RangeMerge: Online Performance Tradeoffs in NoSQL Datastores

Giorgos Margaritis Stergios V. Anastasiadis

Department of Computer Science
University of Ioannina
Ioannina 45110, GREECE
{gmargari,stergios}@cs.uoi.gr

Technical Report DCS 2011-13
September 29, 2011

***Abstract*—Datastores are distributed systems that manage enormous amounts of structured data for online serving and batch processing applications. The NoSQL datastores weaken the traditional relational and transactional model in favor of horizontal scalability. They usually support concurrent operations with demanding throughput and latency requirements which may vary across different workload types. A typical tradeoff between query handling and total update costs frequently leads to design decisions that sacrifice query responsiveness for higher update throughput. For this tradeoff, critical component at each storage server is a storage layer that schedules data transfers between memory and disk. After consideration of a similar function in full-text search engines, we systematically examine alternative approaches for online maintenance of structured data over a datastore. Subsequently, we introduce and analyze the Rangemerge algorithm to minimize search time at reasonable total insertion cost. We implement several representative algorithms and experimentally evaluate their update and query cost under the same conditions. We conclude that the Rangemerge algorithm achieves minimal search time and substantially reduces the data insertion time of known methods with comparable search efficiency.**

I. Introduction

Datastores are a new class of distributed systems specifically developed to manage the petabytes of data required by online serving and bulk processing applications, such as web indexing, social networking, electronic commerce and cloud computing in general [1]–[8]. Due to the enormous scale of the handled data and the respective performance requirements, these systems typically span up to several thousands of commodity servers. Depending on the needs of the supported applications, the stored data is structured as collections of key-value pairs, multi-dimensional sorted maps or relational tables of records with attributes. At the top, a centralized or distributed index is used to identify the server and corresponding file of each stored item. For instance, consistent hashing can be used to dynamically partition the data over multiple servers [3]. Alternatively, interval mapping can be applied to horizontally partition the items into groups, called tablets, which are distributed across different machines [2].

At each server, an underlying storage layer is responsible to manage the stored items within the main memory and the disks. This is a critical component for the system performance and durability, because it schedules the data transfers between memory and disk during bulk loading and regular reads or updates. Alternative design options for this component include relational storage engines, dynamic file collections, or file-based hash tables [2]–[4], [9]. In practice, the dynamically maintained collections of immutable files are a typical choice for production systems; they have been most successfully applied to a broad range of batch and online applications for several years [2], [5], [10], [11].

In general, when incoming data arrives at a specific server for permanent storage, initially it is accumulated in the form of appends in memory to amortize the disk access cost during the subsequent write. Over time, memory gets full and data is transferred to an immutable sorted file on disk [2]. A search over a key range is usually applied to all the immutable files on disk to ensure that the entire set of eligible entries is returned. As the number of immutable files increases, it is necessary to merge them so that search time remains constrained. File merging is similar to external sorting. A single file of sorted items is often called run in sorting terminology. Also, a tree representation can be used to specify the sequence of merging steps [12]. The leaves of the tree correspond to the initial runs, while their internal nodes refer to the runs that result from the merging of the descendants. Previous research in database systems has identified the optimization objective to perform as few merge steps and move as few records as possible [13]. Known heuristics always merge the smallest existing runs or mostly use maximal fan-in.

In the past, the information retrieval community has already extensively investigated the problem of online index maintenance to handle queries concurrently with updates. In full-text search, inverted file is an index structure that maps each term to lists of pointers (posting lists) over documents that contain the term. The indexing cost can be amortized if multiple inverted files are created on disk and occasionally are merged according to specific patterns (merge-based method) [14]. Alternatively, the posting list of each term can be separately

maintained on disk (in-place method). Then, the need for contiguity makes it necessary to relocate the lists, if they run out of space at their end. Hybrid indexing methods separate the terms into short and long, based on their respective number of postings [15]. For efficiency, short terms are maintained using merge-based methods, while long-terms use in-place methods.

Despite the prior systematic research on online indexing, current datastores seem to follow ad-hoc approaches without rigorous justification of the structures and parameters used. The Bigtable designers mention to bound the number of files on disk by periodically executing in the background a merging compaction that includes memory and a few files [2]. Regularly, they also apply a major compaction that merges all files into one and suppresses data that was previously marked as deleted. Cassandra merges into one the sorted files which are similar with respect to size [5]. Anvil combines multiple files into one to ensure that their average number grows at most logarithmically over time, but also recognizes the need for more carefully tuned merging parameters and conditions [11]. The read performance of HBase is reportedly sensitive to the number of files [7]. The system originally sorted files by age and only included in compaction an older file, if it had size within twice the size of the newer file. To avoid excessive increase in the number of files, a recent compaction algorithm only includes an older file, if it has size within the aggregate size of all newer files [7]. In Section II, we explain in more detail the architecture of the above systems.

It is not surprising to see a similarity between full-text search engines and datastores, given that web and mailbox indexing is one of the main functions that datastores originally served [2], [5]. Nevertheless, the application scope of datastores spans a broad range of interactive and batch workloads over datasets with diverse statistical properties. Previous research reported that production systems keep the high percentiles of serving latency within tens or hundreds of milliseconds, while the high percentiles of latency may be an order of magnitude longer than the average [3], [16]. Performance improvement is possible if incoming writes are temporarily buffered in memory before reaching the disk, while durability is also achieved if the buffered writes are previously logged to disk at sequential throughput. In fact, high write throughput is additionally facilitated by the underlying append-optimized filesystem typically used (e.g. GFS [17]).

Therefore, in our present work we mostly focus on the read latency of datastores. We first describe the problem of online index maintenance for datastores and identify several performance characteristics that were previously reported to introduce tuning challenges. We trace several problems back to the way that current datastores manage incoming data in memory and disk. Subsequently, we propose an online maintenance algorithm, called *Rangemerge*. The algorithm practically minimizes data fragmentation on disk, while it substantially reduces the data insertion cost in comparison to known methods with comparable search responsiveness. We implement several online maintenance algorithms over the same platform and experimentally confirm our results across different workloads.

Our contributions in the present work can be summarized as follows:

- Problem description of storage management for NoSQL datastores.
- Unified consideration of previous efforts on related problems.
- Introduction and asymptotic analysis of the Rangemerge algorithm to address the problem.
- Implementation of several algorithms and experimental study of the performance tradeoffs they achieve.

In Section 2, we present previous related work, and in Section 3 we set the assumptions and goals of datastore online maintenance but also outline several existing approaches to solve the problem. In Section 4, we introduce and asymptotically analyze the Rangemerge algorithm. In Section 5, we present the characteristics of the experimentation platform that we implemented. In Section 6, we study the experimental results across different workloads. Finally, in Section 7, we summarize our conclusions and future work.

II. Related Work

In this section, we present previous research activity that relates to the online storage management of datastores.

Dynamic dictionary. A dictionary stores a mapping from keys to values. In a two-level memory hierarchy that consists of an internal memory and a disk (*external memory model*), let N be the number of stored items and B the I/O block size [18]. The B-tree implements a dynamic dictionary structure at $O(log_B N)$ asymptotic query and update cost [19]. Bender et al. introduce B-trees for a multi-level memory hierarchy, where cost is analyzed without explicit consideration of the block size B (*cache-oblivious model*) [20]. Gerth et al. introduce the xDict dynamic dictionary, which theoretically achieves optimal tradeoff in space, query and update costs under a broad range of conditions [21]. Byde et al. propose the stratified B-tree, analogously to xDict, as a dynamic dictionary structure which achieves optimal cost tradeoff for fully-versioned data [22]. However, datastores typically rely on a distributed index and use simple structures locally at each storage server.

