Serving Large-scale Batch Computed Data with Project Voldemort

Roshan Sumbaly Jay Kreps Lei Gao Alex Feinberg Chinmay Soman Sam Shah
LinkedIn

Abstract

Current serving systems lack the ability to bulk load massive immutable data sets without affecting serving performance. The performance degradation is largely due to index creation and modification as CPU and memory resources are shared with request serving. We have extended Project Voldemort, a general-purpose distributed storage and serving system inspired by Amazon's Dynamo, to support bulk loading terabytes of read-only data. This extension constructs the index offline, by leveraging the fault tolerance and parallelism of Hadoop. Compared to MySQL, our compact storage format and data deployment pipeline scales to twice the request throughput while maintaining sub 5 ms median latency. At LinkedIn, the largest professional social network, this system has been running in production for more than 2 years and serves many of the data-intensive social features on the site.

1 Introduction

Many social networking and e-commerce web sites contain data-derived features, which usually consist of some data mining application offering insights to the user. Typical features include: "People You May Know," a link prediction system attempting to find other users you might know on the social network (Figure 1a); collaborative filtering, which showcases relationships between pairs of items based on the wisdom of the crowd (Figure 1b); various entity recommendations; and more. LinkedIn, the largest professional social network with, as of writing, more than 135 million members, consists of these and more than 20 other data-derived features.

The feature data cycle in this context consists of a continuous chain of three phases: data collection, processing, and serving. The data collection phase usually involves log consumption, while the processing phase involves running algorithms on the output. Algorithms such as link prediction or nearest-neighbor computation output hundreds of results per user. For example, the "People You May Know" feature on LinkedIn runs on hundreds of terabytes of offline data daily to make these predictions.

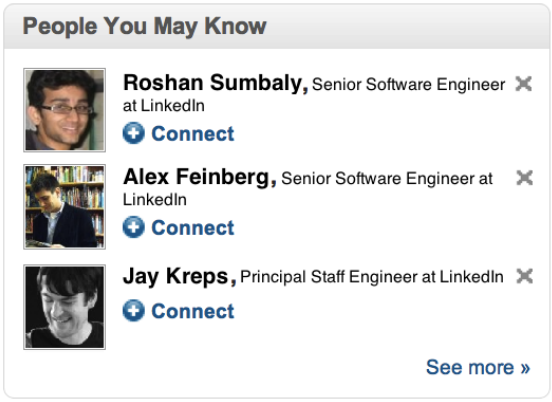
Due to the dynamic nature of the social graph, this derived data changes extremely frequently—requiring an almost complete refresh and bulk load of the data, while continuing to serve existing traffic with minimal additional latency. Naturally, this batch update should complete quickly to engender frequent pushes.

Interestingly, the nature of this complete cycle means that live updates are not necessary and are usually handled by auxiliary data structures. In the collaborative filtering use case, the data is purely static. In the case of "People You May Know", dismissed recommendations (marked by clicking "X") are stored in a separate data store with the difference between the computed recommendations and these dismissals calculated at page load.

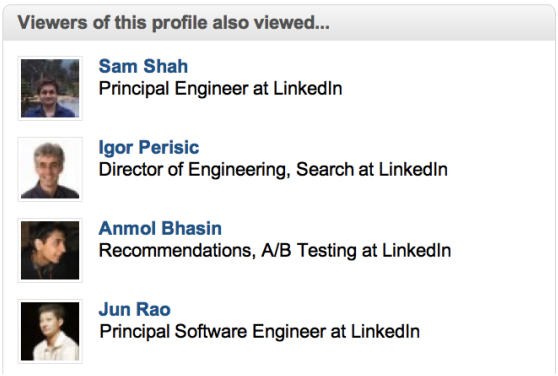
This paper presents read-only extensions to Project Voldemort, our key-value solution for the final serving phase of this cycle and discusses how it fits into our feature ecosystem. Voldemort, which was inspired by Amazon's Dynamo [7], was originally designed to support fast online read-writes. Our system leverages a Hadoop elastic batch computing infrastructure to build its index and data files, thereby supporting high throughput for batch refreshes. A custom read-only storage engine plugs into Voldemort's extensible storage layer. The Voldemort infrastructure then provides excellent live serving performance for this batch output—even during data refreshes.

Our system supports quick rollback, where data can be restored to a clean copy, minimizing the time in error if an algorithm should go awry. This helps support fast, iterative development necessary for new feature improvements. The storage data layout also provides the ability to grow horizontally by rebalancing existing data to new nodes without downtime.

Our system supports twice the request throughput versus MySQL while serving read requests with a median latency of less than 5 ms. At LinkedIn, this system has been running for over 2 years, with one of our largest



(a)



(b)

Figure 1: (a) The "People You May Know" module. (b) An example collaborative filtering module.

clusters loading more than 4 terabytes of new data to the site every day.

The key contributions of this work are:

- A scalable offline index construction, based on MapReduce [6], which produces partitioned data for online consumption
- Complete data cycle to refresh terabytes of data with minimum effect on existing serving latency
- Custom storage format for static data, which leverages the operating system's page cache for cache management

Voldemort and its read-only extensions are open source and are freely available under the Apache 2.0 license.

The rest of the paper is as follows. Section 2 first discusses related work. We then provide an architectural overview of Voldemort in Section 3. We follow with a discussion in Section 4 of existing solutions that we tried, but found insufficient for bulk loading and serving largely static data. Section 5 describes Voldemort's read-only extensions, including our new storage format and how data and indexes are built offline and loaded into the system. Section 6 presents experimental and production results evaluating our solution. We close with a discussion of future directions in Section 7.

2 Related Work

MySQL [16] is a common serving system used in various companies. The two most commonly used MySQL storage engines, MyISAM and InnoDB, provide bulk loading capabilities into a live system with the `LOAD DATA INFILE` statement. MyISAM provides a compact on-disk structure and the ability to delay recreation of the index after the load. However, these benefits come at the expense of requiring considerable memory to maintain a special tree-like cache during bulk loading. Additionally, the MyISAM storage engine locks the complete table for the duration of the load, resulting in queued requests. In comparison, InnoDB supports row-level locking, but its on-disk structure requires considerable disk space and its bulk loading is orders of magnitude slower than MyISAM.

Considerable work has been done to add bulk loading ability to new shared nothing [22] cluster databases similar to Voldemort. Silberstein et al. [19] introduce the problem of bulk insertion into range-partitioned tables in PNUTS [4], which tries to optimize data movement between machines and total transfer time by adding an extra planning phase to gather statistics and prepare the system for the incoming workload. In an extension of that work [20], Hadoop is used to batch insert data into PNUTS in the reduce phase. Both of these approaches optimize the time for data loading into the live system, but incur latency degradation on live serving due to multi-tenant issues with sharing CPU and memory during the full loading process. This is a significant problem with very large data sets, which even after optimizations, might take hours to load.

