run at level ℓ. Whenever memory gets full, it is flushed to a new run on disk at level 0. When k runs accumulate at level i on disk, they are merged into a single run at level $i+1$, $i = 0, \ldots, \ell-1$. SMA achieves insertion cost $O(\frac{1}{B}\log_k \frac{N}{M})$, and query cost $O(k \log_k \frac{N}{M} + \frac{Z}{B})$ under fractional cascading. A compaction method based on SMA (with unlimited ℓ) has alternatively been called *Sorted Array Merge Tree (SAMT)* [12]. If we dynamically set $k = \frac{N}{M}$ to SMA, we get the *Nomerge* method, which creates new sorted files on disk without merging them [34]. Although impractical for searches, Nomerge is a baseline case for low index-building cost. A variation of SMA is applied with k=10 by the Lucene search engine [26], or k=4 by Cassandra and GTSSL [12].

Text indexing maps each term to a list of document locations (postings) where the term occurs. A merge-based method flushes postings from memory to a sorted file on disk and occasionally merges multiple files [35]. Along a sequence of created files, *Geometric* partitioning introduces the parameter r to specify an upper bound ($(r-1)r^{i-1}M$) at the size of the ith file, i = 1, 2, ..., for memory size M. Hierarchical merges guarantee similar sizes among the merged files and limit the total number of files on disk. The I/O costs of insertion and search in Geometric are asymptotically equal to those of the LSM-tree [30], [35] and the Cache-Oblivious Lookahead Array (COLA) [36]. Geometric can directly constrain the maximum number p of files with dynamic adjustment of r. Setting p=1 leads to the *Remerge* method, which always merges the full memory into a single file on disk and requires one I/O to handle a query [35]. A variation of Geometric with r=2 is used by Anvil [16] and r=3 by HBase [12]; SILT uses a single immutable sorted file (similar to p=1) on flash storage [25].

We summarize the asymptotic insertion and range-query costs of the above structures in Table 2. Log-based solutions achieve constant insertion cost, but lack efficient support for range queries. SMA incurs lower insertion cost but higher query cost than the LSM-tree. Geometric with p partitions takes constant time to answer a query, but requires fractional-power complexity for insertion. In our present study, we experimentally measure several of the above costs taking into consideration the interference among concurrent operations.

4 System Assumptions

We mainly target interactive applications of online data serving or analytics processing. The stored data is a collection of key-value pairs, where the key and the value are arbitrary strings of variable size from a few bytes up to several kilobytes. The system supports the operation of a *point query* as value retrieval of a single key, and a *range query* as retrieval of the values in a specified key range. Additionally, the system supports an *update* as insertion or full overwrite of a single-key value. We do not examine the problems of query handling over versioned data, or data loading in bulk.

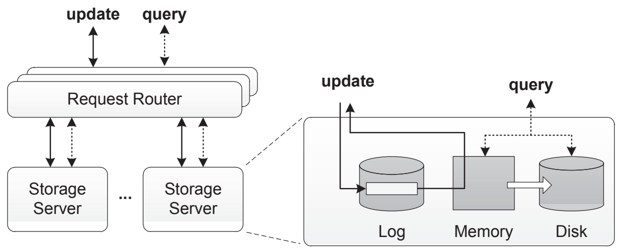


Fig. 2. Assumed datastore architecture.

A datastore uses a centralized or distributed index to locate the server of each stored item. Data partitioning is based on interval mapping for efficiency in handling range queries (Section 15.3). All accepted updates are made immediately durable through write-ahead logging [1], [2]. Thus, updates are commonly handled at sequential disk throughput, and queries involve synchronous random I/O. We focus on the storage functionality of individual servers rather than the higher datastore layers. The storage layer is implemented as a dynamic collection of immutable, sorted files. We require each point query to incur at most one disk I/O operation, and each range query to incur one I/O operation only increased by the extra sequential transfer time involved. Fig. 2 illustrates the path of an update or query through the request router and the storage servers, before returning the respective response back to the datastore client.

With data partitioning, each storage server ends up locally managing up to a few terabytes. The data is indexed by a memory-based sparse index, i.e., a sorted array with pairs of keys and pointers to disk locations every few tens or hundreds of kilobytes. For instance, Cassandra indexes 256KB blocks, while Bigtable, HBase and Hypertable index 64KB blocks [1], [4]. With a 100B entry for every 256KB, we need 400MB of memory to sparsely index 1TB. Compressed trees can reduce the occupied memory space by an order of magnitude at the cost of extra decompression processing [25]. We provide additional details about our assumptions in Section 10.

5 Design and Implementation

In the present section we propose a novel storage layer to efficiently manage the memory and disks of datastore servers. Our design sets the following primary goals: (i) Provide sequential disk scans of sorted data to queries and updates, (ii) Store the data of each key range at a single disk location, (iii) Selectively batch updates and free memory space, (iv) Avoid storage fragmentation or reorganization and minimize reserved storage space. Below, we describe the proposed Rangetable structure

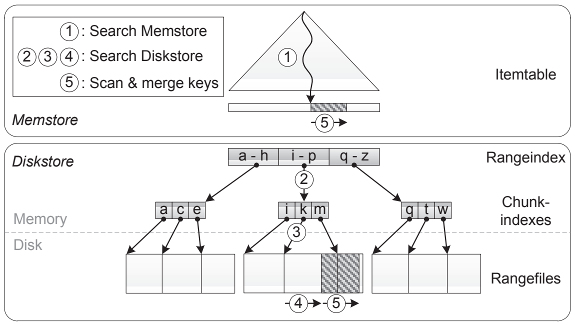


Fig. 3. The organization of the Rangetable structure, and control flow of a handled range query. For presentation clarity we use alphabetic characters as item keys.

and the accompanying Rangemerge method. Then we outline the prototype software that we developed to fairly compare our approach with representative storage structures of existing systems.

5.1 The Rangetable Structure

The main insight of Rangetable is to keep the data on disk in key order, partitioned across large files by key range. We store the data of a range at a single file to avoid multiple seeks for a point or range query. The disk blocks of a file are closely located in typical filesystems, with allocators based on block groups or extents (e.g., ext3/4, Btrfs). If the size of a data request exceeds a few MBs, the disk geometry naturally limits the head movement overhead to below 10%. For instance, if the average rotation and seek take 6.9ms in a 10KRPM SAS drive, the overhead occupies 8.6% of the total time to access 10MB [37]. We do not need enormous files to achieve sequential I/O, as long as each file has size in the tens of megabytes. We avoid frequent I/O by gathering incoming updates in memory, and inexpensively preserve range contiguity on disk by only flushing those ranges that ensure I/O efficiency.

New updates at a server are durably logged, but also temporarily accumulated in memory for subsequent batched flushing to their rangefile on disk (Fig. 2). For fast key lookup and range scan, we keep the data in memory sorted through a mapping structure, called *itemtable* (Fig. 3). We use a concurrent balanced tree (e.g., red-black tree) for this purpose, although a multicore-optimized structure is preferable if the stored data fully resides in memory [10]. For effective I/O management, we partition the data of every server into key-sorted ranges using a memory-based table, called *rangeindex*. Each slot of the rangeindex points to the respective items stored on disk.

We avoid external fragmentation and periodic reorganization on disk by managing the space in files, called *rangefiles*, of maximum size F (e.g., 256MB). Each rangefile is organized as a contiguous sequence of *chunks* with fixed size C (e.g., 64KB). In order to easily locate the rangefile chunks, we maintain a memory-based

Algorithm 1 The RANGEMERGE method
Input: Rangetable with memory size $>= M$
Output: Rangetable with memory size $< M$

```
1: {Victimize a range}
2: $R$ := range whose items occupy max total memory
3: {Flush memory items of $R$ to its rangefile $f_R$}
4: Load empty buffer $b_R$ with contents of rangefile $f_R$
5: Merge $b_R$ with itemtable data of $R$ into empty buffer $b'_R$
6: $v := \lceil \text{sizeof}(b'_R)/F \rceil$
7: Allocate $v$ new rangefiles $f_R^{(1)}, \ldots, f_R^{(v)}$ on disk
8: Split $b'_R$ into $v$ subranges $R^{(1)}, \ldots, R^{(v)}$ of equal size
9: Transfer subranges to respective $f_R^{(1)}, \ldots, f_R^{(v)}$
10: Build chunkindexes for $f_R^{(1)}, \ldots, f_R^{(v)}$
11: Update rangeindex with entries for $R^{(1)}, \ldots, R^{(v)}$
12: {Clean up memory and disk}
13: Free buffers $b_R$, $b'_R$, and itemtable entries of $R$
14: Delete rangefile $f_R$ and its chunkindex
```

sparse index per rangefile, called *chunkindex*, with entries the first key of each chunk and the offset within the rangefile. From the steps shown in Fig. 3, an incoming range query (1) traverses the itemtable in the memstore. Concurrently with step (1), the query searches (2) the rangeindex, (3) the chunkindex and (4) the rangefile of the diskstore. Finally, (5) the requested items from both the itemtable and rangefile are sorted into a single range by the server and returned.