Storage indexing. In transaction systems with high rate of insertions, the log-structured merge-tree defers and batches index changes from memory to a hierarchy of one or more disk-based trees [23]. The rolling merge method is a variation of merge sort used to delete a contiguous segment of entries from one tree and merge it to the next in the hierarchy. For environments with intense update loads and concurrent analytics queries, the partitioned exponential file divides data into partitions with potentially overlapping key ranges [24]. Each partition is organized into multiple levels with geometrically increasing sizes. The first level lies in memory and the remaining ones on disk, while each level includes a tree-like hierarchical index. A query has to go through the index at all levels of one partition (or more) to retrieve the relevant entries. Due to the limited amount of data stored at each storage server

for scalability, simple local structures are often sufficient in datastores.

Text search. Early work on full-text search parsed documents and gathered term occurrences into a memory-based inverted index, which was occasionally flushed to disk for cost amortization [25]. Later research introduced the Geometric Partitioning method that divides the inverted index on disk into a controlled number of partitions [14]. It offered a range of tradeoffs with respect to query time, indexing rate and disk space. In order to reduce the index building time, hybrid methods categorize terms as short or long depending on their respective low or high frequency of occurrence over the document collection [15]. As a result, merge-based methods or in-place appends are used to manage short or long terms. Previously, we introduced the Selective Range Flush method to dynamically keep low the query time and maintenance cost of an inverted index on disk over fixed-sized blocks [26]. Instead, in the present work we consider datastore structured records whose values have comparable size across different keys. For their maintenance we introduce the Rangemerge algorithm, which always merges the range with most accumulated data from memory to disk.

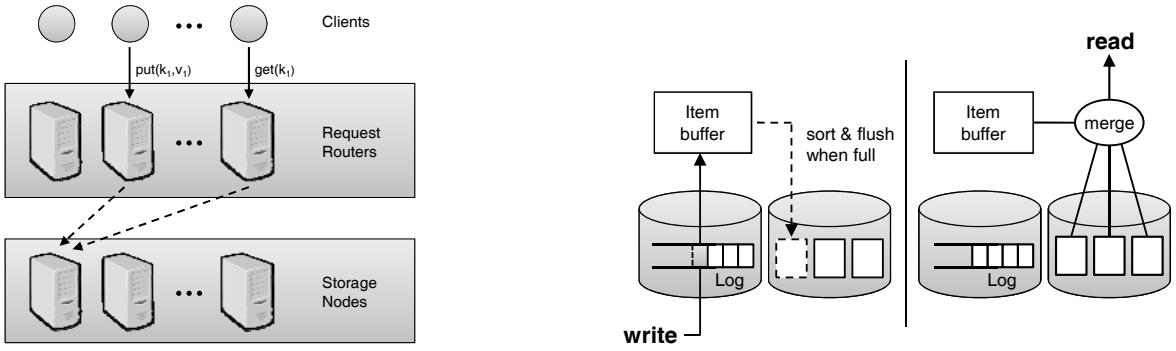
Key-value stores. Cloud data management is comprehensively surveyed by Sakr et al. [27] and Cattell [1]. Bigtable is a structured storage system that partitions stored data across multiple servers, called tablet servers [2]. After being logged on disk, the incoming data is placed in the memory of a tablet server. When a memory threshold is exceeded, a minor compaction transfers the memory data to a disk file, called SSTable. Periodically, a merging compaction transforms multiple SSTables into a single file, while a major compaction produces a single SSTable free of deleted entries. Bloom filters are used to avoid unnecessary key searches over SSTables. A similar design is also adopted by the Cassandra system [5]. Instead, Percolator adds cross-row, cross-table transactions over Bigtable [28]. It uses multi-versioning to support transactions with snapshot-isolation semantics that provide the appearance of reading from a stable snapshot at some timestamp.

The HBASE system is an open-source implementation of Bigtable [6]. The performance of HBASE is sensitive to the number of disk files per key range and the number of writes buffered in memory. Careful tuning of the compaction algorithm combined with special metadata files can be used to improve the read performance [7]. The Amazon Dynamo system stores key-value pairs over a distributed hash table, and accepts a pluggable persistent component for local storage (e.g. the Berkeley Database Transactional Data Store) [3]. The system specifies performance target for high latency percentiles (e.g. 99.9^{th}). Performance increases substantially for reduced durability, if incoming data is initially maintained in memory and only periodically transferred to disk. Service-level objectives based on upper percentile latency are typical in cloud computing and complicate resource provisioning due to the higher variance of upper percentiles in comparison to average latency [16].

Boxwood implements a distributed B-tree as a storage infrastructure with full transaction support [29]. Anvil is a modular, extensible toolkit for database backends, such as abstract key-value stores [11]. The system periodically digests written data into read-only tables, or combines multiple tables into a single one. To amortize the combining cost, a merging pattern similar to Geometric Partitioning is applied. The FAWN system implements a distributed key-value store over flash storage for reduced power consumption [30]. It maintains an in-memory hash table at each server that maps keys to an append-only data log on flash storage. G-Store provides dynamic multi-key transactional access over key-value stores [31]. Haystack is a persistent storage system for photos that implements data volumes as large files over an extent-based file system across clusters of machines [32]. All the above systems face the problem of storage management for structured data, but they don't examine it systematically.

Relational datastores. An evaluation of alternative commercial cloud database systems shows a diversity across the business models of the different providers, which makes choices dependent on the application scalability requirements [33]. The PNUTS system provides a simple relational data model that organizes data into tables of records with attributes [4]. The system uses as a redo log an independent publish/subscribe system, called message broker. For different types of record accesses, such as point or range, the storage unit of PNUTS can use different physical layers, such as a filesystem-based hash table or a MySQL/InnoDB database. The primary bottleneck of the system is the disk seek capacity of the storage units and the message brokers. The bulk record insertion into horizontally-partitioned tables can achieve higher throughput, if a planning phase is used to minimize the sum of partition movement and insertion time [34]. Alternatively, snapshot text files can be used for direct data import into the MySQL tables that lie underneath PNUTS [8].

Megastore adds ACID transactions, indexes and queues to Bigtable [10]. It statically partitions structured data into entity groups, which are independently and synchronously replicated over a wide area. Entities within a single entity group are mutated with ACID transactions, while operations across different entity groups are implemented with efficient asynchronous messaging. The ecStore realizes a scalable range-partitioned storage system over a tree-based structure [35]. The system automatically organizes histogram buckets to accurately estimate access frequencies and replicate the most popular data ranges, while it bases transaction management on multi-versioning and optimistic concurrency control. The ES^2 system supports both vertical and horizontal partitioning of relational data [9]. It also provides efficient bulk-loading, small-range queries for transaction processing and sequential scans for batch analytical processing. Record search is facilitated by a metadata catalog combined with a distributed index. However, the above systems mostly rely on existing relational storage engines for their operation.



(a) Datastore (b) Access Paths

Fig. 1. (a) A datastore receives access requests from the clients. The request routers forward the operation to the appropriate storage node(s). At the storage node, a local index locates the requested item and updates or returns it. (b) A range query reads data from one or multiple files according to the storage algorithm used. A storage server safely logs incoming updates at sequential throughput. As a result, we don't examine the latency of individual updates, and only consider the query latency and the total insertion cost for a dataset.

