Trading Freshness for Performance in Distributed Systems

James Cipar

CMU-CS-14-144

December 2014

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Gregory R. Ganger, Chair
David G. Andersen
Garth A. Gibson
Kimberly Keeton, HP Labs

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the an HP Labs Innovation Research Program award; Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC); National Science Foundation undergrant numbers IIS-0429334, CNS-1042537, CNS-1042543, and CNS-1117567; the US Army Research Office under grant number W911NF0910273; DARPA Grant FA87501220324 and Lockheed Martin Corp. under grant number 4100074797.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Distributed systems, databases, freshness, staleness, machine learning, parameter server, OLAP, data warehouse

To my wife, Melanie, for her patience and frequent encouragement during all of my PhD studies, and to my son and daughter, Oscar and Hazel. They all have been a constant source of support and joy for me.

Abstract

Many data management systems are faced with a constant, high-throughput stream of updates. In some cases, these updates are generated externally: a data warehouse system must ingest a stream of external events and update its state. In other cases, they are generated by the application itself: large-scale machine learning frameworks maintain a global shared state, which is used to store the parameters of a statistical model. These parameters are constantly read and updated by the application.

In many cases, there is a trade-off between the *freshness* of the data returned by read operations and the efficiency of updating and querying the data. For instance, batching many updates together will significantly improve the update throughput for most systems. However, batching introduces a delay between when an update is submitted and when it is available to queries.

In this dissertation, I examine this trade-off in detail. I argue that systems should be designed so that the trade-off can be made by the application, not the data management system. Furthermore, this trade-off should be made at query time, on a per-query basis, not as a global configuration.

To demonstrate this, I describe two novel systems. LazyBase is a data warehouse system originally designed for to store meta-data extracted from enterprise computer files, for the purposes of enterprise information management. It batches updates and processes them through a pipeline of transformations before applying them to the database, allowing it to achieve very high update throughput. The novel *pipeline query* mechanism in LazyBase allows applications to select their desired freshness at query time, potentially reading data that is still in the update pipeline and has not yet been applied to the final database.

LazyTables is a distributed machine learning *parameter server* - a shared storage system for sparse vectors and matrices that make up the bulk of the data in many machine learning applications. To achieve high performance in the face of network delays and performance jitter, it makes extensive use of batching and caching, both in the client and server code. The *Stale Synchronous Parallel* consistency model, conceived for LazyTables, allows clients to specify how out-of-sync different threads of execution may be.

Acknowledgments

I would like to thank my committee members, David Andersen, Gregory Ganger, Garth Gibson, and Kimberly Keeton. Special thanks to my advisor, Greg, and to Kim, for their collaboration and mentorship throughout this process.

In addition to Kim, I would like to thank other members of the HP Labs SIMPLE group, including my coauthors on the LazyBase paper, Charles B. Morrey III, Craig Soules, and Alistair Veitch.

More thanks to the others working on the LazyTables research: Qirong Ho, Henggang Cui, Jin Kyu Kim, Seunghak Lee, and Eric Xing. Special thanks to Qirong Ho. My discussions with him provided the initial motivation for, and design of, LazyTables. And also to Henggang Cui for his excellent work improving and evaluating LazyTables, the Stale Synchronous Parallel model, and A-BSP.

I would also like to make a general "thank you" to everyone who provided guidance and education along the way: other students, professors, PDL consortium members, and others who have taught me things, discussed interesting ideas, or just generally kept me excited about computer systems research. Among these, I would like to single out Michael Kozuch from Intel Labs, my mentor during my summer Fellowship with Intel labs, and collaborator on the Tashi, Rabbit (now SpringFS), and AlSched projects.

Lastly, I would like to thank all of the companies in the PDL consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, IBM, Intel, LSI, Microsoft, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Western Digital) for their interest, feedback, and material donations. Research in large-scale computing requires significant physical resources, and these donations truly enabled this work. This research was sponsored by the an HP Labs Innovation Research Program award; Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC); National Science Foundation undergrant numbers IIS-0429334, CNS-1042537, CNS-1042543 Gibson et al. [2013], and CNS-1117567; the US Army Research Office under grant number W911NF0910273; DARPA Grant FA87501220324 and Lockheed Martin Corp. under grant number 4100074797.