5.2 The Rangemerge Method

In order to serve point and range queries with roughly one disk I/O, the Rangemerge method merges items from memory and disk in range granularity. When we merge items, we target to free as much memory space as possible at minimal flushing cost. The choice of the flushed range affects the system efficiency in several ways: (i) Every time we flush a range, we incur the cost of one rangefile read and write. The more new items we flush, the higher I/O efficiency we achieve. (ii) A flushed range releases memory space that is vital for accepting new updates. The more space we release, the longer it will take to repay the merging cost. (iii) If a range frequently appears in queries or updates, then we should skip flushing it to avoid repetitive I/O.

Memory flushing and file merging are generally regarded as two distinct operations. When memory fills up with new items, the server has to free memory space quickly to continue accepting new updates. Existing systems sequentially transfer to disk the entire memory occupied by new items. Thus, they defer merging to avoid blocking incoming updates for extended time period. This approach has the negative effect of increasing the files and incurring additional I/O traffic to merge the new file with existing ones [9]. To avoid this extra cost, Rangemerge treats memory flushing and file merging as a single operation rather than two. It also limits the duration of update blocking because a range has configurable maximum size, typically a small fraction of the occupied memory at the server (Section 14).

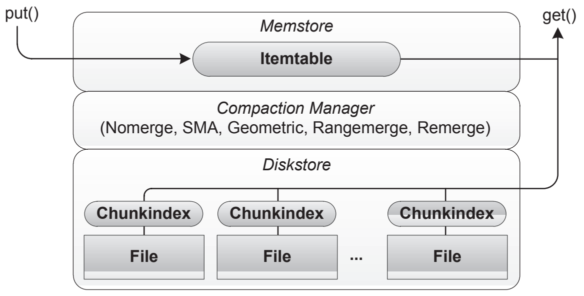


Fig. 4. Prototype framework with several compaction methods as plugins.

We greedily victimize the range with largest amount of occupied memory space. The intuition is to maximize the amount of released memory space along with the I/O efficiency of the memory flush. For simplicity, we take no account of the current rangefile size, although this parameter affects the merging cost, and the probability of having future I/O requests to a particular range. Despite its simplicity, this victimization rule has proved robust across our extensive experimentation. In Section 9 we show that Rangemerge has the asymptotic I/O cost of Remerge (Table 2), but in Sections 6.3 and 12 we experimentally demonstrate Rangemerge to approach the insertion time of Geometric (and even Nomerge) under various conditions (concurrent gets, sufficient memory, or skewed distribution).

The pseudocode of Rangemerge appears in Algorithm 1. The server receives items in the key interval assigned by the datastore index. We insert new items until the occupied memory space reaches the *memory limit* M. At this point, we pick as *victim* R the range of maximum memory space (line 2), read rangefile f_R from disk, merge it with the items of R in memory, and move the merged range back to disk (lines 4-11). The addition of new items may lead the size of range R to exceed the rangefile capacity F. In that case, we equally split R into v subranges (line 6) and move the data to v new rangefiles on disk (line 9). Finally, we free the itemtable space occupied by R, and delete the old rangefile from the disk (lines 13-14). Practically, flushing a single range is sufficient to reduce the occupied memory below the memory limit.

5.3 System Prototype

We developed a general storage framework to persistently manage key-value items over local disks. The interface supports the `put(k,v)` call to insert the pair `(k,v)`, the `get(k)` call to retrieve the value of key `k`, the `get(k,n)` call to retrieve `n` consecutive records from key $\geq$`k`, and the `get(`k_1`,`k_2`)` call to retrieve the records with keys in the range $[k_1, k_2]$. A get request returns the exact specified set of entries, rather than an ordered subset of them that would raise issues of result accuracy. Our prototype adopts a multithreaded approach to support the concurrent execution of queries and updates, and it is designed to easily accept different compaction methods as pluggable modules. The implementation consists of three main components, namely the *Memstore*, the *Diskstore*, and the *Compaction manager* (Fig. 4).

The Memstore uses a thread-safe red-black tree in memory to maintain incoming items in sorted order, and the Diskstore accesses each sorted file on disk through a sparse index maintained in memory. The Compaction manager implements the file merging sequences of the following methods: Nomerge, SMA, Geometric, Rangemerge and Remerge. We implemented the methods using C++ with the standard template library for basic data structures and 3900 uncommented lines of new code.

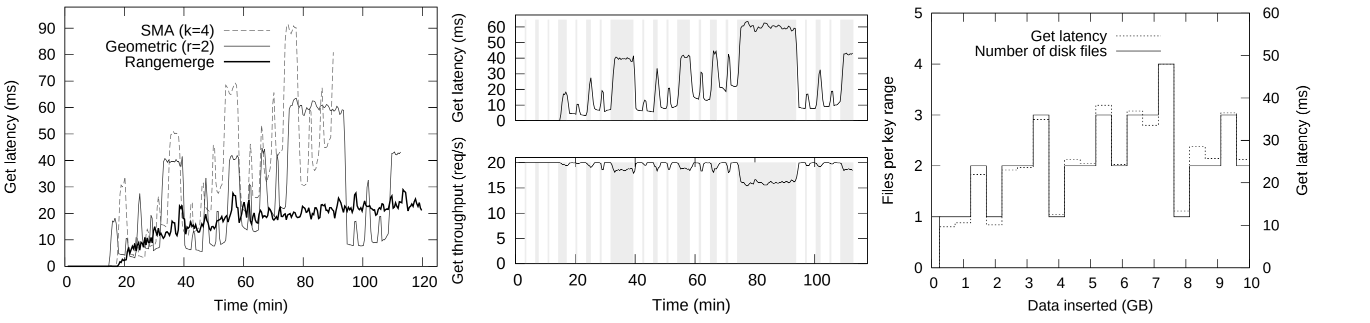
6 Performance Evaluation

We claim that Rangemerge achieves minimal query latency of low sensitivity to the I/O traffic from concurrent compactions, and approximates or even beats the insertion time of write-optimized methods under various conditions. In the present section, we experimentally evaluate the query latency and insertion time across several compaction methods. In Sections 11-14 we validate our prototype against Cassandra, and examine the performance sensitivity to various workload parameters and storage devices. Although not explicitly shown, Rangemerge also trivially avoids the 100% overhead in storage space of other methods [12].

6.1 Experimentation Environment

We did our experiments over servers running Debian Linux 2.6.35.13. Each machine is equipped with one quad-core 2.33GHz processor (64-bit x86), one activated gigabit ethernet port, and two 7200RPM SATA2 disks. Unless otherwise specified, we configure the server RAM equal to 3GB. Each disk has 500GB capacity, 16MB buffer size, 8.5-10ms average seek time, and 72MB/s sustained transfer rate. Similar hardware configuration has been used in a recent related study [38]. We store the data on one disk over the Linux ext3 filesystem. In Rangemerge we use rangefiles of size F=256MB. We also examine Remerge, Nomerge, Geometric (r=2, r=3, or p=2) and SMA (k=2 or k=4, with unlimited ℓ). In all methods we use chunks of size C=64KB. From Section 3, variations of these methods are used by Bigtable, HBase (Geometric, r=3), Anvil and bLSM (Geometric, r=2), GTSSL (SMA, k=4), and Cassandra (SMA, k=4).

We use YCSB to generate key-value pairs of 100 bytes key and 1KB value. On one server we insert a dataset of 9.6M items with total size 10GB. Similar dataset sizes per server are typical in related research (e.g., 1M [3], 9M [19], 10.5GB [38], 16GB [12], 20GB [5]). The 10GB dataset size fills up the server buffer several times (e.g., 20 for 512MB buffer space) and creates interesting compaction activity across the examined algorithms. With larger datasets we experimentally found the server behavior to remain qualitatively similar, while enormous datasets are typically partitioned across multiple servers.



(a) Get latency in three methods (b) Compactions in Geom (r=2) (c) Number of files in Geom (r=2)

Fig. 5. During concurrent inserts and queries, (a) the get latency of Geometric (r=2) and SMA (k=4) has substantially higher variability and average value than Rangemerge, and (b) the get throughput of Geometric (r=2) drops as low as 15.5req/s during compactions (grey background). (c) At the insertion of 10GB with M=512MB using Geometric partitioning (r=2), get latency (at load 10req/s) is closely correlated to the number of files created.

For experimentation flexibility and due to lack of public traces [39], we use synthetic datasets with keys that follow the uniform distribution (default), Zipfian distribution, or are partially sorted [5]. We take average measurements every 5s, and smooth the output with window size 12 (1-min sliding window) for readability. Our default range query reads 10 consecutive items.