III. Online Storage Management in Datastores

In the present section, we describe our assumptions, the goals that we set, and existing algorithms that potentially can serve our objectives.

A. Assumptions and Goals

Datastores handle several types of data-intensive tasks, such as analytical processing, bulk loading and online serving. Given the tuning challenges that reportedly arise in online data serving [7], we focus on the performance tradeoffs that arise in datastores under this type of workload. Even though we consider the cost of data insertion and short scan requests, we leave the issues of analytical processing and bulk loading outside the scope of the present work [8], [36]. In particular, we examine NoSQL datastores, which are differentiated from traditional relational database systems in several aspects [1]. They horizontally partition and replicate data over many servers, provide weak concurrency model and simple call interface, use memory and distributed indexes for storage and allow dynamic addition of attributes to records. The interface supports put request with a single record as parameter and get requests with a range of columns as parameter. For conciseness, in the rest of the present document, we refer to NoSQL datastores simply as datastores.

It is typical for a distributed datastore to employ a multi-layer indexing structure (Fig. 1(a)). The upper layer (sometimes called router or master [2], [4]) is a centralized or distributed index that locates the server where a range of items is stored. The lower index layer is responsible to manage the local data at each server and accordingly handle the incoming queries and updates. Datastores support range queries, as scans along a key range, or single key requests over a version interval. We regard point accesses as a special type of range queries, and only consider range queries along the key dimension of unversioned data. We leave for future work the study of query handling over versioned data with close consideration of specialized dictionary structures recently designed for that purpose [22].

At each server of the datastore, fast lookup operations are often supported by a memory-based sorted array of lookahead pointers to a configurable subset of the stored records on disk [2]. Unnecessary disk accesses for point queries can be significantly reduced through memory-based Bloom filters [37]. However, this reduction is not possible for range queries (Fig. 1(b)). Complex tree-like structures incur asymptotically logarithmic query cost for the case that the index does not entirely fit in memory, as is the case with traditional relational workloads [20], [21]. However, tree-like structures are not required by datastores, which are optimized for low latency and achieve capacity scalability through horizontal partitioning of the data and the corresponding requests across multiple servers.

Availability is increased through eager or lazy data replication across multiple servers, undertaken by the datastore itself or an underlying distributed filesystem [2]–[4], [17]. In general, the corresponding consistency of data across multiple servers can be maintained with a quorum algorithm that specifies the minimum number of servers that participate in a successful operation [3]. Both distributed consistency and availability are high-level issues that we don't consider, since we mainly study the storage functionality of a single server. Durability requirements vary depending on the performance characteristics of the applications. One possibility is to do write-ahead logging of all incoming updates before they reach the memory of the server [2]. Hence, write operations complete at sequential disk throughput if a separate disk is dedicated to logging, while the write path to the final disk location is only occasionally invoked by the storage server. Alternatively, writes reach memory initially, and become durable at a later stage by periodic data transfer to disk, or memory replication among multiple servers [3], [4].

The storage management at each server is often implemented through a dynamic file collection or a relational storage engine [2], [3]. Alternatively, file-based hash tables and tree-like indexes are also mentioned as possible solutions for more specialized applications [9]. In a general evaluation study

conducted by Yahoo!, systems that relied on dynamic file collections (e.g. Cassandra, HBASE) were at least as efficient or better in comparison to a system based on relational storage engine (e.g. PNUTS) [6]. Since we also target general-purpose workloads, we only consider dynamic file collections at the storage layer of each server. They optimize durable data insertion relatively easily at the expense of read processing over multiple files and subsequent aggregation of the results. On the contrary, the read performance of datastores is hard to optimize for several reasons:

- Service-level objectives are usually specified in terms of upper-percentile latency, which lies an order of magnitude higher than average latency [3], [16].
- Read performance is correlated with the number of files at each server, which depends on the respective file compaction algorithm [7].
- The amortization of disk writes may lead to intense device usage that causes intermittent delay (or disruption) of the normal online operation [38].
- The diversity of supported applications requires acceptable operation across different distributions of the input data keys, including several variations of the Zipfian distribution [6].

Previous referenced research has already pointed out these issues and examined tuning options that help improve the performance under particular workloads. However, to the best of our knowledge, the present work is the first to comprehensively investigate the problem of indexed storage management in datastores.

In our study, we deal with the online maintenance of persistent data placed at each storage server of a NoSQL datastore. We aim to achieve search cost of no more than one disk I/O operation per point query only increased by the necessary additional transfer time for range queries. Assuming that updates become safe through write-ahead logging, we efficiently transfer accumulated written data from memory to disk without considerable interference with concurrent read operations. We are interested to keep read latency (including the upper percentiles) within tens or hundreds of milliseconds, as required by online applications.

B. Known Methods

We seek to efficiently transfer accumulated items from memory to disk in a way that guarantees fast range reads and writes. Ideally, we shall keep all items sorted and contiguously stored on disk in a single file to minimize query time. A similar objective previously showed up in the online maintenance of inverted files for text retrieval. However, an item in text retrieval includes a posting list whose size follows the Zipfian distribution. As a result, the latest hybrid methods in text retrieval manage differently the keys according to their list size [15].

In a datastore, all records can be assumed to have similar –but not necessarily equal– storage space requirements and be treated the same. The access frequency of keys may follow a uniform distribution or different variations of the Zipfian [6].

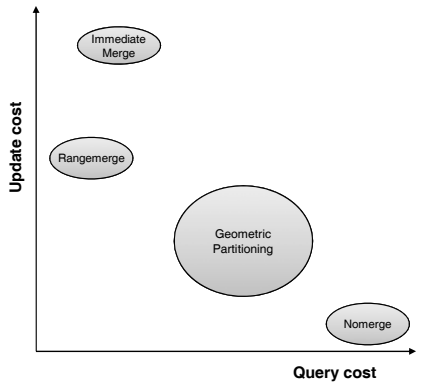


Fig. 2. We illustrate the query/update tradeoff across the different algorithms that we consider. The update cost refers to the total cost of data insertion rather than the latency of individual updates. We have Rangemerge appear to the left of Immediate Merge implying lower query cost, because we directly control the I/O intensity of concurrent file merges (insertions) through the rangefile parameter F (Section VI).

In comparison to search, data freshness in datastores is more demanding due to the semantics of the applications involved. For instance, a stale shopping cart is easily noticeable in electronic commerce, while a recently received message can remain non-searchable for a few minutes in a mailbox without problem. Practically, we shall not backlog read and write requests just because a server is currently busy with file merging. Quite surprisingly, existing datastores only loosely control this maintenance overhead through heuristics about the number and relative size of merged files [7].

Despite the above differences, approaches used by text indexing are already widely adopted in datastores. In particular, merge-based algorithms that transform multiple inverted files into one can be directly applied to the immutable files maintained by a datastore server [11]. As baseline case of file merging, we may refer to the *Nomerge* method which does not merge the files at all [39]. Every time the memory of size M gets full, data is transferred to disk creating a new sorted file of size M. Let's assume that the disk access cost is linear function of the number of data items involved in the operation. For a collection of N items, the method incurs isertion cost $\Theta(N)$ to create the N/M sorted files on disk. Correspondingly, every point query touches all files on disk with asymptotic cost $O(N/M)$, assuming that only a small chunk of each file is retrieved in memory for the search.