Contents

List of Figures

List of Tables

1 Introduction

There is a large and diverse class of parallel applications that generate extremely high throughput updates to shared data and require guarantees about the freshness of that data when they read it. Such applications appear in many fields, including genomics, retail, transportation, finance, and information management. Additionally, these applications frequently require continuous updates to the shared data and strict query latency requirements. In this dissertation, I explore the exploitation of looser freshness requirements in system design to improve the overall performance of these applications.

While query efficiency, low query latency, and fresh results are all desirable properties, many techniques to achieve high update throughput are detrimental to at least one of them. For instance, batching many updates into a single group can vastly improve update throughput. However, the extra time required to collect and process batches introduces a latency between when an update is submitted and when the resulting batch is available to be read. Furthermore, while very large batches are desirable from an efficiency standpoint, a large batch may require an even longer time to process, introducing additional latency. This latency is manifested in reduced data freshness for queries: if the system takes a long time to make updates available for queries, then the data that is available immediately may be stale. The example of batching, as well as other similar techniques such as caching and group commit, demonstrate how the design of storage systems involves a trade-off between the efficiency of updates and the freshness (or inversely, the staleness) of the data that is returned by queries.

For many systems, different queries have different desires regarding this trade-off, even when the queries are accessing the same data set. Consider an analytical database storing Twitter data. An application trying to report the "hot news" that people are currently talking about has a freshness requirement on the order of a few minutes, otherwise the news is not really fresh. Another application that analyzes the structure of the social network graph might have a freshness requirement on the order of days because the large-scale structure of the network changes slowly.

Such diverse applications can benefit from storage systems that allow the application to make decisions regarding this trade-off at query time. However, existing solutions typically

provide only a single point in this trade-off space.

In this thesis, I argue for system designs that allow applications to manage this trade-off *at query time*. I demonstrate that freshness requirements are often a property of the query, not the data set or even the application, and that making the trade-off explicit enables greater overall efficiency.

Thesis Statement

> **There is an inherent trade-off between data freshness and query efficiency. Data management systems can better suit applications' needs by allowing applications to make decisions about this trade-off on a per-query basis.**

To support this thesis, I show that (a) techniques to improve performance often introduce staleness in data, (b) it is often possible to do work during a query's execution to avoid this staleness when fresh results are required, but that (c) staleness tolerance is common enough that many queries can avoid this extra work.

I demonstrate these points in the context of two case studies in applying this trade-off to real data management systems. The first case study (Chapter 3) examines a database that is designed to support continuous, high-throughput updates and inserts while also allowing efficient analytical queries over the data. The second case study (Chapter 4) describes a system to support shared data structures for large-scale, data-intensive machine learning. The remainder of this chapter will provide an overview of previous work in staleness (detailed in Chapter 2), and introduce the two case studies.

Background

The idea of exploiting staleness-tolerance to improve system efficiency has been used in numerous ways, even if it is not always stated as such. Techniques like batching updates, as described above, or caching results, are second-nature to system designers. Both of these techniques can increase the staleness of the data seen by the consumer. Batching does so by increasing the latency of propagating updates from a source to a central repository. Caching increases the time to propagate updates from the central repository to the consumer. Systems that support disconnected operation [Kistler and Satyanarayanan, 1992] or user-driven branching and merging (e.g. source code revision control) represent other cases where staleness is introduced into a system design.

Previous work has studied data freshness (and conversely, staleness) in the context of consistency models. This body of work generally attempts to describe and classify the consistency offered by different system designs. TACT [Yu and Vahdat, 2002] describes

a family of consistency models characterized along three axes, one of which is freshness. Other work [Terry, 2011b] describes a grab-bag of consistency properties that can be mixed and matched, including "bounded staleness", "read-my-writes", and "monotonicity".