The memory limit M refers to the memory space used to buffer incoming updates. Large system installations use dynamic assignment to achieve load balancing by having number of service partitions (*micro-partitions*) that is much larger than the number of available machines [21]. For instance, the Bigtable system stores data in tablets with each machine managing 20 to 1,000 tablets. Consequently, the default buffer space per tablet lies in the range 64-256MB [40]. Other related research configures the memory buffer with size up to several GB [5], [12]. As a compromise between these choices, we set the default memory limit equal to M=512MB; thus we keep realistic (1/20) the ratio of memory over the 10GB dataset size and ensure the occurrence of several compactions throughout an experiment. In Section 6.3 we examine memory limit and dataset size up to 4GB and 80GB respectively. We further study the performance sensitivity to memory limit M in Fig. 13.

6.2 Query Latency and Disk Files

First we measure the query latency of a mixed workload with concurrent puts and gets. An I/O over our disk takes on average 13.4ms allowing maximum rate about 74req/s (can be higher for strictly read workloads). We configure the get load at 20req/s so that part of the disk bandwidth can be used by concurrent compactions. We also set the put rate at 2500req/s, which is about half of the maximum possible with 20get/s (shown in Fig. 10c). The above combined settings occupy roughly two thirds of the total disk bandwidth and correspond to a write-dominated workload (get/put ratio about 1/100 in operations and 1/25 in items) [38]. We examine other combinations of put and get loads in Section 12.

We assume that when memory fills up, the put thread is blocked until we free up memory space. Although write pauses can be controlled through early initiation of memory flushing [9], their actual effect to insertion performance additionally depends on the flushing granularity and duration (explored in Section 14). In order to determine the concurrency level of query handling in the server, we varied the number of get threads between 1 and 20; then we accordingly adjusted the request rate per thread to generate total get load 20req/s. The measured get latency increased with the number of threads, but the relative performance difference between the methods remained the same. For clarity, we only illustrate measurements for one put and one get thread.

In Fig. 5a we examine three representative methods: SMA (k=4), Geometric (r=2), and Rangemerge. The experiment runs separately for each method until loading 10GB. The get latency of Rangemerge (avg: 15.6ms, std: 8.2ms, max: 30.5ms) has lower average value by 51-83% and standard deviation by 2.5-3 times than Geometric (avg: 23.5ms, std: 20.5ms, max: 64.5ms) and SMA (avg: 28.6ms, std: 24.3ms, max: 93.3ms). Also Remerge (not shown) is less responsive and predictable (avg: 21.1ms, std: 10.6ms, max: 35.4ms) than Rangemerge. However, SMA reduces the experiment duration to 90min from 119min required by Rangemerge and 112min by Geometric (see also Fig. 13). In Fig. 5b we illustrate the get performance of Geometric, with concurrent compactions as vertical grey lanes. Compactions increase latency by several factors and reduce throughput from 20req/s to 15.5req/s. The throughput of SMA (not shown) also drops to 10.4req/s, unlike the Rangemerge throughput that remains above 17.4req/s.

In Fig. 5c we depict the number of files (left y axis) and the average get latency (right y axis) for Geometric. After every compaction, we measure the get latency as average over twenty random requests. From every file, the get operation reads the items of the requested key range. Assuming no concurrent compactions, the measured latency varies between 11.9ms and 49.0ms,

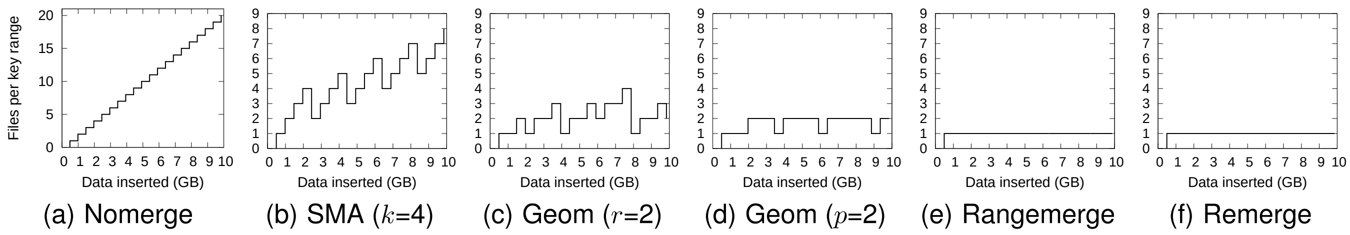


Fig. 6. We show the number of files that store the data across different compaction methods with M=512MB for 10GB dataset size. In the case of Rangemerge, a single file strictly stores the data of a rangefile range.

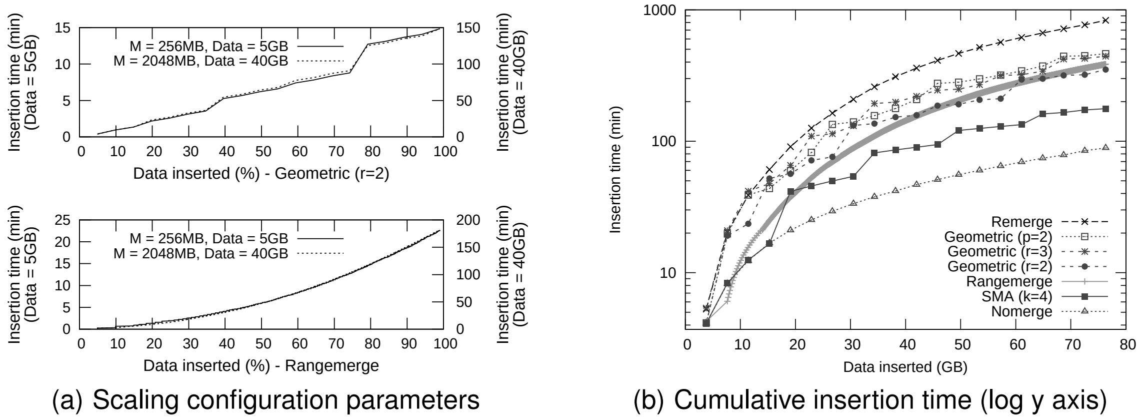


Fig. 7. (a) The insertion progress is similar between the configuration of M=256MB with 5GB dataset (left y-axis) and M=2GB with 40GB (right y-axis). (b) Cumulative insertion with Remerge takes longer by several factors than Geometric, while Rangemerge lies between Geometric (r=2) and SMA (k=4) with M=4GB for 80GB dataset size.

as the number of files per key varies between 1 and 4. The evident correlation between get latency and the number of files in Geometric explains the variation of get performance between compactions in Fig. 5b.

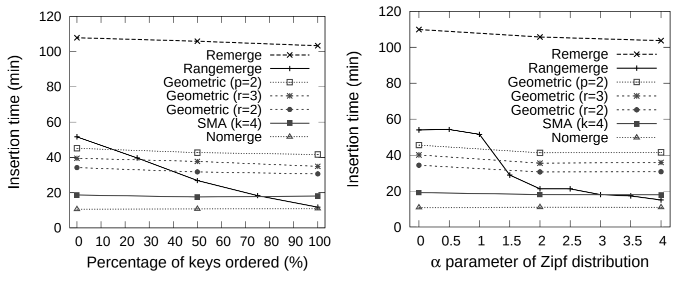
We further explore this issue in Fig. 6, where we show the number of maintained files as function of the dataset size. Nomerge increases the number of sorted files up to 20 (only limited by the dataset size), and SMA (k=4) increases the number of created files up to 8. Geometric with r=2 and p=2 varies the number of files up to 4 and 2, respectively. Instead, Remerge always maintains a single file for the entire dataset, while Rangemerge strictly stores on a single file the items of a rangefile range; both methods lead to roughly one random I/O per get operation. Overall Rangemerge leads to more responsive and predictable get operations with concurrent puts.

6.3 Insertion Time

Next we study the cumulative latency to insert data items one-by-one into the storage server. Insertion includes some processing to sort the data in memory, but mainly involves I/O to flush data and apply compactions over the disk files. In order to ensure the generality of our results, we measured the total insertion time at different scales of dataset size and memory limit. In Fig. 7a the cumulative insertion time of Geometric (and Rangemerge) forms a similar curve as long as the ratio of dataset size over memory limit is constant (e.g., 5GB/256MB=40GB/2GB=20). We confirmed this behavior across several parameter scales that we examined.

