LazyBase: Trading Freshness for Performance in a Scalable Database

James Cipar, Greg Ganger
Carnegie Mellon University

Kimberly Keeton, Charles B. Morrey III
Craig A. N. Soules, Alistair Veitch
HP Labs, Palo Alto

Abstract

The LazyBase scalable database system is specialized for the growing class of data analysis applications that extract knowledge from large, rapidly changing data sets. It provides the scalability of popular NoSQL systems without the query-time complexity associated with their eventual consistency models, offering a clear consistency model and explicit per-query control over the trade-off between latency and result freshness. With an architecture designed around batching and pipelining of updates, LazyBase simultaneously ingests atomic batches of updates at a very high throughput and offers quick read queries to a stale-but-consistent version of the data. Although slightly stale results are sufficient for many analysis queries, fully up-to-date results can be obtained when necessary by also scanning updates still in the pipeline. Compared to the Cassandra NoSQL system, LazyBase provides 4X–5X faster update throughput and 4X faster read query throughput for range queries while remaining competitive for point queries. We demonstrate LazyBase's tradeoff between query latency and result freshness as well as the benefits of its consistency model. We also demonstrate specific cases where Cassandra's consistency model is weaker than LazyBase's.

Categories and Subject Descriptors *H.2.4 Database Management Systems, Parallel Databases, Distributed Databases*

General Terms *Design, Experimentation, Measurement, Performance*

Keywords *Consistency, Freshness, Pipeline*

1. Introduction

Data analytics activities have become major components of enterprise computing. Increasingly, time-critical business decisions are driven by analyses of large data sets that grow and change at high rates, such as purchase transactions, news updates, click streams, hardware monitoring events, tweets and other social media, and so on. They rely on accurate and nearly up-to-date results from sequences of read queries against these data sets, which also need to simultaneously accommodate the high rate of updates.

Unfortunately, current systems fall far short on one or more dimensions. The traditional approach to decision support couples an OLTP system, used for maintaining the primary copy of a database on which update transactions are performed, with a distinct data warehouse system. The latter stores a copy of the data in a format that allows efficient read-only queries, re-populated infrequently (typically daily) from the primary database by a process known as extract, transform, and load (ETL) [19]. For decision support activities that can rely on stale versions of OLTP data, this model is ideal. However, for many modern data analytics activities, which depend upon very high update rates and a greater degree of *freshness* (i.e., up-to-date-ness), it is not.

So-called "NoSQL" database systems, such as Cassandra [1], HBase [2], CouchDB [3] and MongoDB [4], have emerged as an alternate solution. Generally speaking, these systems support arbitrarily high ingest and query rates, scaling effectively by relaxing consistency requirements to eliminate most of the locking and transactional overheads that limit traditional OLTP systems. Most NoSQL systems adopt an *eventual consistency* model, which simplifies the design of the system but complicates its use for correctness-critical data analytics. Programmers of analytics applications often struggle with reasoning about consistency when using such systems, particularly when results depend on data from recent updates. For example, when a client updates a value in the Cassandra database, not all servers receive the new value immediately. Subsequent reads may return either the old value or the new one, depending on which server answers them. Confusingly, a client may see the new value for one read operation, but the old value for a subsequent read if it is serviced by a different server.

LazyBase provides a new point in the solution space, offering a unique blend of properties that matches modern data analytics well. Specifically, it provides scalable high-throughput ingest together with a clear, strong consistency model that allows for a per-read-query tradeoff between latency and result freshness. Exploiting the insight that many queries can be satisfied with slightly out-of-date data, LazyBase batches together seconds' worth of incoming data into sizable atomic transactional units to achieve high throughput updates with strong semantics. LazyBase's batching approach is akin to that of incremental ETL systems, but avoids their freshness delays by using a pipelined architecture that allows different stages of the pipeline to be queried independently. Queries that can use slightly out-of-date data (e.g., a few minutes old) use only the final output of the pipeline, which corresponds to the fully ingested and indexed data. In this case, updates are effectively processed in the background and do not interfere with the foreground query workload, thus resulting in query latencies and throughputs achievable with read-only database systems. Queries that require even fresher results can access data at any stage in the pipeline, at progressively higher processing costs as the freshness increases. This approach provides applications with control over and understanding of the freshness of query results.

LazyBase's architecture also enhances its throughput and scalability. Of course, batching is a well-known approach to improving throughput. In addition, the stages of LazyBase's pipeline can be independently parallelized, permitting flexible allocation of resources (including machines) to ingest stages to accommodate workload variability and overload.

We evaluate LazyBase's performance by comparing to Cassandra both for update performance as well as point and range query performance at various scalability levels. Because of its pipelined architecture and batching of updates, LazyBase maintains 4X–5X higher update throughput than Cassandra at all scalability levels. LazyBase achieves only 45–55% of Cassandra's point query throughput, however, due to the relative efficiency of Cassandra's single row lookups. Conversely, because LazyBase stores data sorted sequentially in the keyspace of the query being performed, LazyBase achieves 4X the range query throughput of Cassandra. LazyBase achieves this performance while maintaining consistent point-in-time views of the data, whereas Cassandra does not.

This paper makes several contributions. Most notably, it describes a novel system (LazyBase) that can provide the scalability of NoSQL database systems while providing strongly consistent query results. LazyBase also demonstrates the ability to explicitly trade off freshness and read-query latency, providing a range of options within which costs are only paid when necessary. It shows how combining batching and pipelining allows for the three key features: scalable ingest, strong consistency, and explicit control of the freshness vs. latency tradeoff.

2. Background and motivation

This section discusses data analytics applications, features desired of a database used to support them, and major shortcomings of the primary current solutions. Related work is discussed in more detail in Section 6.

Data analytics applications. Insights and predictions extracted from large, ever-growing data corpuses have become a crucial aspect of modern computing. Enterprises and service providers often use observational and transactional data to drive various decision support applications, such as sales or advertisement analytics. Generally speaking, the data generation is continuous, requiring the decision support database system to support high update rates. The system must also simultaneously support queries that mine the data to accurately produce the desired insights and predictions. Often, though, query results on a slightly out-of-date version of the data, as long as it is self-consistent, are fine. Table 1 lists various example applications across a number of domains and with varied freshness requirements.

As one example, major retailers now rely heavily on data analytics to enhance sales and inventory efficiency, far beyond traditional nightly report generation [28]. For example, in order to reduce shipping costs, many retailers are shifting to just-in-time inventory delivery at their stores, requiring hour-by-hour inventory information in order to manage transportation of goods [15]. In addition to transaction records from physical point-of-sale systems, modern retailers exploit data like clickstreams, searches, and purchases from their websites to drive additional sales. For example, when a customer accesses a store website, recent activity by the same and other customers can be used to provide on-the-spot discounts, suggestions, advertisements, and assistance.

Social networking is another domain where vast quantities of information are used to enhance user experiences and create revenue opportunities. Most social networking systems rely on graphs of interconnected users that correspond to publish-subscribe communication channels. For example, in Twitter and Facebook, users follow the messages/posts of other users and are notified when new ones are available from the users they follow. Queries of various sorts allow users to examine subsets of interest, such as the most recent messages, the last N messages from a given user, or messages that mention a particular "hashtag" or user's name. A growing number of applications and services also rely on broader analysis of both the graphs themselves and topic popularity within and among the user communities they represent. In addition to targeted advertisements and suggestions of additional social graph connections, social media information can rapidly expose hot-topic current events [10], flu/cold epidemics [22], and even provide an early warning system for earthquakes and other events [5].

Desired properties. Data analytics applications of the types discussed above are demanding and different from more traditional OLTP activities and report generation. Supporting them well requires an interesting system design point with respect to data ingest, consistency, and result freshness.