In addition to considering staleness in the context of describing a consistency model, there is precedent for allowing application designers to control a freshness/latency trade-off for specific systems. Previous work [Bright and Raschid, 2002] has proposed modifications to the HTTP protocol that allow an application to specify a freshness-latency trade-off curve for each request. Caching servers can use this curve to decide when to serve a request from cache and when to fetch results directly from the source. Other work [Alexandros Labrinidis, 2003] looks at the trade-off between freshness and latency in building *materialized web views*, essentially pre-computing graphics and layout for web pages based on potentially stale data in a database.

Analytical Databases

Building on this existing body of work, I explore these ideas in the context of two case studies, the first being an analytical database. Many applications in modern data collection and analytics require analyzing large data sets that grow and change at a high rate. These applications include business and purchase analytics, hardware monitoring, click stream and web user tracking, and social media analysis. In addition to high update throughput, these applications require efficient analysis of a reasonably up-to-date version of the database. For instance, real-time traffic data may be used for emergency response, real-time traffic maps, as well as traffic engineering.

These three example applications are illustrative of the different freshness requirements that applications may have. Emergency response personnel require data that is extremely fresh: a few minutes delay in responding to an accident could have tragic consequences. Real-time traffic maps must be up-to-date to be useful, but changes in traffic patterns typically occur on timescales measured in minutes, not seconds. Furthermore, the consequences for out-of-date information are much less dire than they are for emergency response. Lastly, traffic engineering is not concerned with current patterns, but instead with long-term trends. These trends are aggregated over periods of weeks or months. Thus, very fresh results are inconsequential to the application.

To support such applications, three types of existing solutions have emerged that each settle on a different trade-off. Relational databases (RDBMSs) – generally accessed via the SQL query language – provide a well defined consistency model and very up-to-date results. However, the cost of this is relatively inefficient update and query performance, and limited scalability. Data warehouses – sometimes called analytical databases – are designed for extremely high performance analytical queries, at the cost of very inefficient

updates. To mitigate this inefficiency, updates are performed in large batches (i.e. an extract-transform-load pipeline), a technique that introduces staleness on the order of hours or even days. NoSQL databases drop the strong consistency of RDBMSs in exchange for high performance and scalability. However, these systems can be difficult to work with because they provide no guarantees for either the consistency or freshness of the data. Furthermore, even "write-optimized" NoSQL databases tend to be designed around point updates and point queries and are inefficient when faced with a very large volume of updates.

The first case study presents a novel analytical database design and a prototype implementation called LazyBase. LazyBase supports continuous, very high throughput updates. Experimental results show that it can process updates at more than 4x the rate of Cassandra – a popular NoSQL database known for its write performance. It does this by way of a pipelined design. At the head of the pipeline are *ingest servers* whose sole purpose is to accept updates and collect them into large batches. By making the ingest servers very simple, they can accept high rates of updates even when the updates being sent to them are not batched. After being collected into batches, the updates are transformed by additional pipeline stages that handle foreign key constraints, sorting, and merging of data. These pipeline stages benefit from the batching done by the ingest servers, allowing them to operate with great efficiency.

This design resembles the extract-transform-load (ETL) pipelines of typical data warehouses (a.k.a. Analytical or OLAP databases). Naively-implemented, it would also have the same drawback: very stale data in the database. With a complex workload and large batches, data can take minutes or even hours to move from the ingest server to the end of the pipeline. This means that queries may be operating over data that is many hours old.

To offset this, LazyBase allows the query engine to access intermediate data created by the pipeline stages. Even the raw input saved by the ingest servers is available to query. However, querying this data comes at a cost: it may be unsorted, in many separate files (unmerged), or even have incorrect keys (because foreign keys have not been mapped yet). The query engine can overcome all of these problems, but it incurs overhead in doing so, and efficiency is reduced for queries that have to look at data from earlier pipeline stages. For instance, querying unsorted data requires a full scan. In some cases, the performance cost of this may be prohibitive. However, in many configurations, the pipeline is able to move data from the unsorted to sorted stages quickly. With small amounts of unsorted data, it may be feasible to perform a full scan of the unsorted data, while using more efficient query plans to access other data.