In Fig. 7b (with log y axis) we examine the time required to insert a 80GB dataset with M=4GB over a server with 6GB RAM. Nomerge takes 1.5hr to create 20 files on disk, and SMA (k=4) spends 2.9hr for 8 files. Geometric takes 5.8hr with r=2, 7.4hr with r=3, and 7.7hr with p=2 for up to 2 files. Remerge requires 13.9hr to maintain 1 file on disk, and Rangemerge takes 6.4hr. In general the smaller the number of disk files with overlapping key ranges, the longer it takes to insert the dataset. One exception is Rangemerge that requires half the insertion time of Remerge to effectively store the keys of each rangefile range contiguously at one disk location. Geometric (r=2) reduces the insertion time of Rangemerge by 10%, but requires up to 4 random I/Os for a query (Fig. 6c). We further study memory in Fig. 13.

In Fig. 8a we investigate how insertion time is affected by the percentage of keys inserted in sorted order. Rangemerge approaches Nomerge as the percentage of sorted keys increases from 0% (uniform distribution) to 100% (fully sorted). This behavior is anticipated because the sorted order transforms merges to sequential writes with fewer reads. In Fig. 8b we draw the inserted keys from a Zipfian distribution and study the impact of parameter α to the insertion time. The higher we set the parameter α, the more items appear at the head (popular part) of the distribution. Rangemerge exploits the higher item popularity to again approximate Nomerge.



(a) Ordered keys (b) Zipfian keys

Fig. 8. Sensitivity of insertion time to key distribution, as we generate put requests back-to-back with zero get load.

7 Related Work

Next we present previous research related to datastores (further explored in Section 15).

7.1 Datastores

Bigtable is a structured storage system that partitions data across multiple storage servers with the incoming data logged to disk and kept in server memory [1]. HBase is an open-source variation of Bigtable; it sorts files by age and merges an older file of size within twice the size of the newer file, or within the aggregate size of all newer files [5], [6]. Azure partitions data by key range across different servers [24]. Each partition is stored over multiple checkpoint files whose number is kept small through periodic merging. Dynamo stores key-value pairs over a distributed hash table, and accepts pluggable persistent components for local storage [2]. Cassandra combines the data model of Bigtable with the partitioning scheme of Dynamo, and merges into a single file the sorted files of similar size [4], [20].

With an emphasis on analytics, LazyBase combines update batching with pipelining and trades read freshness for performance. Tree-based merging is triggered by the number of leaves or a time period [14]. RAMcloud uses a log-structured approach to manage data on both memory and disk for fast crash recovery [22]. For low power consumption, the FAWN key-value store uses a log file on flash storage indexed by an in-memory hash table [23]. The SILT key-value store introduces space-efficient indexing and filtering in memory, and it maintains a log file, multiple hash tables and a sorted file on flash storage [25]. HyperDex is a distributed key-value store that supports multidimensional search through space partitioning into non-overlapping subregions and linear scan over log files [27]. In this study we comparatively consider a representative set of storage management techniques from the above systems.

7.2 Storage Structures

For the specialized needs of full-text search, we previously studied the online maintenance of inverted files over fixed-sized, disk blocks [41]. A different study introduced the BR-tree, which integrates Bloom filters into a multidimensional data structure (R-tree). Over multidimensional data items, the BR-tree has been shown to efficiently support different types of complex queries, including point and range queries [42].

For intense update loads and analytics queries, the partitioned exponential file (PE file) dynamically partitions data into distinct key ranges and manages separately each partition similarly to an LSM-tree [43]. Insertion cost varies significantly due to the required storage reorganization and data merging within each partition. Search cost varies because it involves all levels of a partition, uses tree indexing at each level, and interferes with concurrent insertions.

The bLSM-tree applies application backpressure to control write pauses in a three-level LSM-tree, but it cannot gracefully cope with write skew or short scans [9]. Bender et al. introduce the cache-oblivious lookahead array (g-COLA) as a multi-level structure, where g is the factor of size growth between consecutive levels [36]. Due to buffering and amortized I/O, g-COLA achieves faster random inserts than a traditional B-tree, but slower searches and sorted inserts. In our experiments we included Geometric partitioning as a variation of g-COLA (Section 3).

7.3 Metadata Management

Spyglass uses index partitioning, incremental crawling, versioning and signature files to support complex metadata searches over large-scale storage systems [44]. Range and top-k queries over multidimensional metadata attributes are efficiently supported by aggregating correlated files into semantic-aware groups [45]. For the approximate processing of aggregate and top-k metadata queries in hierarchical filesystems, the Glance system combines fast sampling with bounded-variance estimation [46]. The VT-tree extends the write-optimized LSM-tree to index filesystems at reduced storage fragmentation and data copying [47]. Unlike the above studies that focus on exact or approximate search over filesystem metadata, this work investigates the storage management of structured data in scale-out datastores.

8 Conclusions and Future Work

After consideration of existing solutions in storage management of datastores, we point out several weaknesses related to high query latency, interference between queries and updates, and excessive reservation of storage space. To address these issues, we propose and analyze the simple yet efficient Rangemerge method and Rangetable structure. With a prototype implementation, we experimentally demonstrate that Rangemerge minimizes range query time, keeps low its sensitivity to compaction I/O, removes the need for reserved unutilized storage space, under various moderate conditions exceeds the insertion performance of practical write-optimized methods, and naturally exploits the key skewness of the inserted dataset. In our future work, we are primarily interested to incorporate Rangemerge into a multi-tier datastore and handle multi-versioned data.

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *USENIX OSDI Symp.*, Seattle, WA, 2006, pp. 205–220.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP Symp.*, Stevenson, WA, October 2007, pp. 205–220.

[3] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," in *VLDB Conf.*, Auckland, New Zealand, August 2008, pp. 1277–1288.

[4] E. Hewitt, *Cassandra: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, Inc., 2011.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM SOCC Symp.*, Indianapolis, IN, Jun. 2010, pp. 143–154.

[6] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache Hadoop goes realtime at facebook," in *ACM SIGMOD Conf.*, Athens, Greece, Jun. 2011, pp. 1071–1080.

[7] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with Project Voldemort," in *USENIX FAST*, San Jose, CA, Feb. 2012, pp. 223–236.

[8] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The SCADS director: Scaling a distributed storage system under stringent performance requirements," in *USENIX FAST Conf.*, San Jose, CA, Feb. 2011, pp. 163–176.

[9] R. Sears and R. Ramakrishnan, "bLSM: a general purpose log structured merge tree," in *ACM SIGMOD Conf.*, Scottsdale, AZ, May 2012, pp. 217–228.

[10] Y. Mao, E. Kohler, and R. Morris, "Cache craftiness for fast multicore key-value storage," in *ACM EuroSys Conf.*, Apr. 2012, pp. 183–196.

[11] M. Stonebraker and R. Cattell, "10 rules for scalable performance in 'simple operation' datastores," *Commun. ACM*, vol. 54, no. 6, pp. 72–80, Jun. 2011.

[12] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, and S. Archak, "An efficient multi-tier tablet server storage architecture," in *ACM SOCC Symp.*, Cascais, Portugal, Oct. 2011, pp. 1–14.

[13] P. Pirzadeh, J. Tatemura, O. Po, and H. Hacigümüs, "Performance evaluation of range queries in key value stores," *J. Grid Computing*, vol. 10, no. 1, pp. 109–132, 2012.

[14] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. N. Soules, and A. Veitch, "Lazybase: Trading freshness for performance in a scalable database," in *ACM EuroSys Conf.*, Bern, Switzherland, Apr. 2012, pp. 169–182.

[15] http://hypertable.org.

[16] M. Mammarella, S. Hovsepian, and E. Kohler, "Modular data storage with Anvil," in *ACM SOSP Symp.*, Big Sky, MO, 2009, pp. 147–160.

[17] R. Low, "Cassandra under heavy write load," http://www.acunu.com/blogs/richard-low/, Acunu, Ltd., London, UK, Mar. 2011.

[18] "Leveldb: A fast and lightweight key/value database library by google," http://code.google.com/p/leveldb/, May 2011.

[19] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: benchmarking and performance debugging advanced features in scalable table stores," in *ACM SOCC Symp.*, Cascais, Portugal, 2011, pp. 1–14.

[20] J. Ellis, "Leveled compaction in Apache Cassandra," http://www.datastax.com/dev/blog/, DataStax, San Mateo, CA, Jun. 2011.

[21] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[22] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *ACM SOSP Symp.*, Cascais, Portugal, Oct. 2011, pp. 29–41.

[23] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," in *ACM SOSP Symp.*, Big Sky, MO, Oct. 2009, pp. 1–14.

[24] B. Calder, J. Wang, A. Ogus, N. Nilakantan, and A. Skjolsvold et al., "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *ACM SOSP Symp.*, Cascais, Portugal, Oct. 2011, pp. 143–157.

[25] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *ACM SOSP Symp.*, Cascais, Portugal, Oct. 2011, pp. 1–13.

[26] D. Cutting, "Open source search," http://www.scribd.com/doc/18004805/Lucene-Algorithm-Paper, 2005.

[27] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *ACM SIGCOMM Conf.*, Helsinki, Finland, Aug. 2012, pp. 25–36.