Application domain	Desired freshness		
	seconds	minutes	hours+
Retail	real-time coupons, targeted ads and suggestions	just-in-time inventory management	product search, trending, earnings reports
Social networking	message list updates, friend/follower list changes	wall posts, photo sharing, news updates and trending	social graph analytics
Transportation	emergency response, air traffic control	real-time traffic maps, bus/plane arrival prediction	traffic engineering, bus route planning
Investment	real-time micro-trades, stock tickers	web-delivered graphs	trend analyses, growth reports
Enterprise information management	infected machine identification	email, file-based policy violations	enterprise search results, e-discovery requests
Data center and network monitoring	automated problem detection and diagnosis	human-driven diagnosis, online performance charting	capacity planning, availability analyses, workload characterization

Table 1. Freshness requirements for application families from a variety of domains.

Data ingest: Such applications rely on large data corpuses that are updated rapidly, such as clickstreams, Twitter tweets, location updates, or sales records. Any system used to support such analytics must be able to ingest and organize (e.g., index) the data fast enough to keep up with the data updates. At the same time it sustains these high data ingestion rates, the system must also be able to answer the read-only queries it receives. Fortunately though, high-throughput ingest analytics data tends to be observational or declarative data, such that the updates and the queries are independent. That is, the updates add, replace or delete data but do not require read-modify-write transactions on the data corpus. So, a decision support system can handle updates separately from read-only analytic queries. As discussed further below, it is important that the system support atomic updates to a set of related data items, so that consistent query results can be obtained.

Consistency: To simplify the programming of decision support applications, query results typically must have a consistency model that analysts and application writers can understand and reason about. Weak forms of consistency, such as eventual consistency, where the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value [42], are difficult to reason about. As a result, application developers generally seek stronger properties [40, 42], such as a consistent prefix, monotonic reads, "read my writes," causal consistency, or self-consistency. By requesting a consistent prefix, a reader will observe an ordered sequence of writes starting with the first write to a data object. With monotonic read consistency, if a reader has seen a particular value for an object, any subsequent accesses in that session will never return any previous values; as such, it is often called a "session guarantee." The read my writes property guarantees that the effects of all writes performed by a client are visible to the client's subsequent reads. Causal consistency is a generalization of read my writes, where any process with a causal relationship to a process that has updated an object will see the updated value. Self-consistency refers to the property that the data set has been updated in its entirety, in the face of multi-row (or even multi-table) update transactions. Without such stronger consistency properties, application writers struggle to produce accurate insights and predictions. For example, for just-in-time inventory management, not having a consistent view of the data can lead to over- or under-estimating delivery needs, either wasting effort or missing sales opportunities due to product unavailability. As another example, a user notified of a new tweet could, upon trying to retrieve it, be told that it does not exist. Even disaster recovery for such systems becomes more difficult without the ability to access self-consistent states of the data; simple solutions, such as regular backups, require consistent point-in-time snapshots of the data.

Freshness: For clarity, we decouple the concepts of data consistency (discussed above) and data *freshness*. Freshness (also known as bounded staleness [40]) describes the delay between when updates are ingested into the system, and when they are available for query – the "eventual" of eventual consistency. Of course, completely-up-to-date freshness would be ideal, but the scalability and performance costs of such freshness has led almost all analytics applications to accept less. For most modern analytics, bounded staleness (e.g., within seconds or minutes) is good enough. We believe that it would be best for freshness (or the lack thereof) to be explicit and, even better, application-controlled. Application programmers should be able to specify freshness goals in an easy-to-reason-about manner that puts a bound on how out-of-date results are (e.g., "all results as of 5 minutes ago"), with a system supplying results that meet this bound but not doing extra work to be more fresh. The system may provide even fresher results, but only if it can do so without degrading performance. Such an approach matches well with the differences in freshness needs among applications from the various domains listed in Table 1.

In the next section, we describe LazyBase, which provides a new point in the solution space to satisfy the needs of modern data analytics applications. In particular, it combines scalable and high-throughput data ingest, a clear consistency model, and explicit per-read-query control over the tradeoff between latency and result freshness.

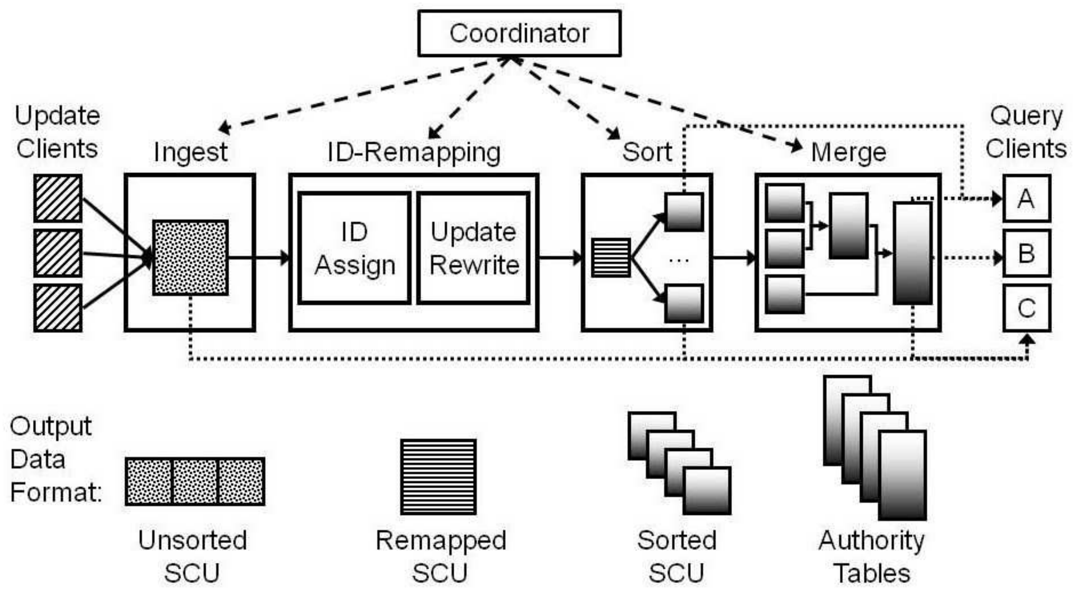

Figure 1. LazyBase pipeline.

3. Design and implementation

LazyBase is a distributed database that focuses on high-throughput updates and high query rates using batching. Unlike most batching systems, LazyBase can dynamically trade data freshness for query latency at query time. It achieves this goal using a pipelined architecture that provides access to update batches at various points throughout their processing. By explicitly accessing internal results from the pipeline stages, applications can trade some query performance to achieve the specific data freshness they require.

This section outlines LazyBase's design and implementation, covering the application service model, how data is organized in the system, its pipelined architecture, work scheduling, scaling, fault tolerance, query model, and on-disk data format.

3.1 Service model

LazyBase provides a *batched update/read query* service model, which decouples update processing from read-only queries. Unlike an OLTP database, LazyBase cannot simultaneously read and write data values in a single operation. Queries can only read data, while updates (e.g., adds, modifies, deletes) are *observational*, meaning that a new/updated value must always be given; this value will overwrite (or delete) existing data, and cannot be based on any data currently stored. For example, there is no way to specify that a value should be incremented, or be assigned the results of subtracting one current value from another, as might happen in a conventional database. This restriction is due to the complexity of maintaining read my writes consistency in a distributed batch-processing system. Methods for providing this functionality are an area of future work, although we have not found it to be a limitation in any of our applications to date.