Our system alleviates this problem by moving the construction of the indexes to an offline system. MapReduce [6] has been used for this offline construction in various search systems [14]. These search layers trigger builds on Hadoop to generate indexes, and on completion, pull the indexes to serve search requests.

This approach has also been extended to various databases. Konstantinou et al. [10] and Barbuzzi et al. [2] suggest building HFiles offline in Hadoop, then shipping them to HBase [9], an open source database modeled after BigTable [3]. These works do not explore the data pipeline, particularly data refreshes and rollback.

The overall architecture of Voldemort was inspired from various DHT storage systems. Unlike the previous DHT systems, such as Chord [21], which provide $O(log\ N)$ lookup, Voldemort's lookups are $O(1)$ because the complete cluster topology is stored on every node. This information allows clients to bootstrap from a random node and direct requests to exact destination nodes. Similar to Dynamo [7], Voldemort also supports per tuple-based replication for availability purposes. Updating replicas is easy in the batch scenario because they

are precomputed and loaded into the Voldemort cluster at once. The novelty of Voldemort, compared to Dynamo, is our custom storage engine for bulk-loaded data sets.

3 Project Voldemort

A Voldemort *cluster* can contain multiple *nodes*, each with a unique identifier. A physical host can run multiple nodes, though at LinkedIn we maintain a one-to-one mapping. All nodes in the cluster have the same number of *stores*, which correspond to database tables. General usage patterns have shown that a site-facing feature can map to one or more stores. For example, a feature dealing with group recommendations will map to two stores: one recording a member id to recommended group ids and another recording a group id to its corresponding description. Every store has the following list of configurable parameters, which are identical to Dynamo's parameters:

- *Replication factor* (N): Number of nodes which each key-value tuple is replicated.
- *Required reads* (R): Number of nodes Voldemort reads from, in parallel, during a get before declaring a success.
- *Required writes* (W): Number of node responses Voldemort blocks for, before declaring success during a put.
- *Key/Value serialization and compression*: Voldemort can have different serialization schemas for key and value. For the custom batch data use case, Voldemort uses a custom binary JSON format. Voldemort also supports per tuple-based compression. Serialization and compression is completely handled by a component that resides on the client side with the server only dealing with raw byte arrays.
- *Storage engine type*: Voldemort supports various read-write storage engine formats: Berkeley DB Java Edition [15] and MySQL [16]. Voldemort also supports a custom read-only storage engine for bulk-loaded data.

Every node in the cluster stores the same 2 pieces of metadata: the complete cluster topology and the store definitions.

Voldemort has a pluggable architecture, as shown in Figure 2. Each box represents a module, all of which share the same code interface. Each module has exactly one functionality, making it easy to interchange modules. For example, we can have the routing module on either the client side or the server side. Functional separation at the module level also allows us to easily mock these modules for testing purposes—for example, a mocked up storage engine backed by a hash map for unit tests.

Many of our modules have been inspired by the original Dynamo paper. Starting from the top of the Voldemort stack, our client has a simple get and put API. Every tuple is replicated for availability, with each value having

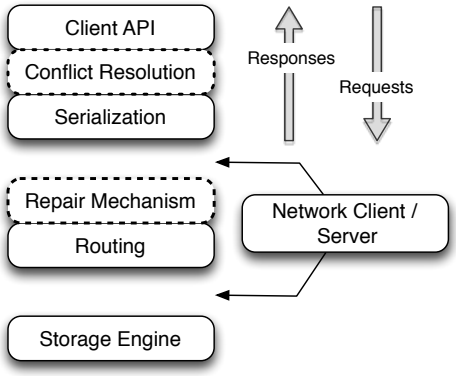
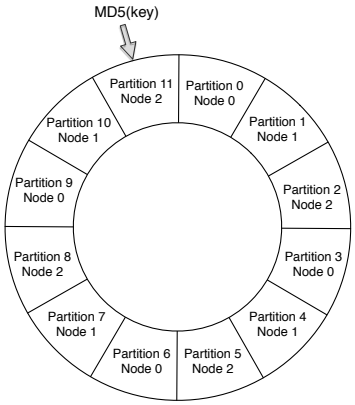


Figure 2: Voldemort architecture containing modules for a single client and server. The dotted modules are not used by the read-only storage engine.



Partition hashed to	Preference list		
	N0	N1	N2
0	**0**	1	
1		**1**	2
2	3		**2**
3	**3**	4	
4		**4**	5
5	6		**5**
6	**6**	7	
7		**7**	8
8	9		**8**
9	**9**	10	
10		**10**	11
11	0		**11**

Figure 3: Simple hash ring cluster topology for 3 nodes and 12 partitions. The preference list generation for a key hashing to partition 11, for a store with N=2, would jump the ring clockwise to place the other $N-1$=1 replica on partition 0. The table shows the preference list generated for every hashed partition. The primary partitions have been highlighted in bold.

vector clock [11] versioning. The "conflict resolution" and "repair mechanism" layer, used only by the read-write storage engines, deal with inconsistent replicas. This does not apply to read-only stores because Voldemort updates all the replicas of a key in a store at once, keeping them in sync.

The "routing" module deals with partitioning as well as replication. Our partitioning scheme is similar to Dynamo's, wherein Voldemort splits the hash ring into equal size partitions, assigns them unique ids, and then maps them to nodes. This ring is then shared with all the stores; that is, changes in the mapping require changes to all the stores. To generate the *preference list* (the list of partition ids where the replicas will be stored), we first hash the key (using MD5) to a range belonging to a partition and then continue jumping the ring clockwise to find $N-1$ partitions belonging to different nodes. For example, for a store with N=2 and partition mapping as shown in Fig-

ure 3, the preference list for a key hashing to partition 11 will be (Partition 11, Partition 0).

The last module, the pluggable storage layer, has the same *get* and *put* functions, along with the ability to stream data out. In addition to running the full stack from Figure 2, every node also runs an administrative service that allows the execution of following privileged commands: add or remove a store, stream data out, and trigger read-only store operations.

Voldemort supports two routing modes: server-side and client-side routing. Client-side routing, the more commonly used routing strategy, requires an initial "bootstrap" step, wherein it retrieves the metadata required for routing (cluster topology and store definitions) by load balancing to a random node. Once the metadata has been retrieved by the client, one fewer hop is necessary compared to server-side routing, because the replica locations can be calculated on the fly. However, as we will further explain in Section 5.7, client-side routing makes rebalancing of data complicated, because we now need a mechanism to update the cluster topology metadata on the live clients.