[28] J. S. Vitter, "External memory algorithms and data structures: dealing with massive data," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209–271, Jun. 2001.

[29] J. L. Bentley and J. B. Saxe, "Decomposable searching problems i. static-to-dynamic transformation," *Journal of Algorithms*, vol. 1, pp. 301–358, 1980.

[30] K. Yi, "Dynamic indexability and lower bounds for dynamic one-dimensional range query indexes," in *ACM PODS Symp.*, Providence, RI, Jul. 2009, pp. 187–196.

[31] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, pp. 351–385, June 1996.

[33] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *VLDB*, Athens, Greece, Aug. 1997, pp. 16–25.

[34] S. Büttcher, C. L. A. Clarke, and B. Lushman, "Hybrid index maintenance for growing text collections," in *ACM SIGIR Conf.*, Seattle, WA, Aug. 2006, pp. 356–363.

[35] N. Lester, A. Moffat, and J. Zobel, "Efficient online index construction for text databases," *ACM Trans. Database Systems (TODS)*, vol. 33, no. 3, pp. 1–33, Aug. 2008.

[36] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious streaming B-trees," in *ACM SPAA Symp.*, San Diego, CA, Jun. 2007, pp. 81–92.

[37] "Savvio 10k.5 data sheet: The optimal balance of capacity, performance and power in a 10k, 2.5 inch enterprise drive," Seagate Tech LLC, 2012.

[38] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Goméz-Villamor, V. Muntés-Mulero, and S. Mankovskii, "Solving big data challenges for enterprise application performance management," in *VLDB Conf.*, Instanbul, Turkey, Aug. 2012, pp. 1724–1735.

[39] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Conf.*, London, UK, Jun. 2012, pp. 53–64.

[40] T. Dory, B. Mejias, P. V. Roy, and N.-L. Tran, "Measuring elasticity for cloud databases," in *IARIA Intl Conf Cloud Computing, GRIDs, and Virtualization*, Rome, Italy, Sep. 2011, pp. 154–160.

[41] G. Margaritis and S. V. Anastasiadis, "Low-cost management of inverted files for online full-text search," in *ACM CIKM*, Hong Kong, China, Nov. 2009, pp. 455–464.

[42] Y. Hua, B. Xiao, and J. Wang, "BR-Tree: A scalable prototype for supporting multiple queries of multidimensional data," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1585–1598, Dec. 2009.

[43] C. Jermaine, E. Omiecinski, and W. G. Yee, "The partitioned exponential file for database storage management," *The VLDB Journal*, vol. 16, pp. 417–437, October 2007.

[44] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *USENIX FAST Conf.*, San Francisco, CA, feb 2009, pp. 153–166.

[45] Y. Hua, H. Jiang, Y. Zhu, , D. Feng, and L. Tian, "Semantic-aware metadata organization paradigm in next-generation file systems," *IEEE TPDS*, vol. 23, no. 2, pp. 337–344, Feb. 2012.

[46] H. H. Huang, N. Zhang, W. Wang, G. Das, and A. S. Szalay, "Just-in-time analytics on large file systems," *IEEE Trans. Comput.*, vol. 61, no. 11, pp. 1651–1664, 2012.

[47] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees," in *USENIX FAST Conf.*, San Jose, CA, Feb. 2013, pp. 17–30.

Giorgos Margaritis is currently Doctoral Candidate in the Department of Computer Science and Engineering, University of Ioannina, Greece. Previously he received BSc ('05) and MSc ('08) degrees from the Department of Computer Science, University of Ioannina, Greece. His research interests include search and storage management for text and structured data on distributed systems.

Stergios V. Anastasiadis is Assistant Professor in the Department of Computer Science and Engineering, University of Ioannina, Greece. Previously he served as Visiting Professor (2009-10) at EPFL, Switzerland and Visiting Assistant Professor (2001-2003) at Duke University, USA. He received MSc ('96) and PhD ('01) degrees in Computer Science from the University of Toronto, Canada. His research interests include operating systems, distributed systems and performance evaluation.

9 The I/O Complexity of Rangemerge

We aim to estimate the total amount of bytes transferred between memory and disk during the insertion of N items to the Rangetable with Rangemerge. For simplicity each item is assumed to occupy one byte. Since the rangefile size is roughly $0.5F$ after a split and cannot exceed F by design, on average it is equal to $0.75F$. Accordingly each merge operation transfers on average a total of $c_{merge} = 1.5F$ bytes, as it reads a rangefile, updates it, and writes it back to disk. For the insertion of N items, the total amount of transferred bytes is equal to $C_{total} = K \cdot c_{merge} = K \cdot \frac{3F}{2}$, where K is the number of merges. In order to estimate an upper bound on C_{total} we assume an insertion workload that maximizes K.

We call *epoch* a time period during which no range split occurs, leaving unmodified the number of ranges (and rangefiles). Let E be the number of epochs involved in the insertion of N items, and k_i be the number of merges during epoch e_i, $i = 1, ..., E$. Then the total number of merges becomes equal to $K = \sum_{i=1}^{E} k_i$.

When memory fills up for first time, there is a single range in memory and no rangefile on disk. The first merge operation transfers all memory items to $r_1 = M/0.5F$ half-filled rangefiles, where r_i is the number of rangefiles (or ranges) during the ith epoch.

The next time memory fills up, we pick to merge the largest range in memory. In order to maximize the number of merges, we minimize the number of items in the largest range through the assumption of uniformly distributed incoming data. Then the largest range has size $m_1 = M/r_1$ items. During the ith epoch, it follows that each merge transfers to disk a range of size $m_i = M/r_i$ items.

A split initiates a new epoch, therefore a new epoch increments the number of rangefiles by one: $r_i = r_{i-1} + 1 = r_1 + i - 1$. Due to the uniform item distribution, a larger number of ranges reduces the amount m_i of items transferred to disk per merge and increases the number k_i of merges for N inserted items. If we shorten the duration of the epochs, then the number of merges will increase as a result of the higher number of rangefiles.

At a minimum, a half-filled rangefile needs to receive $0.5F$ new items before it splits. Therefore the minimum number of merges during the epoch e_i is $k_i = 0.5F/m_i$. Since an epoch flushes $0.5F$ items to disk before a split occurs, it takes $E = N/0.5F$ epochs to insert N items. From C_{total}, K, E, m_i, r_i and r_1 we find:

$$\begin{aligned} C_{total} &= \frac{3F}{2} \cdot K = \frac{3F}{2} \cdot \sum_{i=1}^{E} k_i = \frac{3F^2}{4} \cdot \sum_{i=1}^{E} \frac{1}{m_i} \\ &= \frac{3F^2}{4M} \cdot \sum_{i=1}^{E} r_i = \frac{3F^2}{4M} \sum_{i=1}^{E} (r_1 + i - 1) \\ &= \frac{3F^2}{4M} \left(E \cdot r_1 + \frac{1}{2} E(E+1) - E \right) \\ &= N^2 \frac{6}{4M} + N \left(3 - \frac{3F}{4M} \right) \in O(\frac{N^2}{M}) \end{aligned}$$

If we divide $O(\frac{N^2}{M})$ by the amount of inserted items N and the block size B, the above result becomes the $O(\frac{N}{MB})$ per-item insertion I/O complexity of the Remerge method (Table 2).

The above analysis of Rangemerge estimates the number of I/O operations involved in the worst case during index building. However it does not account for the cost of an individual I/O operation or the interaction of insertion I/O operations with concurrent queries. Through extensive experimentation in Sections 6 and 12-14 we show that Rangemerge combines high performance in both queries and insertions because it achieves search latency comparable to or below that of the read-optimized Remerge and insertion performance comparable to that of the write-optimized methods (e.g., Geometric, Nomerge) under various conditions.

10 Practicalities and Limitations

In this section we describe important practical issues that we considered in our design and potential limitations resulting from our assumptions.

10.1 Queries

Range queries are often used by data serving and analytics applications [3], [5], [6], [10], [14], [15], [24], while time-range queries are applied on versioned data for transactional updates [48]. Accordingly, typical benchmarks support range queries in addition to updates and point queries as workload option [5], [19]. We do not consider Bloom filters because they are not applicable to range queries, and their effectiveness in point queries has been extensively explored previously; in fact, support for range queries can orthogonally coexist with Bloom filters [1].

We recognize that query performance is hard to optimize for the following reasons: (i) Service-level objectives are usually specified in terms of upper-percentile latency [2], [8]. (ii) Query performance is correlated with the number of files at each server [6], [12], [19]. (iii) The amortization of disk writes may lead to intense device usage that causes intermittent delay (or disruption) of normal operation [7], [16], [17], [18]. (iv) The diversity of supported applications requires acceptable operation across different distributions of the input data keys [5].