These *pipeline queries* provide applications with a tunable trade-off between query efficiency (thus latency) and data freshness. A query that can accept stale data does not

need to query any intermediate pipeline stages. It can simply look at the result of the pipeline – called the *authority table*. Querying the authority table is very efficient, therefore query latency will be low. On the other hand, if the query needs fresher data it will have to look at some (or all) of the data that is still in-flight in the pipeline. This may involve full scans to query unsorted/unindexed batches, mapping keys between different files, and merging many smaller files together. While these operations are much faster than waiting for data to get to the end of the pipeline, they do incur significant overhead. This overhead is seen by the application as an increased query latency.

LazyBase is motivated by the diverse range of freshness requirements in analytical database applications. Many applications can tolerate some staleness in their query results. LazyBase exploits this staleness tolerance to allow a system design that meets other requirements of such a database. Specifically, it is designed to support extremely high update throughput through the extensive use of batching and a queryable update pipeline.

Parallel Machine Learning

Another important class of applications employs iterative convergent computations, such as those used in machine learning algorithms. These algorithms have applications in data mining, web search, genomics, advertising, and mapping. In the second case study, I examine systems designed to support the intermediate data that these algorithms create, access, and update as they run.

These applications create large, shared data structures that they operate over. Commonly, these data structures are sparse matrices or vectors of numbers. Each thread of execution performs many updates at a very high rate – hundreds of thousands per thread per second – interspersed with infrequent read accesses. For instance, in Latent Dirichlet Allocation (a data mining algorithm), each thread reads a portion of a data set, then issues many increment and decrement operations to elements of a shared matrix. There is little locality to these operations: any thread may update any part of the matrix at any time.

A few techniques have emerged for running these applications in a parallel or distributed setting. The Bulk Synchronous Parallel model [Valiant, 1990] – often used for simulations in scientific computing – offers a predictable level of delay when updating shared data. This comes at the cost of substantial communication and synchronization overhead. On the other end of the spectrum, completely asynchronous systems offer lower overhead, but with no control over the freshness of data. Other systems such as GraphLab [Low et al., 2010] attempt to improve performance by exploiting locality of communication – where it exists – to avoid some synchronization costs.

In the second case study, I propose a consistency model – called *Stale Synchronous*

Parallel – that suits these types of applications. It is based on the *Bulk Synchronous Parallel* model that is commonly used in scientific computing and parallel machine learning. Stale Synchronous Parallel (SSP) extends this model allowing updates to be delayed by a fixed amount as they are propagated from one thread to another. In other words, the application allows some staleness to be introduced in the updates it receives from other threads.. This delay increases the efficiency of the computation by allowing the system to batch and combine many updates and to cache data rather than reading it from another thread.

As in the first case study, it is important to provide the application with *query-time* control over the freshness of its data. In the case of convergent applications, staleness may affect the convergence rate differently for different applications, configurations, or even data sets. This study demonstrates experimentally that there is a "sweet spot" in the freshness-performance trade-off, and that the location of this sweet spot depends on factors including the configuration of the algorithm, the number of threads, and the particular dataset being analyzed. Therefore, providing run-time configuration of this trade-off is important to letting applications use the correct staleness settings.

In addition to describing a consistency model that supports these applications, the second case study describes the design and implementation of a system employing this consistency model. LazyTables is a distributed parameter server and associated client-side libraries designed to support high performance machine learning algorithms. Using LazyTables, the second case study presents experimental results demonstrating the freshness-performance trade-off for three well-known machine learning algorithms. It examines overall performance – e.g. how long it took to find a good solution – as well as components of that, such as the iteration rate and the rate of convergence per iteration.

This case study also describes another consistency model, called *Bounding Ball Staleness* that allows application programmers to specify freshness requirements in terms of numerical error instead of iterations. It describes how Bounding Ball Staleness can be implemented on top of SSP for a wide range of applications. In addition to a basic implementation on top of SSP, I discuss potential design decisions that could allow a parameter server to support Bounding Ball Staleness more effectively.