4 Alternative Approaches

Before we started building our own custom storage engine, we decided to evaluate the existing read-write storage engines supported in Voldemort, namely, MySQL and Berkeley DB. Our criteria for success was the ability to bulk load massive data sets with minimal disk space overhead, while still serving live traffic.

4.1 Shortcomings of Alternative Approaches

The first approach we tried was to perform multiple *put* requests. This naïve approach is problematic as every request results in an incremental change to the underlying index structure (in most cases, a B+ tree), which in turn, results in many disk seeks. To solve this problem, MySQL provides a `LOAD DATA` statement that tries to bulk update the underlying index. Unfortunately, using this statement for the MyISAM storage engine locks the entire table. InnoDB instead executes this statement with row-level locking, but experiences substantial disk space overhead for every tuple. However, to achieve MyISAM-like bulk loading performance, InnoDB prefers data ordered by primary key. Achieving fast load times with low space overhead in Berkeley DB requires several manual and non-scalable configuration changes, such as shutting down cleaner and checkpointer threads.

The next solution we explored was to bulk load into a different MySQL table on the same cluster and use views to transparently swap to the new table. We used the MyISAM storage engine, opting to skip InnoDB due to the large space requirements. This approach solves the locking problem, but still hurts serving latency during the load due to pressure on shared CPU and memory resources.

We then tried completely offloading the index construction to another system as building the index on the serving system has isolation problems. We leveraged the fact that MyISAM allows copying of database files from another node into a live database directory, automatically making it available for serving. We bulk load to a separate cluster and then copy the resulting database files over to the live cluster. This two-step approach requires the extra maintenance cost of a separate MySQL cluster with exactly the same number of nodes as the live one. Additionally, the inability to load compressed data in the bulk load phase means data is copied multiple times between nodes: first, as a flat file to the bulk load cluster; then as an internal copy during the `LOAD` statement; and finally, as a raw database file copy to the actual live database. These copies make the load more time-consuming.

The previous solution was not ideal, due to its dependency on redundant MySQL servers and the resulting vulnerability to failure downtime. To address this shortcoming, the next attempted approach used the inherent fault tolerance and parallelism of Hadoop and built individual node/partition-level data stores, which could be transferred to Voldemort for serving. A Hadoop job reads data from a source in HDFS [18], repartitions it on a per-node basis, and finally writes the data to individual storage engines (for example, Berkeley DB) on the local filesystem of the reducer phase Hadoop nodes. The number of reducers equals the number of Voldemort nodes, but could have easily been further split on a per-partition basis. This data is then read from the local filesystem and copied onto HDFS, where it can be fetched by Voldemort. The benefit of this approach is that it leverages Hadoop's parallelism to build the indexes offline; however, it suffers from an extra copy from the local filesystem on the reducer nodes to HDFS, which can become a bottleneck with terabytes of data.

4.2 Requirements

The lack of off-the-shelf solutions, along with the inefficiencies of the previous experiments, motivated the building of a new storage engine and deployment pipeline with the following properties.

- *Minimal performance impact on live requests*: The incoming *get* requests to the live store must not be impacted during the bulk load. There is a trade-off between modifying the current index on the live server and a fast bulk load—quicker bulk loads result in increased I/O, which in turn hurts performance. As a result, we should completely rebuild the index offline and also throttle fetches to Voldemort.
- *Fault tolerance and scalability*: Every step of the data load pipeline should handle failures and also

scale horizontally to support future expansion without downtime.

- *Rollback capability*: The general trend we notice in our business is that incorrect or incomplete data due to algorithm changes or source data problems needs immediate remediation. In such scenarios, running a long batch load job to repopulate correct data is not acceptable. To minimize the time in error, our storage engine must support very fast rollback to a previous good state.
- *Ability to handle large data sets*: The easy access to scalable computing through Hadoop, along with the growing use of complex algorithms has resulted in large data sets being used as part of many core products. Classic examples of this, in the context of social networks, include storing relationships between a pair of users, or between users and an entity. When dealing with millions of users, these pairs can easily reach billions of tuples, motivating our storage engine to support terabytes of data and perform well under a large data to memory ratio.

5 Read-only Extensions

To satisfy the requirements laid out in Section 4.2, we built a new data deployment pipeline as shown in Figure 4. We use the existing Voldemort architecture to plug in a new storage engine with a compact custom format (Section 5.1). For many of LinkedIn's user-facing features, data is generated by algorithms run on Hadoop. For example, the "People You May Know" feature runs a complex series of Hadoop jobs on log data. We thus leverage Hadoop as the computation layer for building the index as its MapReduce component handles failures while HDFS replication provides availability. After the algorithm's computation completes, a driver program coordinates a refresh of the data. As shown in steps 1 and 2 in Figure 4, it triggers a build of the output data in our custom storage format and stores it on HDFS (Section 5.2). This data is kept in versioned format (Section 5.3) after being fetched by Voldemort nodes in parallel (Section 5.4), as demonstrated in steps 3 and 4. Once fetched and swapped in, as displayed in steps 5 and 6, the data on the Voldemort nodes is ready for serving (Section 5.5). This section describes this procedure in detail. We also discuss real world production scenarios such as data schema changes (Section 5.6) and the no-downtime addition of new nodes (Section 5.7).

5.1 Storage Format

Many storage formats try to build data structures that keep the data memory resident in the process's address space, ignoring the effects of the operating system's page cache. The several orders of magnitude latency gap between page cache and disk means that most of the real

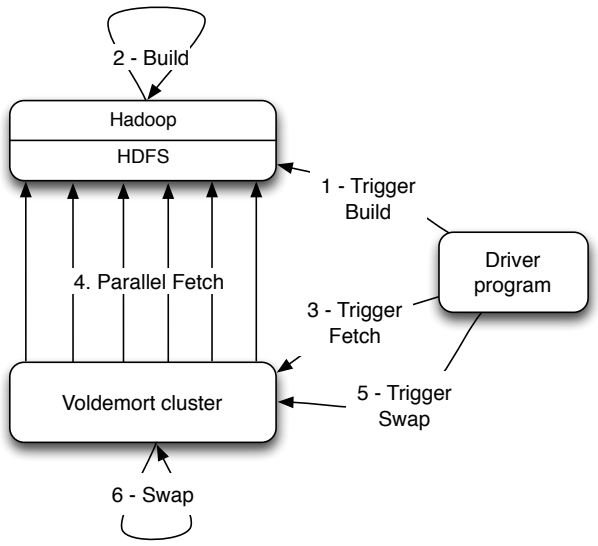


Figure 4: Steps involved in the complete data deployment pipeline. The components involved include Hadoop, HDFS, Voldemort, and a driver program coordinating the full process. The "build" step works on the output of the algorithm's job pipeline.