10.2 Updates

Incoming updates are inserted to the itemtable, and queries are directed to both the itemtable and the rangefiles. Although the itemtable supports concurrent updates at high rate, the rangeindex along with the rangefiles and their chunkindexes remain immutable between range merges. Every few seconds that Rangemerge splits a range and resizes the rangeindex, we protect the rangeindex with a coarse-grain lock. We find this approach acceptable because the rangeindex has relatively small size (in the order of thousands entries) and only takes limited time to insert a new range.

The enormous amount of I/O in write-intensive workloads has led to data structures that involve infrequent but demanding sequential transfers [32], [35]. Excessive consumption of disk bandwidth in maintenance tasks can limit interactive performance. Deamortization is a known way to enforce an upper bound to the amount of consecutive I/O operations at the cost of extra complexity to handle interrupted reorganizations [36]. Instead, Rangemerge naturally avoids to monopolize disk I/O by applying flush operations at granularity of a single range rather than the entire memory buffer and configuring the range size through the rangefile parameter F.

10.3 Availability and Recovery

Availability over multiple machines is generally achieved through data replication by the datastore itself or an underlying distributed filesystem [1], [2], [3]. Durability requirements depend on the semantics and performance characteristics of applications, while data consistency can be enforced with a quorum algorithm across the available servers [2]. We consider important the freshness of accessed data due to the typical semantics of online data serving [14]. For instance, a shopping cart should be almost instantly updated in electronic commerce, and a message should be made accessible almost immediately after it arrives in a mailbox.

At permanent server failure, a datastore recovers the lost state from redundant replicas at other servers. After transient failures, the server rebuilds index structures in volatile memory from the rangefiles and the write-ahead log. We normally log records about incoming updates and ranges that we flush to disk. Thus we recover the itemtable by replaying the log records and omitting items already flushed to rangefiles. Holding a copy of the chunkindex in the respective rangefile makes it easy to recover chunkindexes from disk. We also rebuild the rangeindex from the contents of the itemtable and the rangefiles. The log size should be limited to keep short the recovery time and control the occupied storage space. For that purpose, a background process periodically cleans the log from items which have been flushed to disk or permanently deleted [31].

10.4 Caching

It is possible to improve the query performance with data caching applied at the level of blocks read from disk or data items requested by users [1], [21]. We currently rely on the default page caching of the system without any sophistication related to file mapping or item caching. Prior research suggested the significance of data compaction regardless of caching [12]. We leave for future work the study of multi-level caching and dynamic memory allocation for the competing tasks of update batching and query data reuse.

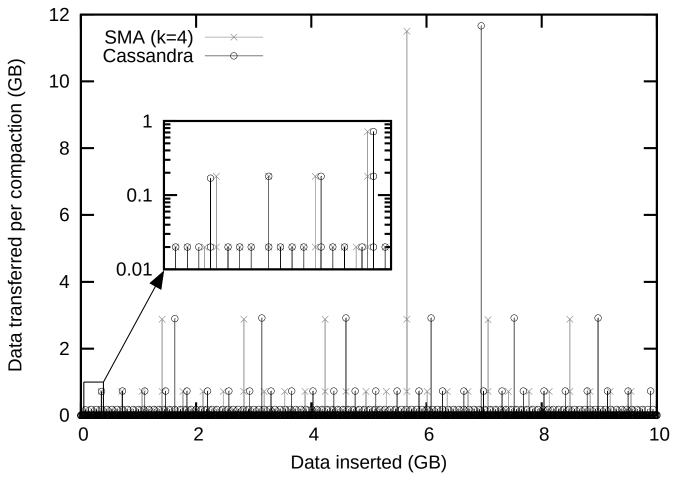
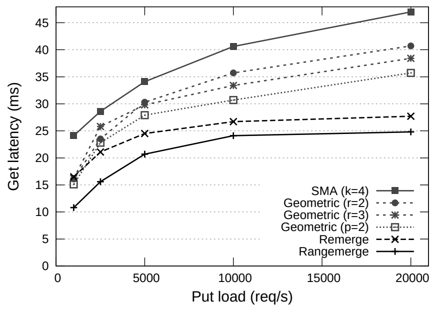


Fig. 9. We observe similar compaction activity between Cassandra and our prototype implementation of SMA (k=4). The height (y-axis value) of each mark denotes the transfer size of the respective compaction.

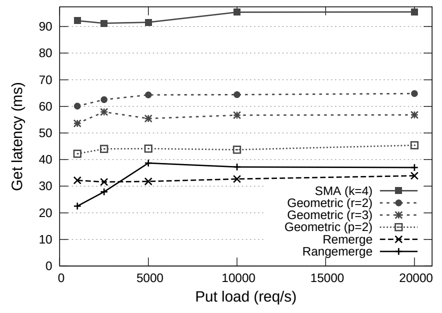
11 Prototype Validation

We validate the accuracy of our experimentation by comparing the compaction activity of our storage framework with that of Cassandra. From review of the published literature and the source code, we found that Cassandra implements a variation of the SMA (k=4) algorithm [4]. Accordingly, the stored data is organized into levels of up to k=4 files; every time the threshold of k=4 files is reached at one level, the files of this level are merged into a single file of the next level (Section 3). In our framework we set M=25MB because we found that Cassandra by default flushes to disk 25MB of data every time memory gets full. For comparison fairness we disable data compression and insertion throttling in Cassandra. We create the Cassandra workload using YCSB with 2 clients, which respectively generate puts at 500req/s and gets at 20req/s. The stored items are key-value pairs with 100B key and 1KB value, while the size of the get range is drawn uniformly from the interval [1,20]. The experiment terminates when 10GB of data is inserted. We generate a similar workload in our framework with two threads.

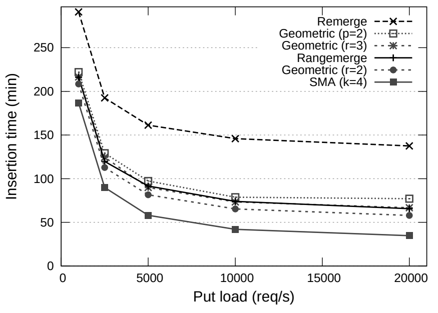
In Fig. 9 we show the amount of transferred data as we insert new items into the Cassandra and our prototype system respectively. The height of each mark refers to the total amount of transferred data during a compaction. Across the two systems we notice quasi-periodic data transfers of exactly the same size. In the case that a merge at one level cascades into further merges at the higher levels, in our prototype we complete all the required data transfers before we accept additional puts. Consequently it is possible to have multiple marks at the same x position. Instead Cassandra allows a limited number of puts to be completed between the cascading merges, which often introduces a lag between the corresponding marks. Overall the two systems transfer equal amount



(a) Average get latency

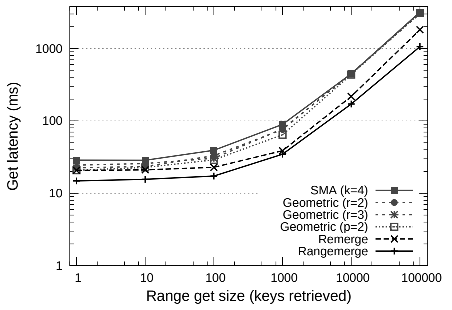


(b) 99th percentile of get latency

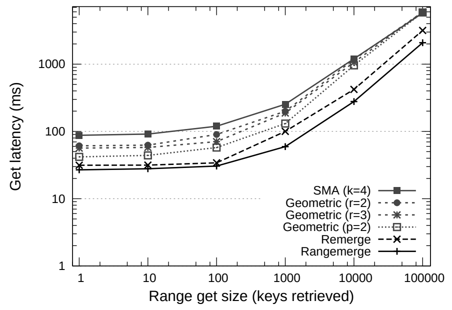


(c) Cumulative insertion time

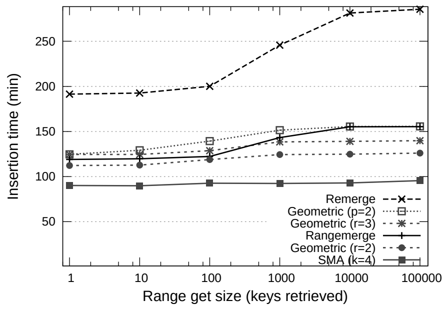
Fig. 10. Performance sensitivity to put load assuming concurrent get requests at rate 20req/s and scan size 10.



(a) Average get latency



(b) 99th percentile of get latency



(c) Cumulative insertion time

Fig. 11. Sensitivity to range get size assuming concurrent load of 2500req/s put rate and 20req/s get rate.

of data using the same compaction pattern during the dataset insertion.