The remainder of this dissertation is structured as follows. The next chapter (Chapter 2) presents background information and related work in detail. Afterwards, Chapters 3 and 4 present the two case studies. Chapter 5 concludes the work with a summary of contributions and directions for future work.

2 Background and Related Work

Staleness tolerance is a common feature among a diverse range of applications, and can be exploited to improve the efficiency of those applications. In addition to the case studies showing this for two classes of applications (Chapters 3 & 4), this chapter presents related work demonstrating these principles in other contexts. Related work that is specifically relevant to either of these case studies will be presented within the context of the respective chapter.

Other work exploiting the staleness/efficiency trade-off falls into four broad categories. (1) Some techniques are used across all areas of computer systems that exploit applications' staleness tolerance without the application designer explicitly recognizing the trade-off. (2) Additionally, there are system designs that acknowledge this trade-off, but do not provide a way for applications to control it. (3) There is additional work – including the case studies presented in this thesis – that recognizes both the trade-off and the utility of allowing applications to control it. (4) Lastly, there is a more abstract area of work describing consistency models that incorporate some notion of staleness.

2.1 Common Techniques Exploiting Trade-Off

There are commonly-used techniques where the idea of exploiting staleness-tolerance for efficiency has been used even when it is rarely stated as such. In some cases, such as batching and caching, these techniques are so commonly applied that they are often not considered part of a wider trade-off. In other cases, such as bulk-loading a data warehouse, the freshness of the data may be considered, but the trade-off is not recognized because the other option (individual insert and update operations) is too slow to be feasible. When the trade-off exists but is rarely recognized, applications cannot be given explicit control over it.

Batching

Batching is a general technique where many updates are collected and applied at once. In many systems, applying many updates at once can improve the efficiency of the entire

update operation. As an example, the time to complete many database transactions is dominated by disk latencies (both seek time and rotational latency), and not disk throughput. This means that the time to execute a transaction is not dependent on the size of the update unless the update becomes very large and write times overwhelm latencies. If multiple logical updates are grouped together and submitted to the database as a single transaction, the cost of the seek and rotational latency can be amortized across all of them. As a concrete example, the SQLite single-disk database has an average per-insert latency of about 13ms when inserts are done one-by-one, but only 37μs when inserts are batched into groups of 25000 [SQLite]. The batched case is more than 350 times faster than the single insert case.

Batching also creates opportunities for other optimizations. For instance, some types of updates may be coalesced into a smaller set of updates. Two `put` operations to the same location (`x = 5; x = 10`) can be replaced by the latter operation (`x = 10`). Similarly, two additions (`x += 5; x += 10`) can be replaced by a single addition (`x += 15`). Coalescing updates can reduce the total volume of updates that need to be executed.

Caching

Another common technique is caching, where stale values are stored in a fast-to-access memory to avoid accesses to an up-to-date but expensive source. Caching is a staple of computer system design, being used across all levels of the memory hierarchy and within applications such as HTTP servers and clients, distributed file systems, databases, and many more. Caches such as the processor's hardware memory cache strive to be fully coherent – that is, never delivering stale data. However, techniques to maintain coherent caches require many messages to be delivered between the different components. While this is practical for dedicated hardware attached to the same memory system, it is usually viewed as unnecessarily expensive in other contexts. In these other contexts (file systems, databases, web caches), the cache is often allowed to deliver stale data to the client in order to save messaging overhead. This decision implicitly makes a trade-off between staleness and efficiency.

Bulk Loading

Data warehouse systems provide efficient read-only queries to large data sets. However, updating the data row-by-row – as in a transactional database – is often very inefficient. To avoid this inefficiency, data is loaded into the system in bulk. For instance, in Amazon's Redshift database, individually inserting rows can take over 10 seconds per row. Bulk loading 3.5 billion rows takes only 6 minutes [Lau and Gabrielski]. With individual inserts, this would take over 1000 years.

Bulk loading is a special case of the batching technique described above, but is worth

considering separately because batching is usually viewed as an optimization that may be useful, but is not strictly necessary. On the other hand, typical data warehouse technology does not allow for effective row-by-row inserts or updates in the same way that a transactional database does.