Immediate Merge is a straightforward merge strategy, which keeps at most one record file on disk [15]. When memory gets full, memory data is merged with the existing file resulting into a new file. This algorithm guarantees that sorted data is stored contiguously on disk leading to at most one disk access per point query, making the respective cost $O(1)$. For every merge, the algorithm reads the entire stored file and incurs asymptotically quadratic cost ($\Theta(N^2/)M$) in total. The reads and writes of index building are simple sequential scans that occur at high disk throughput. Although this observation is not captured by the asymptotic cost, it substantially affects the actual cost of index building.

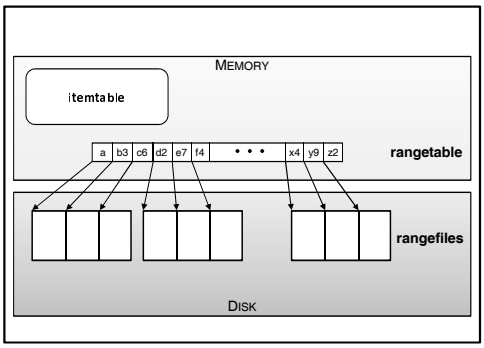


Fig. 3. The Rangemerge algorithm buffers incoming data items in the itemtable and uses rangefiles to organize the data on disk. The rangetable is a sparse index that points to chunks across each rangefile on disk.

Geometric Partitioning strives to minimize the total merging cost by keeping multiple record files on disk, but without allowing their number to grow excessively [14]. For a sequence of files on disk, the method introduces the parameter r to specify a lower ($r^{k-1}M$) and upper bound ($(r-1)r^{k-1}M$) in the capacity of the kth file, where $k = 1, 2, \ldots$ Consequently, we get a series of hierarchical merges that guarantee similar sizes between the files that participate in each merge. The disk access cost involved in index building is asymptotically equal to $\Theta(N \log(N/M))$. A point access incurs cost $O(\log(N/M))$ over a collection of N indexed items. Several datastores (e.g. Cassandra, Anvil, Lucene) adopt a similar method to merge the immutable files maintained on each server [5], [11].

In Geometric Partitioning, one parameter value that was previously suggested is $r = 3$ [14]. *Logarithmic merge* is an alternative hierarchical merging method similar to Geometric Partitioning with $r = 2$ [15]. For a workload dominated by queries, a variation of Geometric Partitioning allows to directly constrain the largest number p of files maintained on disk with dynamic adjustment of parameter r. Then, Immediate Merge can be considered as a special case of Geometric Partitioning with $p = 1$. In general, the indexing cost of Geometric Partitioning with p partitions is asymptotically $\Theta(N(\frac{N}{M})^{\frac{1}{p}})$. The point query cost is $O(p)$, because every key search touches p files on disk. We illustrate the comparative update/query behavior of the algorithms in Fig. 2

IV. The RangeMerge Algorithm

In current datastores, the local indexing methods at the storage server aim to combine high update throughput with fast range queries. They amortize writes through batch sequential transfers from memory to disk and improve read performance of point queries with data-intensive merge sorts and Bloom filters. We introduce the RangeMerge simple indexing method that supports point queries with one block access, and batch updates or range queries with approximately sequential disk accesses. Bloom filters can be used complementarily to RangeMerge for additional efficiency in point queries.

A. Description

Ultimately, we strive to minimize the response time of reads and writes at the lowest total cost of memory space and disk throughput involved at each storage server. We make the practical assumption that datastores partition their data across multiple servers in order to scale their throughput and storage capacity at low latency. In production datastores, each storage server usually ends up with a few terabytes of data on disk, which is easily indexed through a memory-based sparse structure [2]. The necessary structure is a sorted array with pairs of keys and lookahead pointers to disk locations every few tens or hundreds kilobytes of stored data.

For instance, Cassandra indexes chunks of 256KB, while Bigtable indexes blocks of 64KB [2], [5]. If the average rotation and seek time takes about 7ms in a 10KRPM SAS drive, then the data transfer time for 64-256KB takes an additional 10-25% [40]. With an index entry with length 100 bytes for every 256KB, we need 4GB of memory to sparsely index 10TB of data on disk. The estimation drops to 1GB of memory, if we use key hashes of size about 20 bytes each (e.g. with SHA1). Typically, the sparse index is considered soft state that is periodically checkpointed to disk and dynamically reconstructed in case of unexpected server failure. Thus, the remaining challenge in the storage indexing of datastores is to develop a maintenance strategy that: (i) Efficiently flushes data from memory to disk, (ii) Keeps limited the disk space fragmentation, and (iii) Stores at single disk location the data of each key range.

We believe that it is unnecessary to handle separately the flush of all newly accumulated data from memory to disk and the merge of existing data files into larger ones. If we reach a high threshold in the allocated memory, it is sufficient to selectively flush specific data items. Our objective is to free as much memory space as possible with minimal cost to read existing data items from disk and flush them back after their merge with selected new data. Furthermore, it is unnecessary to maintain enormous files on disk for the sake of sequential I/O during disk reads and writes. In current enterprise SAS (or SATA) disks, the disk overhead accounts for about 1% of the total I/O cost if the amount of transferred data is in the order of 10MB. Essentially, we may efficiently handle range queries if we allocate our data in disk blocks of size 10MB or more. In order to additionally serve point queries with one disk operation, we need to always merge new data from memory into existing data items on disk at the corresponding key ranges.

We call Rangemerge the algorithm that we introduce for the storage management of datastores. The pseudocode appears as Algorithm 1. In order to achieve our goals, we allocate the disk space in *rangefiles* of fixed size F (e.g. $F = 256MB$). This makes unnecessary to reorganize the disk space over time for the allocation of variable-sized files [24]. The rangefile size is multiple of the basic I/O block size B configured in the operating system (e.g. $B = 4KB$). We partition our data into key-sorted ranges, each of which fits into a respective

Algorithm 1 RANGEMERGE in pseudocode
Input: Indexed data in memory and disk
Output: Updated indexed data in memory and disk

```
1: Sort ranges by total memory space
2: while (allocated memory space ≥ M) do
3:     {Get range with max size in memory}
4:     R := range of max memory space
5:     {Merge R to disk}
6:     Read the rangefile of R from disk
7:     Merge the rangefile of R with the new data
8:     {Handle rangefile overflow}
9:     if (rangefile overflows) then
10:        Allocate new rangefiles
11:        Split and store data equally across old/new rangefiles
12:    else
13:        Store merged data on old rangefile
14:    end if
15:    Release the memory space reserved for range R
16: end while
```

rangefile. Each rangefile consists of fixed-size chunks of size C (e.g. $C = 256KB$) that we locate with a sparse memory-based index, called *rangetable*. We also maintain in memory a separate mapping structure that we call *itemtable*. We use the itemtable to hold the items currently held in memory, and support fast key lookup and key-sorted scan. We allow the stored items to have variable size.

Initially, the server has a single range $(-\infty, +\infty)$ without any stored data. We insert new data items in memory until the occupied space reaches a preconfigured *memory threshold* M. At this point, we determine the *victim* range that we flush to disk. The victim range is merged with the rangefile that stores the respective key range on disk. The merging requires to read from disk the entire occupied part of the rangefile, merge it with the victim range in memory, and move it back to disk (lines 6-14). Then, we free the memory space reserved for the victim range (line 15). During merge, rangefiles are read from and written back to disk sequentially. When we flush a victim range to disk, it is possible that the needed disk space exceeds the rangefile size (line 9). In that case, we equally split the range into two or more subranges as required. Subsequently, we transfer the data from memory to a corresponding number of rangefiles that we dynamically reserve on disk for that purpose (lines 10-11).