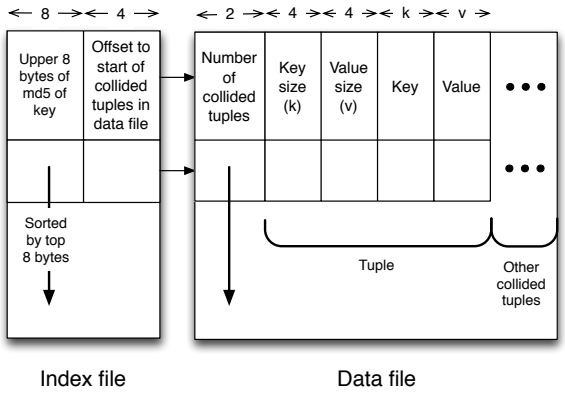


Figure 5: Read-only data is split into multiple chunk buckets, each of which is further split into multiple chunk sets. A chunk set contains an index file and a data file. The diagram shows the data layout in these files. The numbers at the top are sizes in bytes.

performance benefit by maintaining our own structure is for elements already in the page cache. In fact, this custom structure may even start taking memory away from the page cache. This potential interference motivated the need for our storage engine to exploit the page cache instead of maintaining our own complex heap-based data structure. Because our data is immutable, Voldemort memory maps the entire index into the address space. Additionally, because Voldemort is written in Java and runs on the JVM, delegating the memory management to the operating system eases garbage collection tuning.

To take advantage of the parallelism in Hadoop during generation, we split the input data destined for a particular node into multiple *chunk buckets*, which in turn are split into multiple *chunk sets*. Generation of multiple chunk

Node Id	Chunk buckets
0	0_0, 3_0, 6_0, 9_0, 2_1, 5_1, 8_1, 11_1
1	1_0, 4_0, 7_0, 10_0, 0_1, 3_1, 6_1, 9_1
2	2_0, 5_0, 8_0, 11_0, 1_1, 4_1, 7_1, 10_1

Table 1: Every Voldemort node is responsible for chunk buckets based on the primary partition and replica id. This table shows the node id to chunk bucket mapping for the cluster topology defined in Figure 3.

sets can then be done independently and in parallel. A chunk bucket is defined by the primary partition id and replica id, thereby giving it a unique identifier across all nodes. For a store with N=2, the replica id would be either 0 for the primary replica or 1 for the secondary replica. For example, the hashed key in Figure 3 would fall into buckets 11_0 (on node 2) and 11_1 (on node 0). Table 1 summarizes the various chunk buckets for a store with N=2 and cluster topology as shown in Figure 3. Our initial design had started with the simpler design of having one chunk bucket per-node (that is, multiple chunk sets stored on a node with no knowledge of partition/replica), but the current smaller granularity is necessary to aid in rebalancing (Section 5.7).

The number of chunk sets per bucket is decided during generation on the Hadoop side. The default value is one chunk set per bucket, but can be increased by the store owner for more parallelism. The only limitation is that a very large value for this parameter would result in multiple small-sized files—a scenario that HDFS does not handle efficiently. As shown in Figure 5, a chunk set includes a data file and an index file. The standard naming convention for all our chunk sets is *partition id_replica id_chunk set id*.{data, index} where *partition id* is the id of the primary partition, *replica id* is a number between 0 to $N-1$, and *chunk set id* is a number between 0 to the predefined number of sets per bucket-1.

The index file is a compact structure containing the sorted upper 8 bytes of the MD5 of the key followed by the 4 byte offset of the corresponding value in the data file. This simple sorted structure allows us to leverage Hadoop's ability to return sorted data in the reducers. Further, preliminary tests also showed that the index files were generally orders of magnitude smaller than the data files and hence, could fit into the page cache. The use of MD5, instead of any other hash function yielding uniformly distributed values, was an optimization to reuse the calculation from the generation of the preference list.

We had initially started by using the full 16 bytes of the MD5 signature, but saw performance problems as the number of stores grew. In particular, the indexes for all stores were not page cache resident, and thrashing behavior was seen for certain stores due to other high-throughput stores. To alleviate this problem, we needed to cut down on the amount of data being memory mapped, which could be achieved by reducing the available key-space and accepting collisions in the data file.

Our optimization to decrease key-space can be mapped to the classic birthday paradox: if we want to retrieve n random integers from a uniform distribution of range $[1, x]$, the probability that at least 2 numbers are the same is:

$$1 - e^{\frac{-n(n-1)}{2x}} \tag{1}$$

Mapping this to our common scenario of stores keyed by member id, n is our 135 million member user base, while the initial value of x is $2^{128} - 1$ (16 bytes of MD5). The probability of collision in this scenario is close to 0. A key-space of 4 bytes (that is, 32 bits) yields an extremely high collision probability of:

$$1 - e^{\frac{(-135*10^6*(135*10^6-1)}{2*(2^{32}-1)}} \sim 1 \tag{2}$$

Instead, a compromise of 8 bytes (that is, 64 bits) produces:

$$1 - e^{\frac{(-135*10^6*(135*10^6-1)}{2*(2^{64}-1)}} < 0.0004 \tag{3}$$

The probability of more than one collision is even smaller. As a result, by decreasing the number of bytes of the MD5 of the key, we were able to cut down the index size by 40%, allowing more stores to fit into the page cache. The key-space size is an optional parameter the store owner can set depending on the semantics of the data. Unfortunately, this optimization came at the expense of having to save the keys in the data file to use for lookups and handle rare collisions.

The data file is also a very highly-packed structure where we store the number of collided tuples followed by a list of collided tuples *(key size, value size, key, value)*. The order of these multiple lists is the same as the corresponding 8 bytes of MD5 of key in the index file. Here, we need to store the key bytes instead of the MD5 in the tuples to distinguish collided tuples during reads.

5.2 Chunk Set Generation

Construction of the chunk sets for all the Voldemort nodes is a single Hadoop job; the pseudo-code representation is shown in Figure 6. The Hadoop job takes as its input the number of chunk sets per bucket, cluster topology, store definition, and the input data location on HDFS. The job then takes care of replication and partitioning, finally saving the data into separate node-based directories.