Clients upload a set of updates that are batched into a single *self-consistent update* or SCU. An SCU is the granularity of work performed at each pipeline stage, and LazyBase's ACID semantics are on the granularity of an SCU. The set of changes contained within an SCU is applied *atomically* to the tables stored in the database, using the mechanisms described in Section 3.2. An SCU is applied *consistently*, in that all underlying tables in the system must be consistent after the updates are applied. SCUs are stored *durably* on disk when they are first uploaded, and the application of an SCU is *isolated* from other SCUs. In particular, LazyBase provides *snapshot isolation*, where all reads made in a query will see a consistent snapshot of the database; in practice, this is the last SCU that was applied at the time the query started. This design means that LazyBase provides readers self-consistency, monotonic reads, and a consistent prefix with bounded staleness.

Updates are specified as a set of Thrift [6] RPC calls, to provide the data values for each row update. For efficiency, queries are specified programmatically. Like MapReduce [21], each query maps to a restricted dataflow pattern, which includes five phases: filter, uniquify, group_by, aggregate, and post_filter. These phases and their implementation are described in more detail in Section 3.7.

3.2 Data model

Similar to conventional RDBMS/SQL systems, LazyBase organizes data into tables with an arbitrary number of named and typed columns. Each table is stored using a *primary view* that contains all of the data columns and is sorted on a *primary key*: an ordered subset of the columns in the table.

For example, a table might contain three columns ⟨A, B, C⟩ and its primary view key could be ⟨A, B⟩, meaning it's sorted first by A and then by B for equal values of A. The remaining columns are referred to as the *data columns*. Tables may also have any number of materialized *secondary views* which contain a subset of the columns in the table and are sorted on a different *secondary key*. The secondary key columns must be a superset of the primary key columns to enforce uniqueness, but can specify any column order for sorting. LazyBase has no concept of non-materialized views.

Because these tables describe both *authority* data that has been fully processed by the pipeline as well as *update* data that is in-flight, they also contain additional hidden fields. Each row is assigned a timestamp indicating the time at which it should be applied as well as a delete marker indicating if the specified primary key should be removed (as opposed to inserted or modified). Each data column is also assigned an additional timestamp indicating at what time that column was last updated. These timestamps can vary in the case of partial row updates.

Like many other databases, LazyBase supports the concept of an auto-increment column (also known as a database surrogate key), by which a given key (potentially multi-column) can be assigned a single unique incrementing value in the system, allowing users of the system to improve query and join times and reducing storage requirements for large keys. For instance, two strings specifying the hostname and path of a file could be remapped to a single integer value, which is then used in other tables to store observational data about that file. We refer to these auto-increment columns as *ID-key* columns.

3.3 Pipelined design

The design of the LazyBase pipeline is motivated by the goal of ingesting and applying updates as efficiently as possible while allowing queries to access these intermediate stages if needed to achieve their freshness goals. Figure 1 illustrates the pipeline stages of LazyBase: ingest, id-remapping, sort, and merge. In addition to these stages, a coordinator is responsible for tracking and scheduling work in the system.

The *ingest* stage batches updates and makes them durable. Updates are read from clients and written into the current SCU as rows into an unsorted primary view for the appropriate table type. Rows are assigned timestamps based on their ingestion time. ID-keys in the updates are assigned temporary IDs, local to the SCU, and the mapping from key to temporary ID is stored in the SCU (allowing queries to use this mapping if needed). LazyBase marks an SCU as complete once either sufficient time has passed (based on a timeout) or sufficient data has been collected (based on a watermark). At this point, new clients' updates are directed to the next SCU and any remaining previously connected clients complete their updates to the original SCU. Once complete, the ingest stage notifies the coordinator, which assigns it a globally unique SCU number and schedules the SCU to be processed by the rest of the pipeline.

The *id-remapping* stage converts SCUs from using their internal temporary IDs to using the global IDs common across the system. Internally, the stage operates in two phases, id-assignment and update-rewrite, that are also pipelined and distributed. In id-assignment, LazyBase does a bulk lookup on the keys in the SCU to identify existing keys and then assigns new global IDs to any unknown keys, generating a temporary ID:global ID mapping for this update. Because id-assignment does a lookup on a global key-space, it can only be parallelized through the use of *key-space partitioning*, as discussed below. In update-rewrite, LazyBase rewrites the SCU's tables with the correct global IDs and drops the temporary mappings.

The *sort* stage sorts each of the SCU's tables for each of its views based on the view's key. Sorting operates by reading the table data to be sorted into memory and then looping through each view for that table, sorting the data by the view's key. The resulting sorted data sets form the sorted SCU. If the available memory is smaller than the size of the table, sorting will break a table into multiple memory-sized chunks and sort each chunk into a separate output file to be merged in the merge stage. While writing out the sorted data, LazyBase also creates an index of the data that can be used when querying.

The *merge* stage combines multiple sorted SCUs into a single sorted SCU. By merging SCUs together, we reduce the query cost of retrieving fresher results by reducing the number of SCUs that must be examined by the query. LazyBase utilizes a tree-based merging based on the SCU's global number. SCUs are placed into a tree as leaf nodes and once a sufficient span of SCUs is available (or sufficient time has passed), they are merged together. This merging applies the most recent updates to a given row based on its data column timestamps, resulting in a single row for each primary key in the table. Eventually, all of the updates in the system are merged together into a single SCU, referred to as the *authority*. The authority is the minimal amount of data that must be queried to retrieve a result from LazyBase. Just as in the sort stage, LazyBase creates an index of the merged data while it is being written that can be used when querying.

When each stage completes processing, it sends a message to the *coordinator*. The coordinator tracks which nodes in the system hold which SCUs and what stages of processing they have completed, allowing it to schedule each SCU to be processed by the next stage. The coordinator also tracks how long an SCU has existed within the system, allowing it to determine which SCUs must be queried to achieve a desired level of freshness. Finally, the coordinator is responsible for tracking the liveness of nodes in the system and initiating recovery in the case of a failure, as described in Section 3.6.

3.4 Scheduling

LazyBase's centralized coordinator is responsible for scheduling all work in the system. Nodes, also called *workers*, are part of a pool, and the coordinator dynamically schedules tasks on the next available worker. In this manner, a worker may perform tasks for whatever stage is required by the workload. For example, if there is a sudden burst of updates, workers can perform ingest tasks, followed by sorting tasks, followed by merging tasks, based on the SCU's position in the pipeline. Currently workers are assigned tasks without regard for preserving data locality; such optimizations are the subject of future work.

3.5 Scaling

Each of the pipeline stages exhibit different scaling properties, as described above. Ingest, sort, and the update-rewrite sub-phase of id-remapping maintain no global state, and can each be parallelized across any number of individual SCUs.[1] Merge is log-n parallelizable, where n is the fan-out of the merge tree. With many SCUs available, the separate merges can be parallelized; however, as merges work their way toward the authority, eventually only a single merge to the authority can occur. Sort and merge can also be parallelized across the different table types, with different tables being sorted and merged in parallel on separate nodes. Finally, all of the stages could be parallelized through key-space partitioning, in which the primary keys for tables are hashed across a set of nodes with each node handling the updates for that set of keys. This mechanism is commonly employed by many "cloud" systems [1].

Automatically tuning the system parameters and run-time configuration to the available hardware and existing workload is an area of ongoing research. Currently, LazyBase implements the SCU-based parallelism inherent in ingest, sort, update-rewrite, and merge.

3.6 Fault tolerance

Rather than explicitly replicating data at each stage, LazyBase uses its pipelined design to survive node failures: if a processing node fails, it recreates the data on that node by reprocessing the data from an earlier stage in the pipeline. For example, if a merge node fails, losing data for a processed SCU, the coordinator can re-schedule the SCU to be processed through the pipeline again from any earlier stage that still contains the SCU. The obvious exception is the ingest stage, where data must be replicated to ensure availability.