The choice of the victim range affects the system efficiency in several ways. First, every time we flush a range, we incur the approximate cost of one rangefile read and write for the respective merge. If a range is frequently updated, then it may be preferable to keep it in memory and avoid to pay the merge cost multiple times. Second, if a range is used often by point or scan queries, then it is beneficial to keep the range cached and avoid to read it back shortly later. Third, a flushed range releases memory space that is vital for the system to continue accepting new append-like updates. The larger amount of memory space we release, the longer it will take before we pay the merging cost again. In the meantime, the disk throughput can serve synchronous read requests, which are directly visible as query response time to the user.

Essentially, the range flush incurs the cost of rangefile update and the benefit of memory release. We adopt a relatively simple rule to victimize the range with the largest amount of data currently in memory (line 4). The intuition behind this choice is to maximize the freed memory space along with the efficiency of the data transfer to disk. Our approach only greedily considers the current size of each range in memory. However, it does not account for the current size of each rangetable on disk; this affects the merging cost and the distribution of future read/write requests across the different key ranges, which determines the caching behavior of the algorithm [41]. Despite its simplicity, the victimization rule of Rangemerge proved robust across a multitude of experiments that we did. More complex analytical rules could potentially capture more accurately the above cost-benefit tradeoff, but we leave for future work a more thorough consideration of this optimization.

B. Asymptotic Analysis

We aim to estimate the total amount of bytes transferred to/from disk during the insertion of N data items. For simplicity, we assume that each item occupies one byte. Since a rangefile starts with size $0.5F$ right after a split and cannot exceed size F, we estimate the average rangefile size equal to $\bar{F} = 0.75F$. Without loss of generality, we also assume that every time the occupied memory reaches the capacity M, we flush M_f bytes to disk. This is a bookkeeping simplification, which simply implies that we flush the necessary number of ranges required to free space amount M_f in main memory.

Let k_f be the number of flushes of M_f bytes needed to insert N items and k_m be the corresponding number of rangefile merges involved. Then, the total number of bytes read from and written from disk is:

$$T = k_f \cdot M_f + k_m \cdot 2 \cdot \bar{F}, \tag{1}$$

where the factor 2 accounts for the read and write of each rangefile during a merge.

Since at every memory flush we free M_f bytes, the insertion of N items incurs the following number of flushes:

$$k_f = \frac{N}{M_f} \tag{2}$$

Let R_i be the total number of ranges in the server before the ith flush. If the items are uniformly distributed across the ranges, then all ranges occupy the same space B_i in memory during the ith flush. As a result $B_i = M/R_i$, and the total number of rangefile merges can be estimated as follows:

$$\begin{aligned} k_m &= \sum_{i=1}^{\#flushes} (number\ of\ merges\ at\ ith\ flush) \\ &= \sum_{i=1}^{k_f} \frac{M_f}{B_i} = \sum_{i=1}^{k_f} \frac{M_f}{M/R_i} = \sum_{i=1}^{k_f} \frac{M_f \cdot R_i}{M} \end{aligned} \tag{3}$$

In order to estimate the number of ranges R_i in the server, we first note that the $(i-1)$ prior flushes transferred to disk

a total of $(i-1) \cdot M_f$ bytes. Given the average rangefile size $\bar{F}$, we have:

$$R_i = \frac{(i-1) \cdot M_f}{\bar{F}} \tag{4}$$

From equations 3 and 4, we get:

$$\begin{aligned} k_m &= \sum_{i=1}^{k_f} \frac{M_f \cdot \frac{(i-1) \cdot M_f}{\bar{F}}}{M} \\ &= \sum_{i=1}^{k_f} \frac{M_f^2 \cdot (i-1)}{\bar{F} \cdot M} \\ &= \frac{M_f^2}{\bar{F} \cdot M} \sum_{i=1}^{N/M_f} (i-1) \\ &\leq \frac{M_f^2}{\bar{F} \cdot M} \sum_{i=1}^{N/M_f} \frac{N}{M_f} \\ &= \frac{M_f^2}{\bar{F} \cdot M} \cdot \frac{N^2}{M_f^2} = \frac{N^2}{\bar{F} \cdot M} \end{aligned} \tag{5}$$

It is interesting that in Eq. 5 the factor M_f cancels out. Using equations 1, 2 and 5, we conclude:

$$T = \Theta\left(\frac{N^2}{M}\right) \tag{6}$$

The estimation of Eq. 6 is an asymptotic upper bound that we developed for comparison purposes with previous work, based on the amount of data transferred between memory and disk. This cost does take into consideration the low disk access overhead of Rangemerge due to sequential data transfers. Additionally, it does not account for the fact that data insertion with Rangemerge only incurs relatively small disk transfers of low interaction with regular read operations. We do explore experimentally these issues in Section VI.

C. Summary

Overall, the Rangemerge algorithm offers the following important benefits in the storage management of datastores:

1) Keep the data of each key range at a single disk location.
2) Support sequential disk scans over ranges of sorted data.
3) Batch incoming updates in memory for efficiency.
4) Selectively free memory space for new updates and natively cache data for queries.
5) Naturally apply disk updates in range granularity.
6) Reduce disk overhead in merges through sequential accesses.
7) Prevent external fragmentation of disk space and the need for periodic reorganization.

Equation 6 makes the indexing cost of Rangemerge asymptotically equivalent to that of Immediate merge. Also, both these algorithms require at most one disk operation per point query. We show the comparative behavior of Rangemerge with respect to other algorithms in Fig. 2. In the remaining sections, we experimentally evaluate Rangemerge in comparison to representative existing algorithms that we implemented in the same environment.