At a high level, the mapper phase deals with the partitioning of the data depending on the routing strategy; the partitioner phase redirects the key to the correct reducer and the reducer phase deals with writing the data to a single chunk set. Due to Hadoop's generic `InputFormat` mechanics, any source data can be converted to Voldemort's serialization format. The mapper phase emits the

```
Global Input: Num Chunk Sets: Number of chunk sets per bucket
Global Input: Replication Factor: Tuple replicas for the store
Global Input: Cluster: The cluster topology
Function: TopBytes(x,n): Returns top n bytes of x
Function: MD5(x): Returns MD5 of x
Function: PreferenceList(x): Partition list for key x
Function: Size(x): Return size in bytes
Function: Make*(x): Convert x to Voldemort serialization

Input: K/V: Key/value from HDFS files
Data: K'/V': Transformed key/value into Voldemort serialization
map (K, V)
    K' ← MakeKey(K)
    V' ← MakeValue(V)
    Replica Id ← 0
    MD5K' ← MD5(K')
    KOut ← TopBytes( MD5K', 8 )
    foreach Partition Id ∈ PreferenceList(MD5K') do
        Node Id ← PartitionToNode(Partition Id)
        emit(KOut, [Node Id, Partition Id, Replica Id, K', V'])
        Replica Id ← Replica Id + 1
    end
end

Input: K: Top 8 bytes of MD5 of Voldemort key
Input: V: [Node Id, Partition Id, Replica Id, K', V']
partition (K, V): Integer
    Chunk Set Id ← TopBytes( MD5(V.K'), Size(Integer) )
                                % Num Chunk Sets
    Bucket Id ← V.Partition Id * Replication Factor +
                                V.Replica Id
    return Bucket Id * Num Chunk Sets + Chunk Set Id
end

Input: K/V: Same as partitioner
Data: Position: Continuous offset into data file. Initialized to 0
reduce (K, Iterator<V> Values)
    WriteIndexFile(K)
    WriteIndexFile(Position)
    WriteDataFile(Values.length)
    Position += Size(Short)
    foreach V ∈ Values do
        WriteDataFile( Size(V.K') )
        WriteDataFile( Size(V.V') )
        WriteDataFile(V.K')
        WriteDataFile(V.V')
        Position += Size(V.K') + Size(V.V') + Size(2*Integer)
    end
end
```

Figure 6: MapReduce pseudo-code used for chunk set generation.

upper 8 bytes of the MD5 of the Voldemort key N times as the map phase key with the map phase value equal to a grouped tuple of node id, partition id, replica id, and the Voldemort key and value.

The custom partitioner generates the chunk set id within a chunk bucket from this key. Due to the fair distribution of MD5, we partition the data destined for a bucket into sets with a mod of the 4 bytes of MD5 by the predefined number of chunk sets per bucket. This generated chunk set id, along with the partition id and replication factor of the store, is used to route the data further to the correct reducer.

Finally, every reducer is responsible for a single chunk set, meaning that by having more chunk sets, build phase parallelism can be increased. Hadoop automatically sorts input based on the key in the reduce phase, so data arrives in the order necessary for the index and data files, which can be constructed as simple appends on HDFS with no extra processing required. The data layout on HDFS is a directory for each Voldemort node, with the nomenclature of `node-`*`id`*.

5.3 Data Versioning

Before we describe how the generated chunk set files are copied from HDFS, it is essential to understand their storage layout on the Voldemort nodes. This layout is crucial because one of our requirements is the ability to perform instantaneous rollback of data. That is, every time a new copy of the complete data set is created, the system needs to demote the previous copy to an earlier state.

Every store is represented by a directory, which in turn contains directories corresponding to "versions" of the data. A symbolic link per store is used to point to the current serving version directory. Because the data in all version directories except the serving one is inactive, we are not affecting page cache usage and latency. Also, with disks becoming cheaper and providing very fast sequential writes compared to random reads, keeping these previous copies (the number of which is configurable) is beneficial for quick rollback. Every version directory (named `version-`*`no`*) has a configurable number associated with it, which should monotonically increase with every new fetch. A commonly used example for the version number is the timestamp of push.

Swapping in a new data version on a single node is done as follows: copy into a new version directory, close the current set of active chunk set files, open the chunk set files from the new version, memory map all the index files, and change the symbolic link to the new version. The entire operation is coordinated using a read-write lock. A rollback follows the same sequence of steps, except that files are opened in an older version directory. Both of these operations are very fast as they are purely metadata operations: no data reads take place.

5.4 Data Load

Figure 4 shows the complete data loading and swapping process for an individual store. Multiple stores can run this entire process concurrently.

The initiator of this complete construction is a standalone driver program that constructs, fetches, and swaps the data. This program starts the process by triggering the Hadoop job described in Section 5.2. The job generates the data on a per-node basis and stores it in HDFS. While streaming the data to HDFS, the Hadoop job also calculates a checksum on a per-node basis by storing a running MD5 on the individual MD5s of all the chunk set files.

Once the Hadoop job is complete, the driver triggers a fetch request on all the Voldemort nodes. This request is received by each node's "administrative service," which then initiates a parallel fetch from HDFS into its respective new version directory. While the data is being streamed from HDFS, the checksum is validated with the checksum from the build step. Voldemort uses a pull model, rather than a push model, as it allows throttling of this fetch in case of latency spikes.

After the data is available on each node in their new version directory, the driver triggers a swap operation (described in Section 5.3) on all nodes. On one of LinkedIn's largest clusters, described in Table 3, this complete operation takes around 0.012 ms on average with the worst swap time of around 0.050 ms. Also, to provide global atomic semantics, the driver ensures that all the nodes have successfully swapped their data, rolling back the successful swaps in case of any other swap failures.

5.5 Retrieval

To find a key, the client generates the preference list and directs the request to the individual nodes. The following is a sketch of the algorithm to find data once it reaches a particular node.

1. Calculate the MD5 of the key.
2. Generate the (a) primary partition id, (b) replica id (the replica being searched when querying this node), and (c) chunk set id (the first 4 bytes of MD5 of the key modulo the number of chunk sets per bucket).
3. Find the corresponding active chunk set files (a data file and an index file) using the 3 variables from the previous step.
4. Perform a search using the top 8 bytes of MD5 of the key as the search key in the sorted index file. Because there are fixed space requirements for every key (12 bytes: 8 bytes for key and 4 bytes for offset), this search does not require internal pointers within the index file. For example, the data location of the i-th element in the sorted index is simply a jump to the offset $12 \cdot i + 8$.
5. If found, read the corresponding data location from the index file and jump to the location in the data file. Iterate through any potential collided tuples, comparing keys, and return the corresponding value on key match.

The most time-consuming step is to search the index file. A binary search in an index of 1 million keys can result in around 20 key comparisons; if the index file is not cached, then 20 disk seeks are required to read one value. As a small optimization, while fetching the files from HDFS, Voldemort fetches the index files after all data files to aid in keeping the index files in the page cache.

Rather than binary search, another retrieval strategy for sorted disk files is interpolation search [17]. This search strategy uses the key distribution to predict the approximate location of the key, rather than halving the search space for every iteration. Interpolation search works well for uniformly distributed keys, dropping the search complexity from $O(log\ N)$ to $O(log\ log\ N)$. This helps in the uncached scenario by reducing the number of disk seeks.