Similarly, if a node storing a particular SCU representation fails, queries must deal with the fact that the data they desire is unavailable, by either looking at an earlier representation (albeit at slower performance) or waiting for the desired representation to be recomputed (perhaps on another, available node). Once SCUs have reached the authority, it may be replicated both for availability and query load balancing. LazyBase can then garbage collect older representations from the previous stages.

LazyBase detects worker and coordinator failures using a standard heartbeat mechanism. If the coordinator fails, it is restarted and then requests the current state of the pipeline by retrieving SCU information from all live workers. If a worker fails, it is restarted and then performs a local integrity check followed by reintegration with the coordinator. The coordinator can determine what SCU data on the worker is still relevant based on pipeline progress at reintegration time. If a worker is unavailable for an extended period of time, the coordinator may choose to re-process SCU data held on that worker from an earlier stage to keep the pipeline busy.

3.7 Queries

Queries can operate on the SCUs produced by any of LazyBase's stages. In the common case, queries that have best-effort freshness requirements will request results only from the authority SCU. To improve freshness, queries can contact the coordinator, requesting the set of SCUs that must be queried to achieve a given freshness. They then retrieve each SCU depending on the stage at which it has been processed and join the results from each SCU based on the result timestamps to form the final query results. For sorted or merged SCUs, the query uses the index of the appropriate table to do the appropriate lookup. For unsorted SCUs, the query does a scan of the table data to find all of the associated rows. If joins against ID-key columns are required, the unsorted data's internal temporary ID-key mappings must also be consulted.

Queries follow a five-phase dataflow: filter, uniquify, group_by, aggregate and post_filter. Filtering is performed in parallel on the nodes that contain the SCU data being queried, to eliminate rows that do not match the query. For example, a query that requires data from two merged SCUs and a sorted SCU will run the queries against those SCUs in parallel (in this case three-way parallelization). The results are collected and joined by the caller in the uniquify phase. The goal of this phase is to eliminate the duplicates that may exist because multiple SCUs can contain updates to a single row. This phase effectively applies the updates to the row. The group_by and aggregate phases gather data that belongs in matching groups, and compute aggregate functions, such as `sum`, `count`, and `max`, over the data. To access row data directly, we also include a trivial `first` aggregate that simply returns the first value it sees from each group. Finally, the post_filter phase allows the query to filter rows by the results of the aggregates.

3.8 Data storage

Our implementation makes heavy use of DataSeries [12], an open-source compressed table-based storage layer designed for streaming throughput. DataSeries stores sets of

[1] Because the mapping from temporary ID to global ID is unique to the SCU being converted, any number of update-rewrites can be performed in parallel on separate SCUs.

rows into *extents*, which can be dynamically sized and are individually compressed. Extents are the basic unit of access, caching, and indexing within LazyBase. LazyBase currently uses a 64KB extent size for all files. DataSeries files are self-describing, in that they contain one or more XML-specified extent types that describe the schema of the extents. Different extent types are used to implement different table views. DataSeries provides an internal index in each file that can return extents of a particular type.

LazyBase stores an SCU in DataSeries files differently for each stage. In the ingest stage an SCU is stored in a single DataSeries file with each table's primary view schema being used to hold the unsorted data for each table. In the id-remapping stage, the ID-key mappings are written out as sorted DataSeries files, two for each ID-key mapping (one in id order, the other in key order). In the sort and merge stages, the SCU is stored as a separate sorted DataSeries file for each view in each table.

LazyBase also uses external extent-based indexes for all of its sorted files. These index files store the minimum and maximum values of the view's sort key for each extent in a file. Unlike a traditional B+tree index, this *extent-based* index is very small (two keys and an offset for each extent), and with 64KB extents can index a 4 TB table with a 64-bit key in as little as 250 MB, which can easily fit into the main memory of modern machine. Although the range-based nature of the index may result in false positives, reducing any lookup to a single disk access still dramatically improves query performance for very large tables.

In addition to its focus on improved disk performance, DataSeries uses type-specific code to improve its performance over many other table-based storage layers. LazyBase automatically generates code for ingestion, sorting, merging, and basic querying of indexes from XML-based schema definitions.

4. Evaluation

The goal of LazyBase is to provide a high-throughput, scalable update system that can trade between query freshness and query latency while providing an understandable consistency model. We evaluate LazyBase's pipeline and query performance against Cassandra [1], a popular scalable NoSQL database that is considered to be "write-optimized" in its design [20]. In addition, we demonstrate the effect of LazyBase's batching on update performance, and LazyBase's unique freshness-queries, demonstrating the user-tunable trade-off between latency and freshness. Finally, we compare the consistency models of LazyBase and Cassandra.

4.1 Experimental setup

All nodes in our experiments were Linux Kernel-based Virtual Machines (KVM's) with 6 CPU cores, 12 GB of RAM, and 300 GB of local disk allocated through the Linux logical volume manager (LVM), running Linux 2.6.32 as the guest OS. Each KVM was run on a separate physical host, with 2-way SMP quad-core 2.66 GHz Intel Xeon E5430 processors, 16GB of RAM and an internal 7200 RPM 1 TB Seagate SATAII Enterprise disk drive. A second internal 400 GB disk stored the OS, 64-bit Debian "squeeze." Nodes were connected via switched Gigabit Ethernet using a Force10 S50 switch. These physical nodes were part of the OpenCirrus cloud computing research testbed [17].

Unless otherwise stated, experiments were run using 10 database nodes and 20 upload nodes. In LazyBase, the 10 database nodes were split between a single node running the id-assignment sub-stage and the coordinator and the other nine worker nodes running an ingest stage as well as a second dynamically assigned stage (e.g., id-update-rewrite, sort or merge). Cassandra was configured with nine nodes, equivalent to the nine workers in LazyBase. We use a default unsorted SCU size of 1.95M rows and in the merge stage we merge at most eight SCUs together at a time.

Our experimental data set is a set of 38.4 million tweets collected from Twitter's streaming API between January and February 2010 as part of a study on spam detection in Twitter [25]. Each observation contains the tweet's ID, text, source, and creation time, as well as information about the user (e.g., id, name, description, location, follower count, and friends count). The table's primary view is sorted on tweet id. The total data set is 50GB uncompressed, the average row size (a single tweet) is slightly over 1KB. While the Twitter data is a realistic example of an application that requires high-throughput update, it uses a simple schema with only one sort order and no ID-keys. In addition to the Twitter data we used an artificial data generator that can create arbitrarily large data sets that exercise the indexing and sorting mechanisms of LazyBase. Section 4.5 describes this data set in more detail.

4.2 Update

LazyBase strives for a high-performance pipeline in two respects: efficiency and scalability. This section evaluates LazyBase's choices for batching to achieve high efficiency and explores LazyBase's scalability and how it compares with the scalability of Cassandra.

4.2.1 Efficiency

LazyBase batches together updates into sizable SCUs to achieve high-throughput ingest with stronger semantics than eventually consistent systems. Figure 2 illustrates the sensitivity of LazyBase's update throughput to SCU size. As expected, as the SCUs get larger, update throughput increases. However, the per-pipeline stage latency also increases, leading to a greater freshness spectrum for LazyBase queries. To maximize throughput, we choose a default SCU size of 1.95M rows for use in subsequent experiments.