2.2 Trade-Off Recognized, but Not Controllable

Many systems acknowledge a trade-off between freshness and efficiency, but do not allow the application to control this trade-off on a per-query basis. Research papers in this category present their design as one point on the spectrum of this trade-off, or allow a global setting to control the trade-off for the entire system. Additionally, these papers typically try to provide a definition for freshness.

Applying Update Streams in a Soft Real-Time Database System

This paper [Adelberg et al., 1995] describes view management in the STRIP real-time database. STRIP attempts to schedule real-time transactions to meet deadlines, while simultaneously ingesting a continuous stream of updates. The authors argue that these updates should not be treated as real-time transactions, as the overhead of managing them would be too high. Therefore, STRIP has separate components for transaction processing and ingesting updates. Allocating resources between these systems presents a trade-off between the transaction latency and the freshness of the underlying data.

The paper provides a few definitions of data staleness that can be considered for such a system. *Maximum age* assumes that each update contains a timestamp from the source of the update, which is appended to the corresponding row in the database. If the timestamp of a row is greater than some application-defined maximum, the row is considered stale. This definition assumes that there are periodic updates even when values haven't changed, otherwise rows will become stale. *Unapplied update* is an optimistic definition. It assumes that data is fresh unless a newer update has been received that has not yet been applied. Unlike *maximum age*, this definition ignores delays that occur before data is handed to the database system.

In addition to the previous two definitions, the paper describes variations and combinations of them. For instance, replacing generation time with arrival time for *maximum age*, or considering an object stale if it is stale under either *maximum age* or *unapplied update*.

The paper proposes a set of metrics for measuring the effectiveness of a real-time database with a notion of staleness. They are the average fraction of objects that are stale, the fraction

of transactions that don't complete by their deadlines, the fraction of transactions that complete without having to use stale data, the fraction of completed transactions that used fresh data, and the average "value per second" of completed transactions for some utility function over freshness.

How to Roll a Join: Asynchronous Incremental View Maintenance

This work [Salem et al., 2000] targets the problem of maintaining materialized views of external data. It argues that keeping materialized views completely up-to-date is too expensive and unnecessary for most applications. Additionally, large updates to materialized views cause contention for locks and other resources that can delay other transactions. The paper argues in favor of sacrificing some freshness of the materialized view in exchange for overall increased performance for the database.

To support this, the paper describes a technique called *"rolling join propagation"* that reduces the latency of updating materialized views. In a typical system it is only possible to update the materialized view to the current state of the database. When the materialized view has not been updated in a long time, this update will take a long time and will cause significant delays. With rolling join propagation the database keeps an update log for all tables that the materialized view depends on. This log is augmented with timestamps, allowing the database to incrementally roll the view forward in a series of short transactions.

Performance Trade-Offs in Write Optimized Databases

> *"We want to provide both good query performance and good update performance. To achieve this, we allow a limited amount of staleness in the query answers."* [Hildenbrand, 2008]

This paper [Hildenbrand, 2008] makes the observation that most database architectures make a trade-off between update performance and query performance. It posits that there is another dimension to the trade-off: freshness. By allowing some staleness in query results, the system can achieve both high update performance and high query performance.

The paper describes a write optimized database meant to ingest constant streams of data, e.g. for environmental monitoring, while supporting efficient ad-hoc queries. The database does this by keeping most of the data in a read-optimized B+-Tree data structure, but committing updates to a write-optimized Δ log. Queries are only sent to the read-optimized data structures, ensuring efficient performance at the cost of freshness.

The proposed architecture requires that the freshness/efficiency trade-off be configured globally for the database (or table), rather than adjusted dynamically based on the needs of the application or the specific query being executed. Experiments demonstrate that their

applications typically observe about 1 second of staleness.

2.3 Controllable by Applications

The most closely-related body of related work not only recognizes a trade-off between freshness and latency, but also allows this trade-off to be controlled in detail by the application.