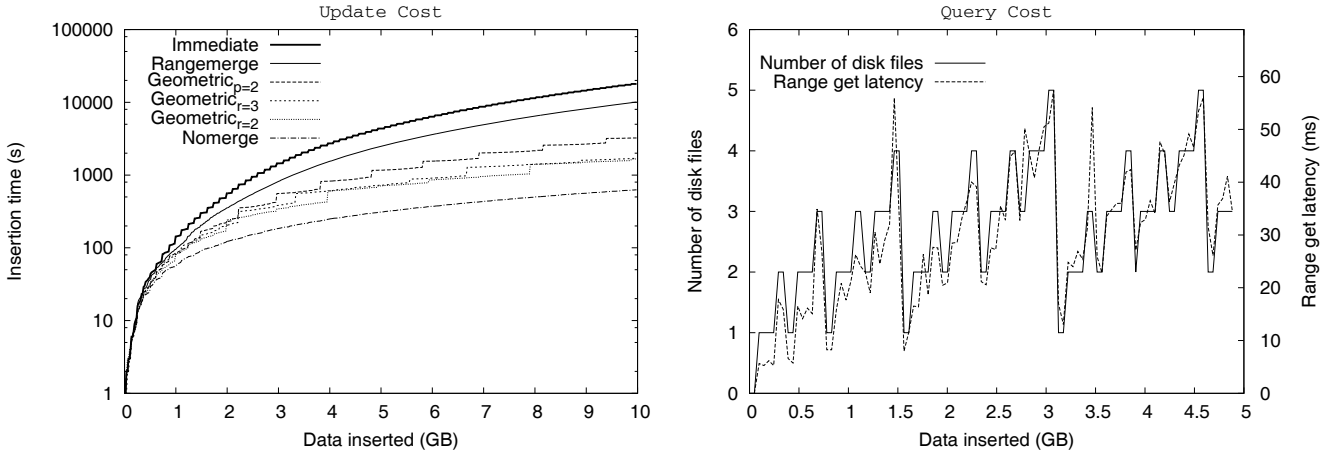
V. Experimentation Environment

We developed a simple storage manager that implements file compaction on disk with the following algorithms: Nomerge, Immediate Merge, Geometric Partitioning and Rangemerge. We use a red-black tree in memory to maintain the incoming data in sorted order. Nomerge dumps the memory data to a new sorted file every time memory gets full. Immediate Merge merges memory data into the single sorted file that it holds on disk. Geometric Partitioning keeps a controlled number of sorted files for alternative parameter values $r = 2$, $r = 3$ and $p = 2$. The value $r = 3$ was used in the original work on Geometric Partitioning, the value $r = 2$ is similar to the Logarithmic method and $p = 2$ is another interesting case extensively studied previously [14], [15], [39]. Rangemerge picks the range with the largest amount of data in memory and merges it into the respective rangefile on disk. For all the algorithms, we maintain the same sparse index in memory to efficiently find the file chunks that store the items of a particular key range. We implemented the algorithms using C++ and the standard template library (STL). The new code that we developed consists of 3588 uncommented lines. It is important to point out that our implementation of Rangemerge is unoptimized especially with respect to the memory management operations that it heavily involves.

We did our experiments on a server running the Debian distribution of Linux version 2.6.18. The server is equipped with one quad-core x86 2.33GHz processor, 3GB RAM, gigabit ethernet and two 7200RPM SATA disks of 500GB each. The disk specification mentions 16MB buffer size, 8.9ms average seek time, and 72MB/s sustained transfer rate. We store all the data files on one (non-root) disk over the default (ext3) file system of Linux. In the Rangemerge algorithm, we used rangefiles of size F=256MB and chunks of size C=64KB. In all our experiments we load the system with key-value pairs of 100 bytes key and 1KB value. We experimented over synthetic datasets that follow the uniform and Zipfian key distribution [6], [42]. In our figures, we only show results for the uniform distribution. The numbers were similar for a Zipfian keys distribution across the different ranges (unless we trivially allowed insertions with overwrites of existing items to reduce disk traffic). In the range queries we use requests for 10 consecutive items. We run all the experiments on the same machine to avoid any measurement variations due to minor system configuration differences across our cluster.

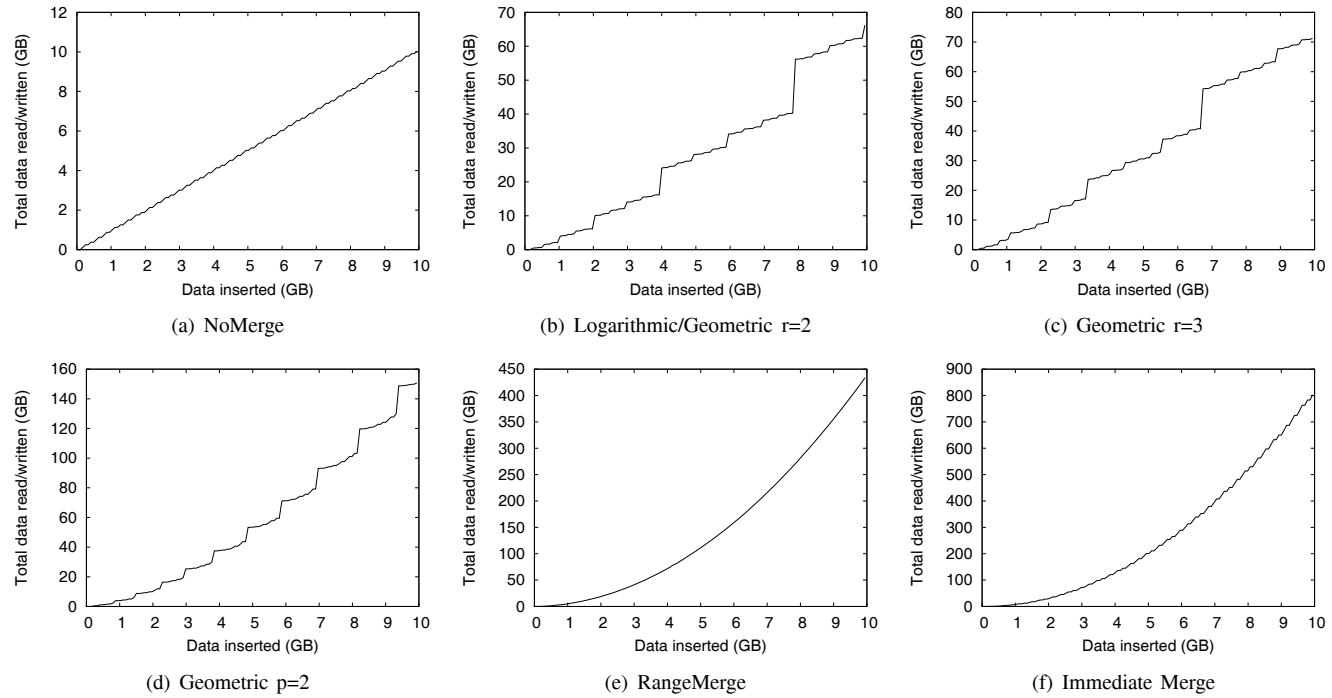
VI. Performance Evaluation

In the present section, we experimentally evaluate the data insertion time of different algorithms, the relationship between read latency and number of files for Geometric Partitioning, the disk transfer activity and the number of files maintained by the different algorithms, the indexing time for different amounts of memory. This quantification is important, because it captures important issues not accounted for by an asymptotic analysis. In particular, we show that Immediate Merge maintains a single file on disk, but increases substantially the insertion cost. Geometric Partitioning reduces substantially the



(a) (b)

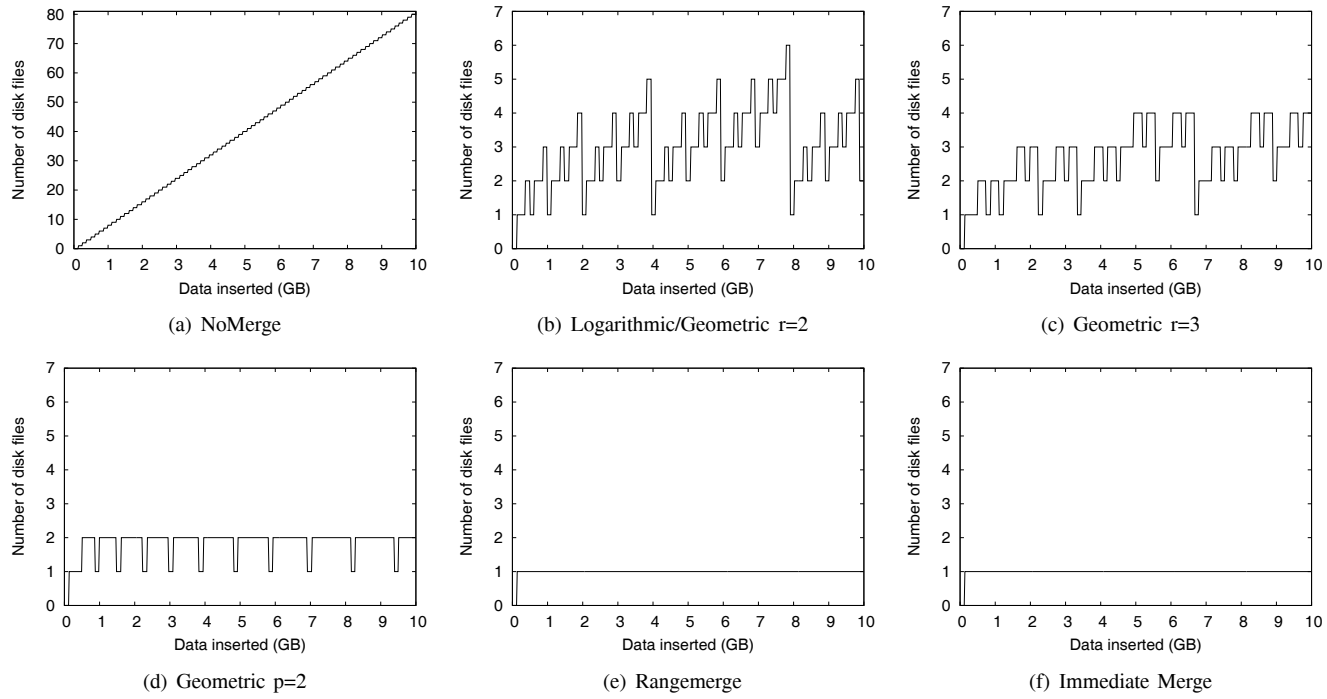
Fig. 4. (a) We measure the indexing time for 10GB of data across six algorithms including three variations of Geometric Partitioning. We notice that Immediate Merge takes an order of magnitude longer than Geometric Partitioning, while Rangemerge lies in-between. (b) On an otherwise idle system, we measure the latency of a single range read for Geometric Partitioning with r=2 across different dataset sizes up to 5GB. We notice a visible correlation between read latency and the number of files that hold the data on disk.