We also evaluated other strategies like Fast [12] and Pegasus [8]. As proved in Manolopoulos and Poulakas [13], most of these are better suited for non-uniform distributions. As MD5 (and its subsets) provides a fairly representative uniform distribution, there will be minimal speedup from these techniques.

5.6 Schema Upgrades

As product features evolve, there are bound to be changes to the underlying data model. For example, an administrator may want to add a new dimension to a store's value or do a complete non-backwards compatible change from storing an array to a map. Because our data is static and the system does a full refresh, Voldemort supports the ability to change the schema of the key and value without downtime. For the client to transparently handle this change, the binary JSON serialization format adds a special version byte during serialization. The mapping of version byte to schema is saved in the store definition metadata. The updated store definition metadata can be propagated to clients by forcing a rebootstrap. Introduction of a new schema after a push is now discovered by the client during deserialization as it can look up the new information after reading the version byte. Similarly, during rollback, the client toggles to an older version of schema and is able to read the data with no downtime.

5.7 Rebalancing

Over time as new stores get added to the cluster, the disk to memory ratio increases beyond initial capacity planning, resulting in increased read latency. Our data being static, the naïve approach of starting a new larger cluster, repushing the data, and switching clients does not scale as it requires massive coordination of multiple clients communicating with many stores.

This necessitates the need to transparently and incrementally add capacity to the cluster independent of data pushes. The rebalancing feature allows us to add new nodes to a live cluster without downtime. This feature was initially written for read-write stores but easily fits into the read-only cycle due to the static nature and fine-grained replication of the data. Our smallest unit of rebalancing is a partition. In other words, the addition of a new node translates to giving the ownership of some partitions to that node. The rebalancing process is run by a tool that coordinates the full process.

The following describes the rebalancing strategy during the addition of a new node. First, the rebalancing tool is provided with the future cluster topology metadata, and with this data, it generates a list of all primary partitions that need to be moved. The tool moves partitions in small batches so as to checkpoint and not refetch too much data in case of failure.

For every small batch of primary partitions, the system generates an intermediate cluster topology metadata, which is the current cluster topology plus changes in ownership of the batch of partitions moved. Voldemort must take care of all secondary replica movements that might be required due to the primary partition movement. A plan is generated that lists the set of donating and stealing node-id pairs along with the chunk buckets being moved. With this plan, the rebalancing tool starts asynchronous processes (through the administrative service) on the stealer nodes to copy all chunk sets corresponding to the moving chunk buckets from their respective donor nodes. Rebalancing works only on the active version of the data, ignoring the previous versions. During this copying, the nodes go into a "rebalancing state" and are not allowed to swap any new data. Here it is important to note that the granularity of the bucket selected makes this process as simple as copying files. If buckets were defined on a per-node basis (that is, have multiple chunk sets on a per-node basis), the system would have had to iterate over all the keys on the node and find the keys belonging to the moving partition, finally running an extra merge step to coalesce with the live index on the stealer node's end.

Once the fetches are complete, the rebalancing tool updates the intermediate cluster topology on all the nodes while also running the swap operation, described in Section 5.3, for all the stores on the stealer and donor nodes. The entire process repeats for every batch of primary partitions.

The intermediate topology change also needs to be propagated to all the clients. Voldemort propagates this information as a lazy process where the clients still use the old metadata. If they contact a node with a request for a key in a partition that the node is no longer responsible for, a special exception is propagated, which results in a rebootstrap step along with a retry of the previous request.

The rebalancing tool has also been designed to handle failure scenarios elegantly. Failure during a fetch is not a problem as no new data has been swapped. However, failure during the topology change and swap phase on some nodes requires (a) changing the cluster topology to the previous good cluster topology on all nodes and (b) rolling back the data on nodes that had successfully swapped.

Table 2a shows the new preference list generation when a new node is introduced to a cluster with the original partition mapping as in Figure 3. For simplicity, we show

Partition hashed to	Preference list		
	N0	N1	N2
0	***0***	1	
1		***1***	2
2	3		***2***
3	***3***	4	
4		***4***	5
5	6		***5***
6	***6***	7	
7		***7***	8
8	9		***8***
9	***9***	10	
10		***10***	11
11	0		***11***

Partition hashed to	Preference list			
	N0	N1	N2	N3
0	***0***	1		
1		***1***	2	
2			***2***	3
3		4		***3***
4		***4***	5	
5	6		***5***	
6	***6***	7		
7		***7***	8	
8	9		***8***	
9	***9***	10		
10		***10***	11	
11	0		***11***	

(a)

Stealer Node Id	Donor Node Id	Chunk buckets to steal
3	0	3_0, 2_1

(b)

Table 2: (a) Change of preference list generation after addition of 4th node (node id 3) to the cluster defined by ring in Figure 3. The highlighted cells show how moving partition 3 to this new node results in secondary movement of keys hashing to partition 2. (b) Rebalancing plan generated for addition of a new node.

an imbalanced move of only one partition, partition 3, to the new node 3. Table 2b shows the plan that would be generated during rebalancing. The movement of partition 3 results in secondary movement for partition 2 due to node mapping changes in its preference list.

6 Evaluation

Our evaluation answers the following questions:

- Can the system rapidly deploy new data sets?
- What is the read latency, and does it scale with data size and nodes?
- What is the impact on latency during a new data deployment?

We use a simulated data set where the key is a long integer between 0 and a varying number and the value is a fixed size 1024 byte random string. All tests were run on Linux 2.6.18 machines with Dual CPU (each having 64-bit 8 cores running at 2.67 GHz), 24 GB of RAM, 6 disk RAID-10 array and Gigabit Ethernet. We used Community Edition version 5.0.27 and the MyISAM storage engine for all the MySQL tests.

As the read-only storage engine relies on the operating system's page cache, we allocated only 4 GB JVM heap. Similarly, as MyISAM uses a special key cache for index blocks and the page cache for data blocks, we chose the same 4 GB for *key_buffer_size*.

6.1 Build Times

One of the important goals of Voldemort is rapid data deployment, which means the build and push phase must

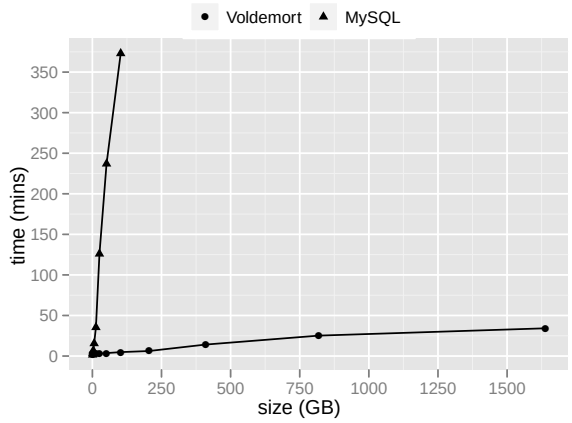


Figure 7: The time to complete the build for the random data set. We vary the input data size by increasing the number of tuples. We terminated the MySQL test early due to prolonged data load time.