12 Sensitivity Study

We did an extensive sensitivity study with respect to the concurrent load and available memory. First we evaluate the impact of the put load to the query and insertion time. As we vary the put load between 1000-20000req/s, the average latency of concurrent gets is lowest under Rangemerge (Fig. 10a). According to the needs of Service Level Agreements [2], [7], [8], we also consider the 99th percentile of get latency in Fig. 10b. Rangemerge and Remerge are the fastest two methods. Moreover, under concurrent puts and gets, the insertion time of Rangemerge closely tracks that of Geometric (r=3) and lies below that of Remerge and Geometric (p=2) (Fig. 10c). We omit Nomerge because it leads to excessively long get latency.

We also examine the sensitivity to the get size assuming gets of rate 20req/s concurrently served with puts of rate 2500req/s. In Figures 11a and 11b we use logarithmic y axis to depict the latency of get requests. Across different get sizes and especially at the larger ones (e.g., 10MB or 100MB), Rangemerge is distinctly faster (up to twice or more) than the other methods both in terms of average get latency and the respective 99th percentile. From Fig. 11c it also follows that the concurrent get load has an impact to the insertion time. Remerge takes as high as 285min with larger get sizes, unlike Rangemerge that remains between two instances of the Geometric method (r=2 and p=2). In Fig. 12, as the load of concurrent gets varies up to 40req/s, the insertion time of Rangemerge lies at the same level as Geometric and well below Remerge. Under mixed workloads with both puts and gets, from Figures 10, 11 and 12 we conclude that Rangemerge achieves the get latency of Remerge and the insertion time of Geometric.

Also we evaluate how insertion time depends on the memory limit M (Fig. 13). As we increase M from our default value 512MB to 2GB, both Remerge and Rangemerge proportionally reduce the disk I/O time. This behavior is consistent with the respective I/O complexities in Table 2 and Section 9. At M=2GB, Rangemerge lowers insertion time to 15.2min, which approximates the 10.2min required by Nomerge. The remaining methods require more time, e.g., 19.3min for Geometric (r=2), 19.8min for SMA (k=4) and 30.9min for Remerge. From additional experiments (not shown) we found that a higher M does not substantially reduce the get latency of the remaining methods except for the trivial case that the entire dataset fits in memory. We conclude that the insertion time of Rangemerge approximates that of Nomerge at higher ratio of memory over dataset size.

Finally we examine the generated I/O activity of compactions. In Fig. 14 we illustrate the data amount written to and read from disk for 10GB dataset and M=512MB. The plots of the figure are ordered according to the decreasing size of the maximum transferred amount. Remerge (a) merges data from memory to an unbounded disk file with 10GB final size. At the last compaction, the

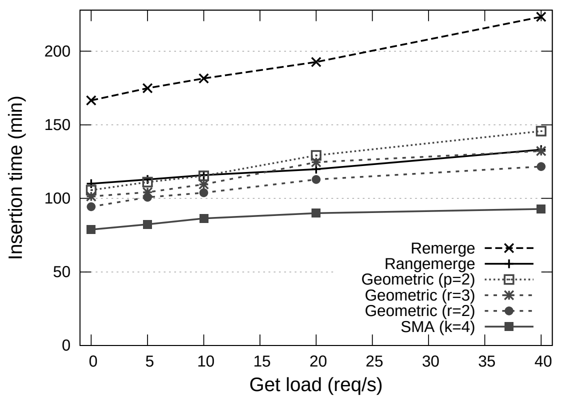


Fig. 12. Sensitivity of insertion time to get rate of scan size 10 with concurrent put rate set at 2500req/s.

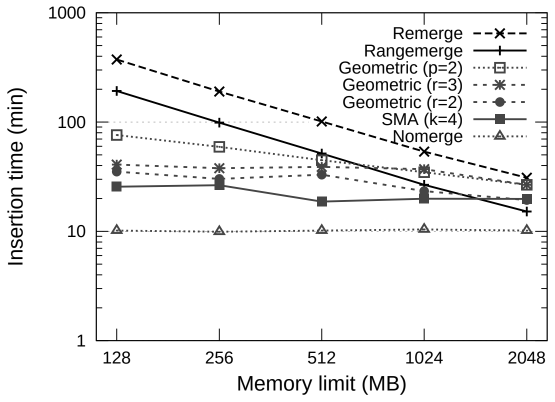


Fig. 13. Sensitivity of insertion time to M. With M=2GB, Rangemerge approaches Nomerge and stays by at least 21% below the other methods.

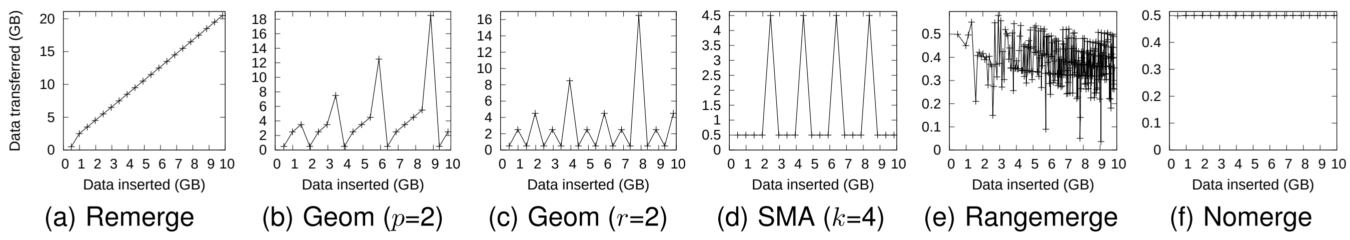


Fig. 14. The total disk traffic of compactions in Rangemerge is comparable to that of Nomerge with M=512MB.

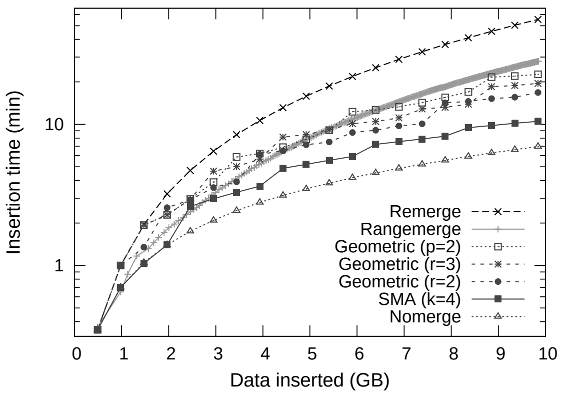
amount of transferred data becomes 20.5GB. Geometric (b-c) reduces the transferred amount down to 16.5GB for r=2. In SMA (d), k=4 limits the transferred amount to 4.5GB; k=2 (not shown) leads to 14.5GB maximum compaction transfer with 6 files. It is interesting that Rangemerge (e) reduces to 594MB the maximum transferred amount per compaction bringing it very close to 512MB periodically transferred by Nomerge (f). Thus Rangemerge makes compactions less I/O aggressive with respect to concurrent gets (Section 6.2).

13 Solid-State Drives

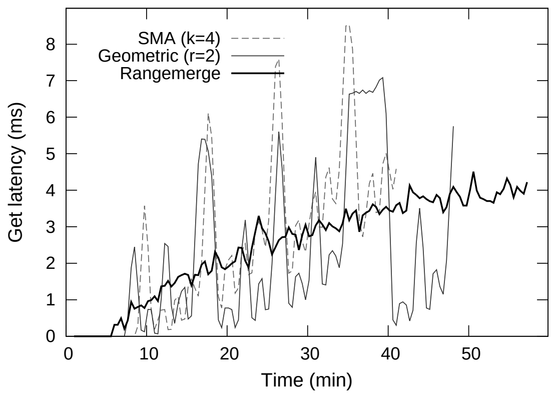
Given the enormous technological improvement of solid-state drives (SSD) over the last decade, it is reasonable to consider their behavior as part of the storage hierarchy in a datastore server. Flash SSDs reduce I/O latency at the cost of hardware equipment and system complexity; the limited lifespan and the relatively poor random-write performance have been recognized as problem for the wider deployment of SSDs [49]. In our following experiments we assume that an SSD fully replaces the hard disk drive (HDD) as medium of persistent storage for the written key-value pairs. Our SSD is a SATA2 60GB solid-state drive of max read throughput 285MB/s and max write throughput 275MB/s. In Fig. 15a we show the cumulative insertion time over an SSD for a 10GB dataset and memory limit 512MB. The compaction methods applied over the SSD reduce by 28%-60% the insertion time measured over the HDD (Fig. 13). However, the relative performance between the methods remains similar across the two devices. In particular, Rangemerge reduces the insertion time of Remerge by 49% with SSD, and by 53% with HDD.