Table 2 lists the average latency and throughput of each pipeline stage for our workload. Because the merge stage

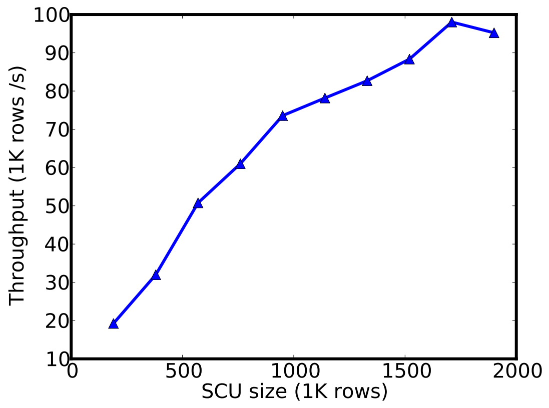

Figure 2. Inserts per second for different SCU sizes

Stage	Latency (s)	Rows/s
Ingest	49.7	39,000
ID Remap	5.5	327,000
Sort	12.0	158,000
Merge	31.0+	120,000

Table 2. Performance of individual pipeline stages. Note that this workload does not contain ID-keys, making the ID Remap phase very fast.

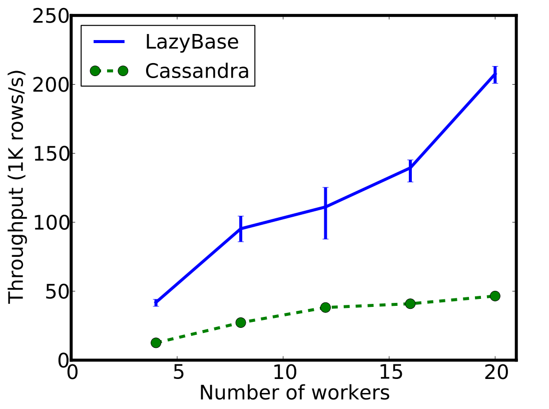

Figure 3. Inserts processed per second. Error bars represent the min and max of three runs, while the line plots the mean.

latency is highly dependent upon the total size of the SCUs being merged, the reported latency is for the smallest 8-SCU merge. We observe that stages run at varying speeds, indicating that the ability to parallelize individual stages is important to prevent bottlenecks and improve overall system throughput.

4.2.2 Update scalability

To demonstrate LazyBase's update scalability, we compared equally sized LazyBase and Cassandra clusters. Figure 3 illustrates this comparison from four worker nodes up through 20 worker nodes. In the LazyBase configuration, one additional node ran the id-assignment and coordinator stages and the remaining nodes ran one ingest worker and one dynamically assigned worker (e.g., id-update-rewrite, sort or merge). In the Cassandra configuration, each node ran one Cassandra process using a write-one policy (no replication). For each cluster size, we measured the time to ingest the entire Twitter data set into each system.

We observe that both systems scale with the number of workers, but LazyBase outperforms Cassandra by a factor of 4X to 5X, due to the architectural differences between the systems. Updates in Cassandra are hashed by key to separate nodes. Each Cassandra node utilizes a combination of a commit log for durability and an indexed in-memory structure in which it keeps updates until memory pressure flushes them to disk. Cassandra also performs background compaction of on-disk data similar to LazyBase's merge stage. Unlike LazyBase, Cassandra shows little improvement with increased batch sizes, and very large batches cause errors during the upload process. Conversely, LazyBase dedicates all of the resources of a node to processing a large batch of updates. In turn, this reduction in contention allows LazyBase to take better advantage of system resources, improving its performance. LazyBase's use of compression improves disk throughput, while its use of schemas improves individual stage processing performance, giving it additional performance advantages over Cassandra.

4.3 Query

We evaluate LazyBase's query performance on two metrics. First, we compare it to Cassandra for both point and range queries, showing query throughput for increasing numbers of query clients. Second, we demonstrate LazyBase's unique ability to provide user-specified query freshness, showing the effects of query freshness on query latency.

To provide an equitable query comparison to Cassandra, we added a distribution step at the end of LazyBase's pipeline to stripe authority data across all of the worker nodes in batches of 64K rows. Because this distribution is not required for low-latency authority queries, we did not include its overhead in the ingestion results; for our workload the worst-case measured cost of this distribution for the final authority is 314 seconds (a rate of 120K rows/s). We further discuss how such striping could be tightly integrated into LazyBase's design in Section 5.

To support range queries in Cassandra, we added a set of *index rows* that store contiguous ranges of keys. Similar to the striped authority file in LazyBase, each index row is keyed by the high-order 48 bits of the 64 bit primary table key (i.e., each index row represents a 64K range of keys). We add a column to the index row for every key that is stored in that range, allowing Cassandra to quickly find the keys of all rows within a range using a small number of point queries. In our tests, the range queries only retrieve the

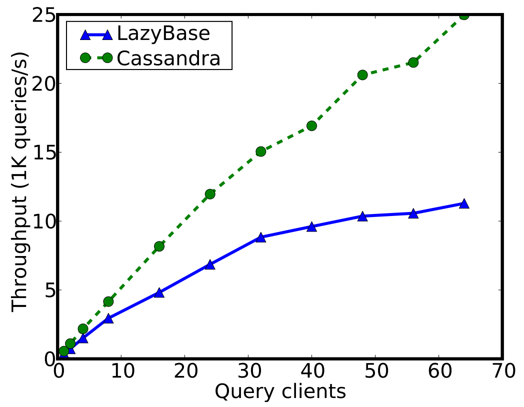

Figure 4. Random single row queries for LazyBase and Cassandra. Measured as queries per second with increasing numbers of query clients running 1000 queries per client.

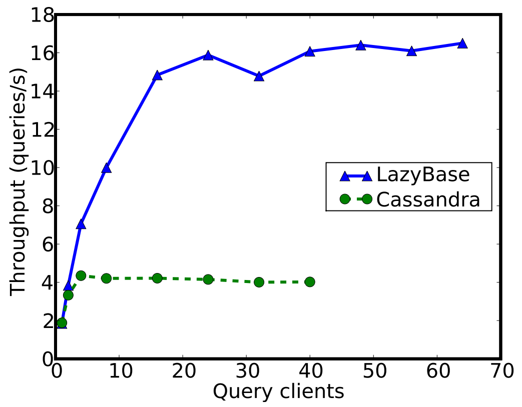

Figure 5. Random range queries for LazyBase and Cassandra with 0.1% selectivity. Measured as queries per second with increasing numbers of query clients running 10 queries per client.

existing keys in the range, and no data columns, allowing Cassandra to answer the query by reading only the index rows. Having Cassandra also perform the point lookups to retrieve the data columns made individual 0.1% selectivity queries in Cassandra take over 30 minutes.

We also ran all experiments on a workload that would fit into the memory of the Cassandra nodes, since Cassandra's performance out-of-core was more than 10 times slower than LazyBase. We believe this to be due to Cassandra's background compaction approach, in which all uncompacted tables must be examined in reverse-time order when querying until the key is found. To mitigate this effect, we issued each query twice: once to warm up the cache, and a second time to measure the performance.

Figure 4 illustrates single-row query performance of both LazyBase and Cassandra. We exercise the query throughput of the two systems using increasing numbers of query clients, each issuing 1000 random point queries. We see that Cassandra's throughput is approximately twice that of LazyBase, due primarily to the underlying design of the two systems. Cassandra distributes rows across nodes using consistent hashing and maintains an in-memory hash table to provide extremely fast individual row lookups. LazyBase keeps a small in-memory index of extents, but must decompress and scan a full extent in order to retrieve an individual row. The result is that LazyBase has a high query latency, which results in slower absolute throughput for the fixed workload.