Figure 8: Single node median read latency taken at 1 minute intervals since the swap. The distribution of requests is uniform. The slope of the graph shows the rate of cache warming.

be fast. Push times are entirely dependent on available network bandwidth, so we focus on build times.

We define the build time in the case of Voldemort as the time starting from the first mapper to the end of all reducers. The number of mappers and reducers was fixed across runs to steady the amount of parallelism and generate fixed number of chunk sets per bucket.

In the case of MySQL, the build time is the completion time of the `LOAD DATA INFILE` command on an empty table. This metric ignores the time it took to convert the data to TSV and copy it to the MySQL node. We applied several optimizations to make MySQL faster, including increasing the MySQL bulk insert buffer size and the MyISAM specific sort buffer size to 256 MB each, and also delaying the re-creation of the index to a latter time by running the `ALTER TABLE...DISABLE KEYS` statement before the load.

Figure 7 shows the build time as we increased the size of the input data set. As is clearly evident, MySQL exhibits extremely slow build times because it buffers changes to the index before flushing it to the disk. Also, due to the incremental changes required to the index on disk, MySQL does roughly 1.4 times more I/O than our implementation. This factor would increase if we had bulk loaded into a non-empty table.

6.2 Read Latency

Besides rapid data deployments, read latency must be acceptable and the system must scale with the number of nodes. In these experiments, we used 10 million requests with simulated keys following a uniform distribution between 0 to number of tuples in the data set.

We first measure how fast the index loads into the operating system's page cache. We ran tests on a 100 GB data set on a single node and reported the median latency after swap for a continuous stream of uniformly-distributed

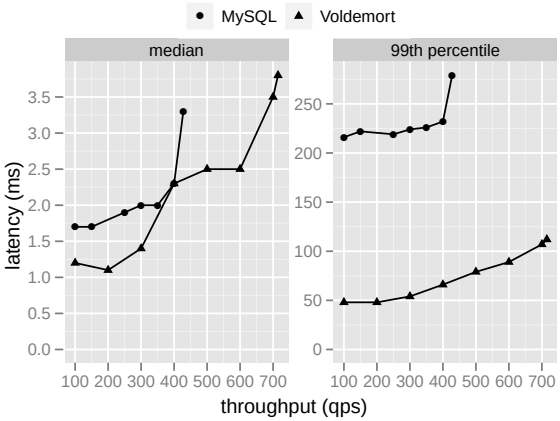


Figure 9: Single node read latency after warming up the cache. This figure shows the change in latency, for uniformly-distributed requests, as we vary the client throughput.

requests. For MySQL, we created a view on an existing table, bulk loaded into a new table, and swapped the view to the new table without stopping the requests. For our read-only storage engine, we used the complete data load process (described in Section 5.4), to swap new data. The single node was configured to have just one partition and one chunk set per bucket. We also compared the binary and interpolation search algorithms for the read-only storage engine.

Figure 8 shows the median latency, at 1 minute intervals, starting from the swap. MySQL starts with a very high median latency due to the uncached index and falls slowly to the stable 1.4 ms mark. Our storage engine starts with low latency because some indexes are already page cache resident, with the fetch phase from HDFS retrieving all index files after the data files. Binary search initially starts with a high median latency compared to interpolation, but the slope of the line is steeper, because

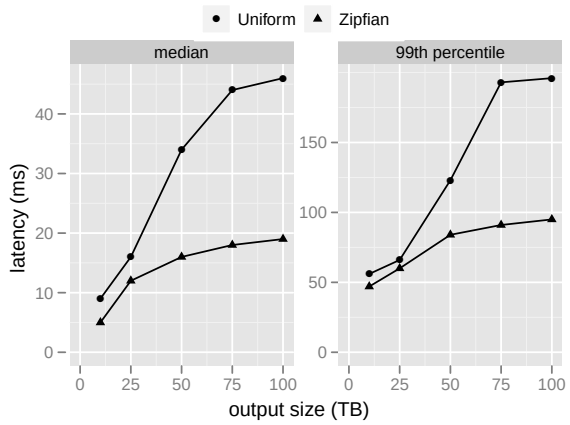


Figure 10: Client-side median latency with varying data size. This test was run on a 32 node cluster with 2 different request distributions.

binary search does an average of 8 lookups, thereby touching more parts of the index; interpolation search performs an average of only 1 lookup. While this results in an initial low read latency, it means that much of the index is uncached in the long run. Our production systems are currently running binary search due to this faster cache warming process. All numbers presented from this point for the read-only storage engine use binary search.

Figure 9 shows a comparison of Voldemort's performance compared to MySQL on the same 100 GB data set for varying throughput. The numbers reported are steady-state latencies; that is, latency reported after the cache is warmed. For comparison, the steady state latency for the read-only storage engine in Figure 9 is around 0.3 ms and is achieved around 90 minutes after the swap. We observed that the time to achieve this steady state, starting from the swap time, is linear in the size of the data set. We increased the client request throughput until the achieved throughput stopped increasing. These results indicate that our implementation scales to roughly twice the number of queries per second while maintaining the same median latency as MySQL.

To test whether our read-only extensions scale with the number of nodes, we evaluated read latency for the same random data set but spread over 32 machines and a store with N=1. The read tests were run for both uniform as well as a Zipfian distribution using YCSB [5], an open source framework for benchmarking cloud data serving systems, with the number of clients fixed at 100 threads. The Zipfian distribution ensures that some keys are more frequently accessed compared to others, simulating the general site visiting patterns of most websites [1]. Figure 10 shows the overall client-side median latency while varying the data set sizes. Querying for frequently accessed keys naturally aids caching certain sections of the indexes, thereby exhibiting an overall lower latency for Zipfian compared to the uniform distribution. We do not report numbers for a store with N>1 because latency is a function of data size and is independent of the replication factor. The results indicate that the system scales with the data set size and the number of nodes. As the data set size increases, we are decreasing the memory to data ratio, affecting read performance. Reducing latency in this case would require adding memory or additional nodes. Users can tune this ratio to achieve the desired latency versus the necessary hardware footprint.

Number of nodes	25
Total (active + backup) data size per node	940 GB
RAM per node	48 GB
Current active data to memory ratio	$\sim$ 10:1
Total number of stores	123
Replication factor for all stores	2
Largest store size (active)	4.15 TB
Smallest store size (active)	700 KB
Max number of store swaps per day	76

Table 3: Statistics for one of LinkedIn's read-only clusters.