(a) NoMerge (b) Logarithmic/Geometric r=2 (c) Geometric r=3

(d) Geometric p=2 (e) RangeMerge (f) Immediate Merge

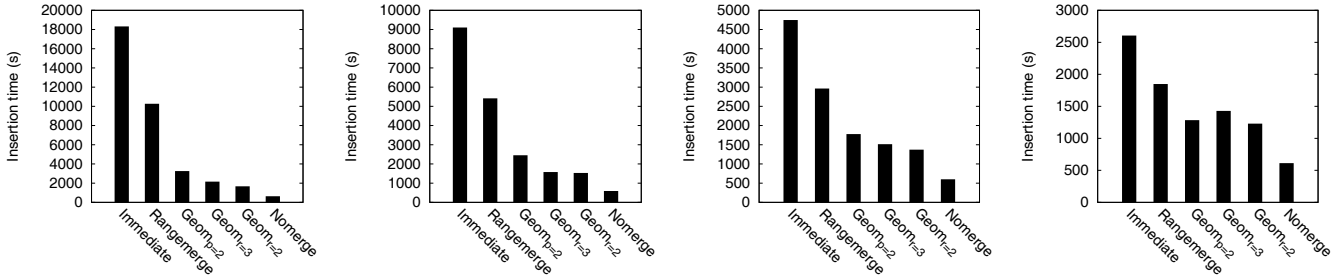
Fig. 5. We measure the cumulative volume of data transferred to/from disk during the insertion of a dataset with increasing size. Nomerge only writes the 10GB that it receives as input, while the remaining algorithms incur additional transfer activity for the merges. Geometric Partitioning demonstrates a stair-wise increase in disk access intensity over time due to the amortizations of the applied merging pattern.



(a) NoMerge (b) Logarithmic/Geometric r=2 (c) Geometric r=3

(d) Geometric p=2 (e) Rangemerge (f) Immediate Merge

Fig. 6. During the insertion of a dataset, we evaluate the number of disk files across which we store the data of a key range. As expected, Immediate Merge and Rangemerge store each range at a single file, while Geometric Partitioning varies it in a controlled way. Nomerge stores data over an unbounded number of files.



(a) Memory size = 128MB (b) Memory size = 256MB (c) Memory size = 512MB (d) Memory size = 1GB

Fig. 7. We examine the sensitivity of insertion time to memory space M. Rangemerge and Immediate merge reduce insertion time in proportion to the extra memory space. In the case of $M = 1GB$, Rangemerge takes 30.8min, while Geometric Partitioning with $p = 2$ takes 21.5min, i.e., 44% less.

insertion cost, but maintains multiple files on disk increasing the read latency. Also, the respective access activity shows a stair-wise intensity behavior, which reportedly interferes with the handling of regular data reads [38], [43]. Rangemerge strikes a desirable balance among the above issues, because it maintains every key range at a single disk location, merges data from memory to disk in configurable granularity of rangefiles and incurs total insertion cost that lies between Immediate Merge and Geometric Partitioning.

A. Insertion Time and Disk Transfer Volume

Insertion time refers to the total delay to insert a dataset to the storage manager. It is predominantly spent on I/O, although in addition to disk I/O operations it includes the processing time to sort and maintain the data in memory. We first examine the insertion time for a dataset across the Nomerge, Rangemerge, Immediate Merge, and three variations of Geometric Partitioning for p=2, r=2 and r=3. Geometric Partitioning for p=2 guarantees that the server uses up to two files to store the data on disk. In order to have each experiment complete within a few hours, we scale down the amount of inserted data to 10GB and correspondingly limit to 128MB the amount of memory used for indexing [6]. In Section VI-C, we examine in detail the sensitivity of indexing time to memory space.

In Fig. 4(a) we illustrate the cumulative insertion time for an increasing dataset size. Nomerge only takes 10min because it simply dumps the data from memory to disk as 81 sorted files of 128MB each. At the other extreme we have the Immediate Merge that only maintains one sorted file on disk

but takes more than 5 hours. Rangemerge keeps each range at a single disk location with total indexing time about 2.85 hours. Geometric Partitioning for $p = 2$ requires about 54 min and for $r = 2$ it takes 28 min, but it has to maintain multiple files on disk. Furthermore, in Fig. 5 we measure the cumulative volume of data transferred between memory and disk during the indexing of the dataset. It is interesting that Geometric Partitioning leads to a stair-wise curve, which also appeared in Fig. 4(a) for the insertion time. The observed pattern is reasonable because Geometric Partitioning merges the files to keep their number small but leads to unbounded file size. This disk access activity is documented as undesirable, because file merging heavily loads the storage devices and negatively affects the system responsiveness [38], [43]. Similarly, algorithm designers strive to deamortize the complexity of data structures [20].

Although less visible in the figure, Immediate Merge also regularly merges memory data to an unbounded disk file which eventually becomes 10GB. On the contrary, Nomerge transfers data in relatively small units of 128MB, while Rangemerge uses rangefiles whose configurable size keeps them up to 256MB in our settings. In fact, we can set the rangefile parameter F to smaller values without considerably affecting the overall system efficiency. Overall, in Fig. 5 the amount of disk data read and written varies significantly across the different algorithms. Nomerge only transfers to disk the amount of 10GB that it receives as input, Immediate Merge incurs a total disk access volume of 803GB and the remaining algorithms lie in-between. Across all the cases, the disk accesses are efficient due to the prevalent sequential transfers.

B. Read Latency and Number of Disk Files

In our experiments, we only consider the latency of get requests (Fig. 1(a)). However, we don't examine put requests, because updates return immediately after the data reaches a write-ahead log at sequential disk throughput (Fig. 1(b)). Thus, in Fig. 4(b) we consider the Geometric Partitioning for $r = 2$. We measure the latency of a range get as we increase the amount of indexed data in steps of 50MB. In the same plot, we also show the number of sorted files maintained by the system on disk. Even though a Bloom filter can potentially suppress point queries to a file, this is not the case with range queries. Thus, we assume that a range get requires to read from every disk file the chunk that contains the requested key range. This translates to multiple random I/O operations, whose number depends on the number of disk files.