Figure 5 illustrates range query performance of both LazyBase and Cassandra. These queries simply return the set of valid tweet IDs within the range, allowing Cassandra to service the query from the index rows without using a set query to access the data belonging to those tweets. We exercise the query throughput of the two systems using increasing numbers of query clients, each issuing 10 random queries retrieving a range over 0.1% of the key-space. With a low number of query clients, LazyBase and Cassandra have

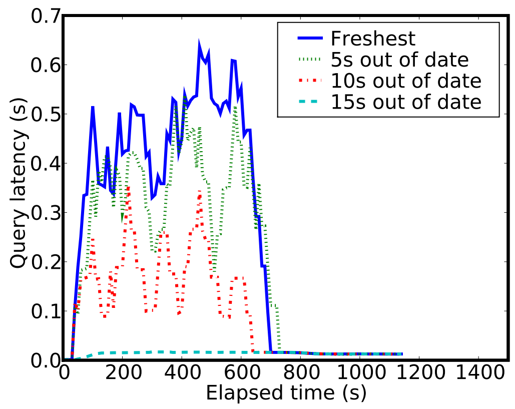

Figure 6. Query latency over time for steady-state workload. Sampled every 15s, and smoothed with a sliding window average over the past minute.

similar performance; however, LazyBase continues to scale up to 24 clients, while Cassandra tops out at four. LazyBase's sorted on-disk data format allows it to serve range queries as streaming disk I/O, while Cassandra must read many tables to retrieve a range.

Figure 6 illustrates the query latency when run with concurrent updates in LazyBase for four different point query freshnesses, 15 seconds, 10 seconds, 5 seconds, and 0 seconds. In this experiment, we ran an ingest workload of 59K inserts per second for a period of 650 seconds. Pipeline processing continued until 1133 seconds at which point the merged SCUs were fully up-to-date.

Figure 7 illustrates a 200-second window of this ingest workload, measuring the effective staleness of the sorted SCUs in the system. When a sort process completes, the staleness drops to zero, however, as the next sort process

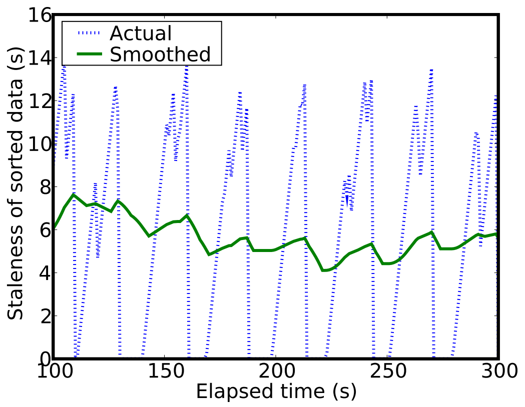

Figure 7. The staleness of the sorted data over a 200-second window of the steady-state workload. That is, at any point in time, the freshness limit that would have to be set to avoid expensive queries to unsorted data. The smoothed line is over a 60-second sliding window.

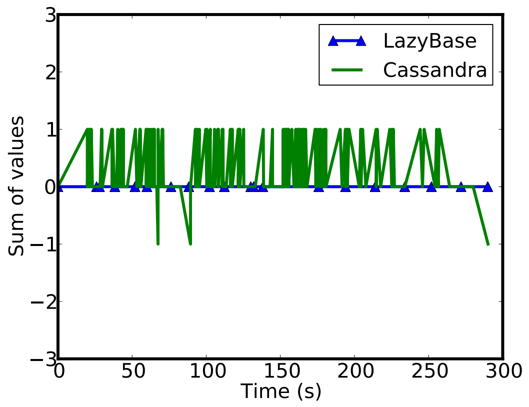

Figure 8. The sum of the values of two rows updated within a single client session. Non-zero values indicate an inconsistency in the view of the two rows.

continues, the staleness continues to increase until the next SCU is available at that stage. The result is that although the average staleness is around 6 seconds, the instantaneous staleness varies between 0 and 14 seconds.

Taken together, Figures 6 and 7 effectively demonstrate the cost of query freshness in a live system. In the case of the 15-second query, we see that query latency remained stable throughout the run. Because newly ingested data was always processed through the sort stage before the 15 second deadline, the 15-second query always ran against sorted SCUs, with its latency increasing only slightly as the index size for these files increased. In the case of a 0-second query, the query results occasionally fall into the window where sorted SCUs are completely fresh, but usually included unsorted SCUs that required linear scans, increasing query latency. Because the results in Figure 6 are smoothed over a 60-second window, points where only sorted data was examined show up as dips rather than dropping to the lowest latency. The 10-second query was able to satisfy its freshness constraint using only sorted SCUs much more frequently, causing its instantaneous query latency to occasionally dip to that of the 15-second query. As a result, its average query latency falls between the freshest and least fresh queries. At 650 seconds ingest stops and shortly thereafter all SCUs have been sorted. This results in all query freshnesses achieving the same latency past 662 seconds.

4.4 Consistency

To compare the consistency properties of LazyBase and Cassandra, we performed an experiment involving a single-integer-column table with two rows. Our LazyBase client connected to the database, issued two row updates incrementing the value of the first row and decrementing the value of the second row, and then issued a commit. Our Cassandra client performed the same updates as part of a “batch update” call and used the `quorum` consistency model, in which updates are sent to a majority of nodes before returning from an update call. Conceptually, from a programmer’s perspective, this would be equivalent to doing a transactional update of two rows, to try to ensure that the sum of the two rows’ values is zero. We also maintained a background workload to keep the databases moderately busy, as they might be in a production system. During the experiment we ran a query client that continuously queried the two rows of the table and recorded their values. In the case of Cassandra, our query client again used the `quorum` consistency model, in which queries do not return until they receive results from a majority of the nodes.

Figure 8 graphs the sum of the values of the two rows over the lifetime of the experiment. In the case of LazyBase, we see that the sum is always zero, illustrating its consistency model: all updates within a single SCU are applied together atomically. In the case of Cassandra, we see that the sum of the values varies between -1, 0, and 1, illustrating that Cassandra does not provide self-consistency to its users.

Figure 9 further illustrates the difficulty placed on users of systems like Cassandra, showing the effective timestamps of the retrieved rows for each query over a 100-second period of the experiment.[2] In the case of LazyBase, we see that its batch consistency model results in a step function, in which new values are seen once a batch of updates has been processed. Under this model, the two rows always remain consistent. In the case of Cassandra, not only do the two rows differ in timestamp, but often the returned result of a given value is older than the previously returned result, illustrated by a dip in the effective time for that row. This

[2] Note that we show both rows for Cassandra, but only a single row for LazyBase, as LazyBase’s consistency model ensures that both rows always have the same timestamp.

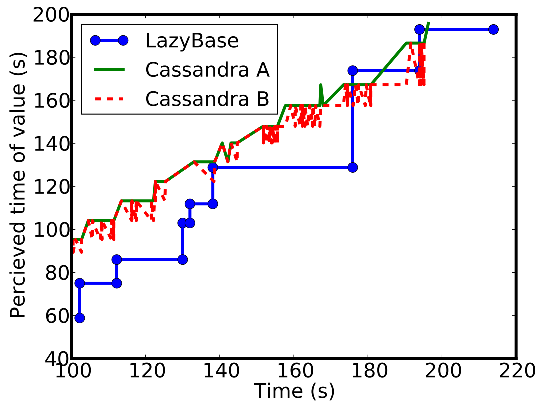

Figure 9. The effective timestamp of the returned rows at query time. Differences between Cassandra row A and Cassandra row B indicate inconsistencies between the rows at query time. Dips in the timestamps of a single row indicate violations in monotonic read consistency.