6.3 Production Workloads

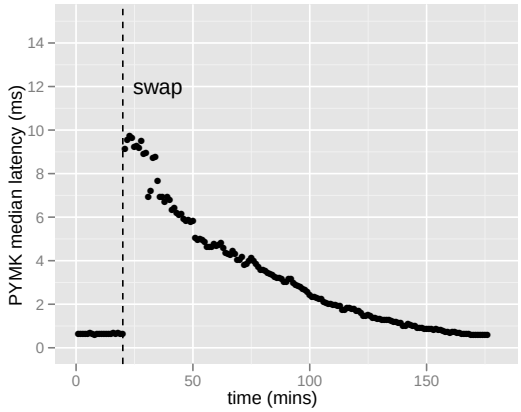
Finally, we show the production performance data for two user-facing features: "People You May Know" (Figure 1a) and collaborative filtering (Figure 1b):

- *People You May Know (PYMK) data set*: Users are presented with a suggested set of other users they might know and would like to connect with. This information is kept as a store where the key is the user's id and the value is a list of integer recommended user ids and a float score.
- *Collaborative filtering (CF) data set*: This feature shows other profiles viewed in the same session as the visited member's profile. The value is a list of two integer ids, a string indicating the entity type, and a float score.

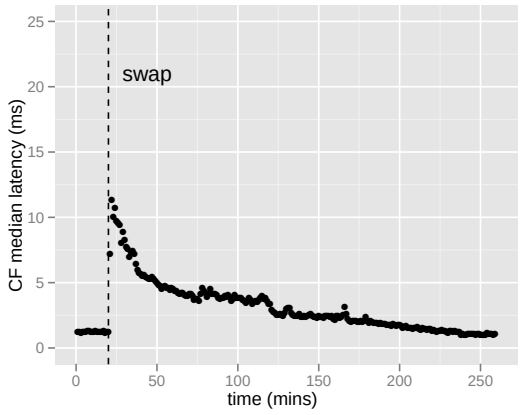
Table 3 shows some statistics for one of LinkedIn's largest clusters. Figure 11 shows the PYMK and CF median client-side read latencies as a function of time since a swap on this cluster (both stores use N=2 and R=1) for one high traffic day. CF has a higher latency than that of PYMK primarily because of the larger value size. We see sub-12 ms latency immediately after a swap with relatively quick stabilization to sub-5 ms latency. This low latency post-swap allows us to push updates to these features multiple times per day.

7 Conclusion and Future Work

In this paper, we present a low-latency bulk loading system capable of serving multiple terabytes of data. By moving the index construction offline to a batch system like Hadoop, our serving layer's performance becomes



(a)



(b)

Figure 11: Median client-side read latency for one of LinkedIn's largest production clusters, as described in Table 3, for the (a) PYMK and (b) CF data sets. The dashed line shows the time when the new data set was swapped in.

more stable and reliable. LinkedIn has been successfully running read-only Voldemort clusters for the past 2 years. It has become an integral part of the product ecosystem with various engineers also using it frequently for quick prototyping of features. The complete system is open-source and freely available.

We plan to add other interesting features to the read-only storage pipeline. Over time we have found that during fetches we exhaust the full bandwidth between data centers running Hadoop (in particular HDFS) and Voldemort. We therefore need improvements to the push process to reduce network usage with minimal impact on build time.

To start with, we are exploring incremental loads. This can be done by generating data file patches on Hadoop by comparing against the previous data snapshot in HDFS and then applying these on the Voldemort side during the fetch phase. We can send the complete index files because (a) they are relatively small files and (b) we can exploit the operating system caching of these files during the fetch phase. This capability has seen few use cases until recently as most of our stores back recommendation features where the delta between data pushes is prohibitively large. Another important feature to save inter-data center bandwidth is the ability to only fetch one replica of the data from HDFS and then propagate it among the Voldemort nodes.

Finally, we are investigating additional index structures that could improve lookup speed and that can easily be built in Hadoop. In particular, cache-oblivious trees, such as van Emde Boas trees [23], require no page size knowledge for optimal cache performance.

Acknowledgements

Bhupesh Bansal and Elias Torres worked on an early version of this project. Thanks to Allen Wittenauer, Anil Alluri, and Bob Liu for their Hadoop support. Several people provided helpful feedback on the paper, including Jun Rao, Bhaskar Ghosh, Igor Perisic, Andrea Dutra, Deepa Elizabeth Jacob, the anonymous FAST reviewers, and our shepherd Kimberly Keeton.

References

[1] Lada Adamic and Bernardo Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[2] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel Bulk Insertion for Large-scale Analytics Applications. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS '10)*, pages 27–31, New York, NY, USA, 2010.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Berkeley, CA, USA, 2006.

[4] Brian Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1:1277–1288, August 2008.

[5] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, New York, NY, USA, 2010.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation – Volume 6 (OSDI '04)*, Berkeley, CA, USA, 2004.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Operating Systems Review*, 41:205–220, October 2007.

[8] M. Dowell and P. Jarratt. The Pegasus method for computing the root of an equation. *BIT Numerical Mathematics*, 12:503–508, 1972.

[9] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[10] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, and Nectarios Koziris. Distributed Indexing of Web Scale Datasets for the Cloud. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud (MDAC '10)*, pages 1:1–1:6, New York, NY, USA, 2010.

[11] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, July 1978.

[12] Gilbert N. Lewis, Nancy J. Boynton, and F. Warren Burton. Expected Complexity of Fast Search with Uniformly Distributed Data. *Inf. Process. Lett.*, 13 (1):4–7, 1981.

[13] Yannis Manolopoulos and G Poulakas. An adaptation of a rootfinding method to searching ordered disk files revisited. *BIT Numerical Mathematics*, 29: 364–368, June 1989.

[14] Peter Mika. Distributed Indexing for Semantic Search. In *Proceedings of the 3rd International Semantic Search Workshop (SEMSEARCH '10)*, pages 3:1–3:4, New York, NY, USA, 2010.

[15] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '99)*, Berkeley, CA, USA, 1999.

[16] Sasha Pachev. *Understanding MySQL Internals*. O'Reilly Media, 2007.

[17] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log log N search. *Communications of the ACM*, 21:550–553, 1978.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. *Mass Storage Systems and Technologies, IEEE*, 0:1–10, 2010.

[19] Adam Silberstein, Brian Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient Bulk Insertion into a Distributed Ordered Table. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 765–778, New York, NY, USA, 2008.

[20] Adam Silberstein, Russell Sears, Wenchao Zhou, and Brian Cooper. A batch of PNUTS: experiences connecting cloud batch and serving systems. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD '11)*, pages 1101–1112, New York, NY, USA, 2011.

[21] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM Computer Communication Review*, 31:149–160, August 2001.

[22] Michael Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[23] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976.