In particular, Fig. 4(b) shows that the get latency varies roughly between 10-60ms respectively in close correlation with the number of files that changes between 1 and 6. The reported numbers are best case in the sense that we only have one read operation running in the system. The reported latency is average over ten repetitions. In production environments, the server handles multiple concurrent read operations which usually increase the load and corresponding latency. Also, we don't consider the additional delays introduced for consistency when the results by multiple servers are combined –for instance, through a quorum algorithm [3]– to the response returned to the client. Nonetheless, it is clear that multiple disk files affect by several factors the read latency of the system.

We further investigate this issue in Fig. 6 for all the six algorithms. In Fig. 6(b-d) Geometric Partitioning allows the number of files to reach respectively 6, 4 or 2, according to the indicated values of r and p. Immediate Merge stores all data on a single file, which allows a range read to locate the requested data at a single disk location. We relax this restriction of Immediate Merge in Rangemerge to store every key range at a single disk location but partition the data into rangefiles of configurable capacity. Thus, given the requirement of online applications for interactive operation, we manage to minimize the latency of reads by avoiding multiple seeks across different files at the data storage level.

C. Sensitivity to Memory Size

In another set of experiments, we examine the sensitivity of indexing time to the available amount of memory space. We aim to evaluate how the different algorithms use extra memory space to decrease the insertion delay. Along with the 128MB that we used in our previous experiments, we also consider $M = 256MB$, $M = 512MB$ and $M = 1GB$. As we see in Fig. 7(a-d), an increasing amount of memory tends to reduce proportionally the indexing time of Immediate Merge and Rangemerge. Essentially, as we devote more space for batching of incoming data in memory, we accordingly reduce the number of merging steps and the respective amount of data that we transfer between memory and disk. It is interesting that Geometric Partitioning does not obtain the same proportional benefit from extra memory. In fact, the reduction of indexing time diminishes across the algorithm variations that we examine and especially for $p = 3$. Also, for $M = 1GB$ the insertion time of Rangemerge becomes 29.4% more than Geometric Partitioning with $r = 3$. We conclude that as server configurations become more powerful, we anticipate the reduced read latency of Rangemerge to be combined with comparatively lower total insertion time.

VII. Conclusions and Future Work

We describe the problem of indexed storage management in datastores and consider existing algorithms from literature. We specify the requirements for minimal query time and reasonable insertion cost with consideration of the I/O intensity during data merges between memory and disk. Then, we propose and analyze the Rangemerge algorithm. In experimental comparison with existing algorithms, we show that Rangemerge practically achieves minimal search time with configurable intensity of disk updates. In our future work, we plan to incorporate Rangemerge into a multi-tier datastore and evaluate the latency of the access path along the distributed index. Other extensions that we plan to do include the handling of multi-versioned data and the optimization of the range victimatization rule that we use in Rangemerge.

REFERENCES

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

[2] 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 Symposium on Operating Systems Design and Implementation*, Seattle, WA, 2006, pp. 205–220.

[3] 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 Symposium on Operating Systems Principles*, Stevenson, WA, October 2007, pp. 205–220.

[4] 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 Conference*, August 2008, pp. 1277–1288.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, April 2010.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM Symp. on Cloud computing*, Indianapolis, IN, June 2010, pp. 143–154.

[7] 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, June 2011, pp. 1071–1080.

[8] A. Silberstein, R. Sears, W. Zhou, and B. Cooper, "A batch of pnuts: Experiences connecting cloud batch and serving systems," in *ACM SIGMOD Conf*, Athens, Greece, June 2011, pp. 1101–1112.

[9] 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 Intl Conf Data Engineering*, Hannover, Germany, Apr. 2011, pp. 291–302.

[10] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. L, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore : Providing scalable, highly available storage for interactive services," in *Biennial Conf. on Innovative Data Systems Research*, Asilomar, CA, Jan. 2011, pp. 223–234.

[11] M. Mammarella, S. Hovsepian, and E. Kohler, "Modular data storage with anvil," in *ACM Symposium on Operating Systems Principles*, Big Sky, MO, 2009, pp. 147–160.

[12] D. E. Knuth, *The Art of Computer Programming: Searching and Sorting*, 2nd ed. Addison Wesley Longman, 1998, vol. 3.

[13] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys*, vol. 25, no. 2, pp. 73–170, June 1993.

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

[15] S. Büttcher and C. L. A. Clarke, "Hybrid index maintenance for contiguous inverted lists," *Information Retrieval*, vol. 11, pp. 197–207, June 2008.

[16] 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 Conf on File and Storage Technologies*, San Jose, CA, Feb. 2011, pp. 163–176.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003, pp. 29–43.

[18] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, September 1988.

[19] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, Feb. 1972.

[20] 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 Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2007, pp. 81–92.

[21] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro, "Cache-oblivious dynamic dictionaries with update/query tradeoff," in *ACM-SIAM Symposium on Discrete Algorithms*, Austin, TX, Jan. 2010, pp. 1448–1456.

[22] A. Twigg, A. Byde, G. Milos, T. Moreton, J. Wilkes, and T. Wilkie, "Stratified b-trees and versioned dictionaries," in *USENIX Workshop on Hot Topics in Storage and File Systems*, Portland, OR, June 2011.

[23] 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.

[24] 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.

[25] A. Tomasic, H. Garcia-Molina, and K. Shoens, "Incremental updates of inverted lists for text document retrieval," in *ACM SIGMOD Conference*, Minneapolis, Minnesota, May 1994, pp. 289–300.

[26] G. Margaritis and S. V. Anastasiadis, "Low-cost management of inverted files for online full-text search," in *ACM Conference on Information and Knowledge Management*, Hong Kong, Nov. 2009, pp. 455–464.

[27] 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, (accepted for publication).

[28] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *USENIX Conf. on Operating Systems Design and Implementation*, Vancouver, Canada, Oct. 2010, pp. 1–15.

[29] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004, pp. 105–120.

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

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

[32] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, Oct. 2010, pp. 47–60.

[33] D. Kossmann, T. Kraska, and S. Loesing, "An evaluation of alternative architectures for transaction processing in the cloud," in *ACM SIGMOD Conf.*, Indianapolis, IN, June 2010, pp. 579–590.

[34] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan, "Efficient bulk insertion into a distributed ordered table," in *ACM SIGMOD Conf.*, Vancouver, Canada, June 2008, pp. 765–778.

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

[36] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *ACM SIGMOD Conf.*, Providence, RI, June 2009, pp. 165–178.

[37] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2002.

[38] R. Low, "Cassandra under heavy write load," Acunu, Ltd., London, United Kingdom, Mar. 2011. [Online]. Available: http://www.acunu.com/blogs/richard-low/cassandra-under-heavy-write-load-part-i/

[39] 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.

[40] "Cheetah ns.2 data sheet: Lowest power, highest reliability for 3.5-inch tier 1 solutions," Seagate Tech LLC, 2009.

[41] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995, pp. 79–95.

[42] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *ACM SIGMOD Conf.*, Minneapolis, MI, May 1994, pp. 243–252.

[43] "Zoie: real-time search and indexing system built on apache lucene." [Online]. Available: http://code.google.com/p/zoie/wiki/ZoieMergePolicy