Next we examine the query latency over the SSD device. From the previous paragraph, the write data throughput of our SSD device is about twice as high as that of the HDD. Therefore we increase the put rate of the background traffic to 5000req/s for the SSD from 2500req/s previously used for the HDD (Section 6.2). In order to estimate the query transaction capacity of the two devices, we use a synthetic benchmark with each request involving a random seek followed by a read of 512B block. Thus we found the read performance of the HDD equal to 76req/s, and that of the SSD 4323req/s. First we tried get load of the SSD at rate 1000req/s in analogy to 20req/s that we used for the HDD (26% of 76req/s). However the SSD device is saturated (dramatic drop of throughput) with concurrent workload of 5000req/s puts and 1000req/s gets. Thus we reduced the get load to 100req/s, so that we stay below the performance capacity of the SSD (and keep close to 1/100 the operation get/put ratio as with the HDD).

In Fig. 15b we compare the get latencies of SMA (k=4), Geometric (r=2) and Rangemerge. We terminate the experiment after we insert 10GB into the system concurrently with the get load. In comparison to the get latency over the HDD (Fig. 5a), the measured latencies of the SSD are about an order of magnitude lower. However the curves of the three methods look similar across the two devices. In fact the maximum get latency



(a) Cumulative insertion time



(b) Get latency

Fig. 15. (a) Over an SSD, the insertion time of Rangemerge lies halfway between that of Nomerge and Remerge. (b) Rangemerge reduces the variability of get latency in comparison to SMA (k=4) and Geometric (r=2).

of Rangemerge reaches 4.5ms, while that of Geometric (r=2) gets as high as 7.1ms and that of SMA (k=4) 8.6ms. We conclude that the relative insertion and query performance of the compaction methods remains similar across the two different types of storage devices that we experimented with.

14 Discussion

Motivated from the highly variable query latency in several existing datastores, we propose the Rangemerge method to reduce the I/O intensity of file merging in several ways: (i) We only flush a single range from memory rather the entire buffer space, and keep the amount of I/O during a flush independent of the memory limit. (ii) We combine flushing and compaction into a single operation to avoid extra disk reads during merging. (iii) We keep the size of disk files bounded in order to avoid I/O spikes during file creation. The configurable size of the rangefile provides direct control of the I/O involved in a range flush.

In Table 3 we consider loading 10GB to a datastore at unthrottled insertion rate. From the transferred data and the compaction time we estimate every compaction to require 32.7-36.2MB/s, which is about half of the sequential disk bandwidth. If we reduce the rangefile size from F=256MB to F=32MB, the average duration of a compaction drops from 11.0s to 1.5s, but the respective total insertion time varies in the range 51.6min to 54.2min. It is not surprising that M=32MB raises insertion time to 54.2min, because a smaller rangefile causes more frequent and less efficient data flushes. In practice we can configure the rangefile size according to the insertion and query requirements of the application.

Previous research has already explored ways to continue accepting insertions during memory flushing. When the memory limit M is reached, it is possible to allocate additional memory space of size M to store new inserts, and let previously buffered data be flushed to disk in the background [1]. Alternatively, a low and high watermark can be set for the used fraction of memory space. The system slows down application inserts when the high watermark is exceeded, and it stops merges when the occupied memory drops below the low watermark [9]. Depending on the rate of incoming inserts, such approaches can defer the pause of inserts. However they do not eliminate the interference of compaction I/O with query requests that we focus on in our present study. Essentially the above approaches can be applied orthogonally to the Rangemerge compaction mechanism that allows queries to gracefully coexist with inserts.

Delays and Transferred Data over a Hard Disk

Rangefile (MB)	Flushed (MB)	Transferred (MB)	Compaction Time (s)	Insertion Total (min)
32	4.9	49.1	1.5	54.2
64	9.6	97.8	2.7	51.6
128	19.1	196.0	5.5	51.6
256	37.1	386.8	11.0	53.8

TABLE 3
Amount of flushed and totally transferred data per compaction, delay per compaction, and total insertion time for different rangefile sizes of Rangemerge.

15 Additional Related Work

Cloud data management is comprehensively surveyed by Sakr et al. [50] and Cattell [51].

15.1 Transactions

The General Tablet Server Storage Layer (GTSSL) focuses on the transactional storage efficiency of a single server [12]. The Multi-Tier Sorted Array Merge Tree is a multi-level structure that improves lookup performance with caching, Bloom filters and flash-based storage. Yet the GTSSL designers regard the compaction inefficiency of existing systems as throughput constraint and subject of future work (Section 5.5 [12]). Megastore organizes structured data over a wide-area network as a collection of small databases with strong consistency guarantees and high availability [52]. Percolator extends Bigtable to support cross-row, cross-table transactions through versioning [48]. Spanner manages clock uncertainty to support general transactions over a scalable, globally-distributed, synchronously-replicated database [53]. The PNUTS system supports both point and range queries over a geographically-distributed database that has disk seek capacity as the primary bottleneck [3].

Anvil is a modular toolkit for building storage backends with transactional semantics; the system merges multiple files to ensure logarithmic increase of their number over time, and it runs merges in the background for reduced impact to online requests [16]. Masstree is a shared-memory, concurrent-access structure for data that fully fits in memory [10]. The ecStore builds transactions over a scalable range-partitioned storage system with versioning and optimistic concurrency control [54]. The ES^2 system supports both vertical and horizontal partitioning of relational data [55]. G-Store allows the dynamic creation of key groups for the support of multi-key transactional access [56]. Transactional support is complementary to storage management (e.g. through versioning) and outside the scope of the present work.

15.2 Benchmarking

The performance and scalability of several data serving systems has been studied under the Yahoo! Cloud Serving Benchmark (YCSB) [5]. Cassandra achieves higher performance at write-heavy workloads, PNUTS at read-heavy workloads, and HBase at range queries. The YCSB++ adds extensions to YCSB for advanced features that include bulk data loading, server-side filtering and fine-granularity access control [19]. In a mixed workload over the Accumulo system, read latency varies dramatically in close correlation with concurrent compactions. As a result the compaction policies and mechanisms are emphasized as subject of important future work (Section 3.3 [19]). A synthetic benchmark for graph databases adopts several trace characteristics from a Facebook social graph including the support of range scans [57]. A datastore workload analysis focuses on memory-based caches rather than the backend persistent storage that we examine [39].

15.3 Storage Management

Existing systems alternatively organize data in (i) arrival order, for write-intensive workloads [22], [23], (ii) hash order, for memory efficiency [25], or (iii) key order, for fast handling of range (and point) queries [1]. Possible storage designs include relational engines, dynamic file collections, and file-based hash tables [1], [2], [3].

An index partitions data through either interval mapping to efficiently handle range queries and isolate performance hotspots [1], [3], [6], [14], [15], [24], or key hashing to facilitate load balancing at lower performance of range queries [13], [14]. The indexed sequential access method (ISAM) refers to a disk-based tree used by database systems [58]. ISAM is static after it is created, because inserts and deletes only affect the leaf pages. Overflow inserts are stored at arrival order in chained blocks.

Despite the caching effectiveness of the content-distribution networks used by Facebook, an underlying storage system with high throughput and low latency is required to process the long tail of less popular photos [59]. The online defragmentation supported in a recent filesystem allocates storage space using as few extents as possible, where extent is a contiguous area on disk [60]; such filesystem functionality is complementary to the contiguous placement of a key range at a single file by Rangemerge in the Rangetable structure.

Acknowledgments

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

[48] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *USENIX OSDI Symp.*, Vancouver, Canada, Oct. 2010, pp. 1–15.

[49] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: random write considered harmful in solid state drives," in *USENIX FAST Conf.*, San Jose, CA, Feb. 2012, pp. 139–154.

[50] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Communications Surveys & Tutorials*, 2011.

[51] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, Dec. 2010.

[52] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore : Providing scalable, highly available storage for interactive services," in *CIDR Conf.*, Asilomar, CA, Jan. 2011, pp. 223–234.

[53] J. C. Corbett et al., "Spanner: Google's globally-distributed database," in *USENIX OSDI Symp.*, Oct. 2012, pp. 251–264.

[54] H. T. Vo, C. Chen, and B. C. Ooi, "Towards elastic transactional cloud storage with range query support," in *VLDB Conf*, Singapore, Sep. 2010, pp. 506–514.

[55] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "ES^2: A cloud data storage system for supporting both OLTP and OLAP," in *IEEE ICDE*, Hannover, Germany, Apr. 2011, pp. 291–302.

[56] S. Das, D. Agrawal, and A. El Abbadi, "G-store: a scalable data store for transactional multi key access in the cloud," in *ACM SOCC Symp.*, Indianapolis, Indiana, USA, Jun. 2010, pp. 163–174.

[57] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: a database benchmark based on the Facebook social graph," in *ACM SIGMOD Conf.*, New York, NY, Jun. 2013.

[58] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed. New York, NY: McGraw-Hill, 2003.

[59] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage," in *USENIX OSDI Symp.*, Vancouver, Canada, Oct. 2010, pp. 47–60.

[60] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013.