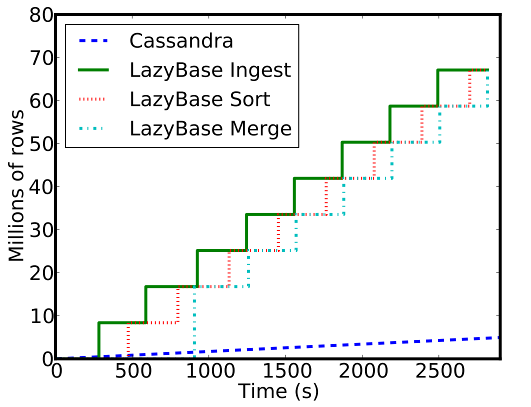

Figure 10. Ingest performance for the complex schema workload. The x-axis represents the elapsed time, and the y-axis represents the number of rows that are available in the database. For LazyBase each line represents the number of rows available at a particular stage of the pipeline.

violation of monotonic read consistency adds a second layer of complexity for users of the system.

We also see that LazyBase generally provides less fresh results than Cassandra, as is expected in a batch processing system. However, because LazyBase is able to process updates more effectively, it less frequently goes into overload. Thus, in some cases of higher load, it is actually able to provide fresher results than Cassandra, as demonstrated in the period from 180 to 185 seconds in the experiment.

4.5 Complex schema

Although the Twitter data set used in the previous examples represents an interesting real-world application, it has a fairly simple schema and indexing structure. We also evaluate the performance of LazyBase with a more complex artificial data set. This data set consists of three tables, `TestID`, `TestOne` and `TestTwo`. `TestID` contains a 64-bit integer ID-key, which is used in the primary key for `TestOne` and `TestTwo`. For each ID-key, `TestOne` contains a string, `stringData`, and a 32-bit integer `intData`. `TestTwo` contains a one-to-many mapping from each ID-key to attribute-value pairs, both 32-bit integers. In addition to the primary view, each table also has a secondary view, sorted on a secondary key. `TestOne`'s secondary view is sorted by the `stringData` field, while `TestTwo`'s is sorted by the `value` field.

When uploading this data set to Cassandra, both primary and secondary views are stored using the technique described above for range queries: a separate "index table" stores one row for each value (e.g., the sort key for the view) and a column for every primary key that has that value. For example, the secondary index table for `TestTwo` contains one row for each value and a column for every primary key that has that value in an attribute.

Stage	Latency (s)	Rows/s
Ingest	311	27,000
ID Remap	61	138,000
Sort	206	41,000
Merge	115	73,000

Table 3. Performance of individual pipeline stages on the complex workload.

Figure 10 shows the performance of ingesting the complex data set to a cluster of nine servers for both LazyBase and Cassandra. For LazyBase the graph shows the number of rows available at each stage of the pipeline. The batch size for Cassandra was set to 1K rows, while the batch size for LazyBase was set to 8M rows. Cassandra's batch size is limited because batching must be done by the client, and each batch must be transmitted in a single RPC. Even within the hard limits imposed by Cassandra, larger batch sizes produce performance problems: we found that Cassandra's write performance for batches of 10K rows was worse than with smaller batches. LazyBase, which is designed with batching in mind, performs batching on the ingest server and stores batches on disk, allowing them to be arbitrarily large. Because of the large batch sizes, the number of rows available at any stage of LazyBase is a step function. The same is true for Cassandra, but because the batch sizes are much smaller, the steps are not visible in the graph.

We also computed the average per-stage latency, similar to Table 2. These results are shown in Table 3. Compared to Table 2, all stages of the pipeline are slowed down in this test, due to two factors: the complexity of the schema and the lack of parallelism in ingest. Compared to the first schema, which required no remapping and produced only a single sort order, this schema must remap the two tables' primary keys, and produce and merge two sort orders for each table. Furthermore, the previous experiment was able to take advantage of parallelism during the ingest stage: multiple data streams were uploaded simultaneously and merged into a single SCU, increasing the throughput of the ingest stage. In this experiment a single client is performing all uploads, so there is no ingest stage parallelism.

4.6 Summary

The results of our evaluation of LazyBase illustrate its many benefits. By trading off query freshness for batching, LazyBase's pipelined architecture can process updates at rates 4X to 5X faster than the popular Cassandra NoSQL database. Although Cassandra's in-memory hash-table has twice the point query throughput as LazyBase, LazyBase's sorted on-disk format gives 4X the range query throughput. When performing out-of-core queries, LazyBase exceeds Cassandra's query latency by more than 10 times. We also illustrate how users of LazyBase, unlike other systems, can dynamically choose to trade query latency for query freshness at query time.

LazyBase exhibits these properties while also providing a clear consistency model that we believe fits a broader range of applications. Our evaluation of Cassandra's consistency illustrated violations of both self-consistency, where we observed inconsistencies in multi-row queries, and monotonic read consistency, where we observed clients seeing an older version of a row up to 5 seconds after seeing the newest version of that row. These types of inconsistencies are not possible in LazyBase by design, and our experimental results showed that they do no occur in practice.

5. Discussion

We have developed LazyBase to perform well under a particular set of use cases. In this section, we describe modifications to LazyBase's design that could broaden its applicability or make it more suitable to a different set of use cases.

Data distribution: LazyBase's pipelined design offers several trade-offs in how data is distributed amongst nodes for the purposes of query scalability. Our experiments were run using an authority file striping approach; however, it may be advantageous to perform this striping during the sort stage of the pipeline to provide the same kind of query scalability to freshness queries. This approach would require a more robust striping strategy, perhaps using consistent hashing of key ranges to ensure even distribution of work. Furthermore, the choice of stripe size has an effect on both range query performance and workload distribution.

Alternate freshness models: LazyBase's freshness model currently describes requirements by putting a bound on how out-of-date results are (e.g., "all results as of an hour ago"). Other freshness models may also be desirable, such as only the most recent updates (e.g., "all results that were received in the last hour") or a past point in time (e.g., "all results as of three weeks ago"). Given LazyBase's pipelined design, one could easily add analysis stages after ingest to provide stream query analysis of incoming updates. Because LazyBase doesn't overwrite data in-place, by not deleting data, LazyBase could generate consistent point-in-time snapshots of the system that could be tracked by the coordinator to provide historical queries.

Scheduling for different freshnesses: LazyBase's scheduler provides the same priority for processing of all tables in the system. However, some applications may desire that queries to particular tables (e.g., a table of security settings) are always fresher. To better support those applications, LazyBase could adjust its scheduling to prioritize work based on desired freshness for those tables.

Integration with other big-data analysis: In some cases, it may be desirable to integrate the resulting data tables of LazyBase with other big-data analysis techniques such as those offered by the Hadoop project. LazyBase could easily integrate with such frameworks, potentially even treating the analysis as an additional pipeline stage and scheduling it with the coordinator on the same nodes used to run the rest of LazyBase.

6. Related work

The traditional approach to decision support loads data generated by operational OLTP databases, via an ETL process, into a system optimized for efficient data analytics and read-only queries. Recent "big data" systems such as Hadoop [26] and Dremel [31] allow large-scale analysis of data across hundreds of machines, but also tend to work on read-only data sets.

Several research efforts (e.g. [33, 36]) have examined efficient means to support both data models from the same database and storage engine using specialized in-memory layouts. RiTE [41] caches updates in-memory, and batches updates to the underlying data store to achieve batch update speeds with insert-like freshness. Similarly, Vertica's hybrid storage model caches updates in a write-optimized memory format and uses a background process to transform updates into the read-optimized on-disk format [7]. Both of these techniques provide up-to-date freshness, but require sufficient memory to cache the update stream. LazyBase separates the capture of the update stream from the transform step, reducing freshness but improving resource utilization and responsiveness to burst traffic.

FAS [34] maintains a set of databases at different freshness levels, achieving a similar effect to LazyBase, but requiring significant data replication, reducing their scalability and increasing cost. Other groups have examined techniques for incrementally updating materialized views, but rely on the standard mechanisms to ingest data to the base table, leaving the problem described here mostly unsolved [11, 35]. Further, researchers have examined data structures to provide a spectrum between update and query performance [23, 24]. These structures provide a flexible tradeoff similar to that of LazyBase, and we believe that LazyBase could benefit by employing some of these techniques in the future.

LazyBase's use of update tables is similar to ideas used in other communities. Update files are commonly used for search applications in information retrieval [16, 30]. Consulting differential files to provide up-to-date query results is a long-standing database technique, but has been restricted to large databases with read-mostly access patterns [37]. More recent work for write-optimized databases limits queries to the base table, which is lazily updated [29]. Google's BigTable [18] provides a large-scale distributed database that uses similar update and merge techniques, but its focus on OLTP-style updates requires large write-caches and cannot take advantage of the trade-off between freshness and performance inherent in LazyBase's design.

BigTable is also one of a class of systems, including Dynamo [27], SimpleDB [8], SCADS [13], Cassandra [1], Greenplum [9] and HBase [2] in which application developers have built their own data stores with a focus on high availability in the face of component failures in highly distributed systems. These systems provide performance scalability with a relaxed, eventual consistency model, but do not allow the application to specify desired query freshness. Unlike LazyBase, the possibility of inconsistent query results limits application scope, and can make application development more challenging.

SCADS [13] describes a scalable, adjustable-consistency key-value store for interactive Web 2.0 application queries. SCADS also trades off performance and freshness, but does so in a greedy manner, relaxing goals as much as possible to save resources for additional queries.

Incremental data processing systems, such as Percolator [32], incrementally update an index or query result as new values are ingested. These systems focus on continuous queries whose results are updated over time, rather than sequences of queries to the overall corpus. They do not address the same application needs (e.g., freshness vs. query speed) as LazyBase, but their techniques could be used in a LazyBase-like system to maintain internal indices.

Sumbaly et al. [39] extend the Voldemort key-value store to provide efficient bulk loading of large, immutable data sets. Like LazyBase, the system uses a pipelined approach to compute the next authority version of the data set, in this case using Hadoop to compute indices offline. Unlike LazyBase, it does not permit queries to access the intermediate data for fresher results.

Some distributed "continuous query" systems, such as Flux [38] and River [14], used a similar event-based parallel processing model to achieve scalability when scheduling a set of independent operations (e.g., analysis of time series data) across a set of nodes. However, they have no concept of coordinating a set of operations, as is required for providing self-consistency. Additionally, they are not designed to provide access to the results of intermediate steps, as required for the freshness/performance trade-off in LazyBase.

7. Conclusions

We propose a new data storage system, LazyBase, that lets applications specify their *result freshness* requirements, allowing a dynamic tradeoff between freshness and query performance. LazyBase optimizes updates through bulk processing and background execution of SCUs, which allows consistent query access to data at each stage of the pipeline.

Our experiments demonstrate that LazyBase's pipelined architecture and batched updates provide update throughput that is 4X to 5X higher than Cassandra's at all scalability levels. Although Cassandra's point queries outperform LazyBase's, LazyBase's range query throughput is 4X higher than Cassandra's. We also show that LazyBase's pipelined design provides a range of options on the read-query freshness-latency spectrum and that LazyBase's consistency model provides clear benefits over other "eventually" consistent models such as Cassandra's.

8. Acknowledgments

We wish to thank our shepherd, Guillaume Pierre; the anonymous reviewers; and Ilari Shafer for their suggestions for improvements to the paper. We thank Eric Anderson for his assistance with DataSeries. We thank the members and companies of the PDL Consortium (including APC, EMC, Emulex, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Riverbed, Samsung, Seagate, STEC, Symantec, and VMWare) for their interest, insights, feedback, and support. This research was sponsored in part by an HP Labs Innovation Research Program award and Intel, via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). This research was enabled by hardware donations from Intel and NetApp.

References

[1] Apache Cassandra, http://cassandra.apache.org/.

[2] Apache HBase, http://hbase.apache.org/.

[3] CouchDB, http://couchdb.apacheorg/.

[4] MongoDB, http://www.mongodb.org/.

[5] Twitter Earthquake Detector, http://recovery.doi.gov/press/us-geological-survey-twitter-earthquake-detector-ted/.

[6] Apache Thrift, http://thrift.apache.org/.

[7] HP Vertica, http://www.vertica.com/.

[8] Amazon SimpleDB, http://aws.amazon.com/simpledb/.

[9] EMC Greenplum, http://www.greenplum.com/.

[10] A look at Twitter in Iran . http://blog.sysomos.com/2009/06/21/a-look-at-twitter-in-iran/.

[11] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proc. SIGMOD*, 1995.

[12] E. Anderson, M. Arlitt, C. B. Morrey III, and A. Veitch. DataSeries: An efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1):70–75, January 2009.

[13] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. In *Proc. CIDR*, January 2009.

[14] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proc. Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, May 1999.

[15] C. Babcock. Data, data, everywhere. *Information Week*, January 2006.

[16] S. Buttcher and C. L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. *Proc. 14th ACM Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 317–318, 2005.

[17] R. Campbell, I. Gupta, M. Heath, S. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H. Lee, M. Lyons, D. Milojicic, D. O'Hallaron, and Y. Soh. Open cirrus cloud computing testbed: Federated data centers for open source systems and services research. In *Proc. of USENIX HotCloud*, June 2009.

[18] 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 *Proc. OSDI*, November 2006.

[19] S. Chaudri, U. Dayal, and V. Ganti. Database technology for decision support systems. *Computer*, 34(12):48–55, December 2001.

[20] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of Symposium on Cloud Computing (SOCC)*, June 2010.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004.

[22] J. Ginsberg, M. H. Mohebbi, R. S. Patel, L. Brammer, M. S. Smolinski, and L. Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, pages 1012–1014, February 2009.

[23] G. Graefe. Write-optimized B-trees. In *Proc. VLDB*, pages 672–683, 2004.

[24] G. Graefe. B-tree indexes for high update rates. *ACM SIGMOD Record*, 35(1):39–44, March 2006.

[25] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The underground on 140 characters or less. In *Proc. of ACM Conf. on Computer and Communications Security*, October 2010.

[26] Hadoop. http:// hadoop.apache.org/.

[27] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, pages 205–220, 2007.

[28] D. Henschen. 3 big data challenges: Expert advice. *Information Week*, October 2011.

[29] S. Hildenbrand. Performance tradeoffs in write-optimized databases. Technical report, Eidgenossiche Technische Hochschule Zurich (ETHZ), 2008.

[30] N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. *Proc. 27th Australian Conf. on Computer Science (ACSC)*, 2004.

[31] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB*, pages 330–339, September 2010.

[32] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. OSDI*, pages 1–15, 2010.

[33] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proc. SIGMOD*, July 2009.

[34] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS - a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proc. VLDB*, pages 754–765, 2002.

[35] K. Salem, K. Beyer, and B. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In *Proc. SIGMOD*, 2000.

[36] J. Schaffner, A. Bog, J. Kruger, and A. Zeier. A hybrid row-column OLTP database architecture for operational reporting. In *Proc. Intl. Conf. on Business Intelligence for the Real-Time Enterprise*, 2008.

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

[38] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. *Proc. ICDE*, pages 25–36, 2003.

[39] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *Proc. 10th USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[40] D. Terry. Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research, October 2011.

[41] C. Thomsen, T. B. Pedersen, and W. Lehner. RiTE: Providing on-demand data for right-time data warehousing. In *Proc. ICDE*, 2008.

[42] W. Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, January 2009.