Quorum Requirements

A class of distributed key-value stores – including Dynamo [Hastorun et al., 2007] and Cassandra [Cassandra] – provide the application designer with a trade-off between latency and freshness by allowing fine-grained control over quorum requirements used when reading and writing data. A Dynamo cluster consists of a number of servers storing keys and their associated values. Each key-value pair is replicated across N servers, both for performance and fault tolerance. When an application wishes to write a new key-value pair to the system, or update an existing one, it sends the request to one of the Dynamo servers, along with an integer parameter $W \leq N$. That server, the *coordinator* for the write, determines the N servers that the data should be replicated to, and forwards the request to them. It returns a success message to the application once at least W of the servers have responded.

Similarly, when an application wishes to read a value, it sends a request parameterized by an integer $R \leq N$. The coordinator for the read operation forwards the request to all N servers that may have data for that key. Once R of them have returned to the coordinator, the coordinator returns the value that has the most recent timestamp. The parameters W and R provide a trade-off between latency, freshness, and consistency.

If $W + R > N$, read operations will always see the value of the latest completed write. However, even in this scenario there is some staleness: a write may have been issued, but not yet completed. Thus, the write latency introduces a degree of staleness to the result of read operations. By adjusting the relative values of W and R, the application designer can choose between favoring write latency or read latency. In other words, this presents a trade-off between read latency and freshness.

A more extreme case of staleness occurs when read and write latency are both important, and the application is willing to sacrifice consistency. By setting $W + R < N$, the system becomes eventually consistent. It is possible for a read operation to return an old value for a key, even after the write operation has completed. This can introduce an even greater

degree of staleness than the case where W is large. However, this added staleness may be acceptable in exchange for efficient read *and* write operations.

Using Latency-Recency Profiles for Data Delivery on the Web

Bright and Raschid [Bright and Raschid, 2002] describe a latency-freshness trade-off that exists when delivering content from a web cache. In a web cache system, when a request is sent to a cache server, the server must decide whether it will serve the request from cache, or get a new copy from the source. This represents a trade-off between high latency (refreshing) or potential staleness (using cache). Policies like cache time-to-live timestamps enforce a staleness bound, but do so in a global way.

The authors argue that this trade-off should be configurable by the requesting application on a per-request basis. The basis for this argument is twofold: first, a single application may have different requirements depending on the reason it is querying the data. Secondly, a web service may have many thousands or even millions of unique clients. Maintaining freshness preferences for all of them presents a scalability challenge. Not only could this be a significant amount of data, but it would require augmenting the cache server with mechanisms for registering new clients and expiring retired clients.

In order to configure this trade-off on a per-query basis, each request includes a *latency-recency profile* describing the client's preference for that request. These profiles are the parameters to an objective function that balances latency and recency. By estimating the staleness of the cache and the time to refresh from the source, a cache server is able to optimize the utility of this objective function. The result of this optimization is the decision to refresh, or serve from cache.

Balancing Performance and Data Freshness in Web Database Servers

Labrinidis and Roussopoulos [Alexandros Labrinidis, 2003] describe a trade-off between performance and freshness in the context of web servers delivering dynamic content. The server architecture consists of four layers. 1) A web server that handles outside requests. 2) An application server that builds responses. 2) A database server that stores the underlying data, and receives updates from external sources. 4) An asynchronous cache that sits between the application server and the database server.

The focus of the work is on the policies used to manage data in the asynchronous cache. The cache stores *WebViews*, i.e. snippets of HTML or XML that are built by querying the underlying database and applying some application-specific logic to the results of those queries. WebViews are concatenated together by the application server to build a complete web page in response to a user request. The authors recognize two ways that the application server can maintain WebViews.

Cached views are created on-demand in response to application server requests. The resulting WebView is stored in the cache until it is invalidated by an update to the underlying database. After being invalidated, it is removed from the cache, and will not be rebuilt until a subsequent request requires it. This can be thought of as a "pull" model for maintaining the cache, where data is pulled by the application server requests into the cache.

Materialized web views follow a "push" model, where the back-end database pushes updates to the cache. In this case, whenever an update is received, the database enqueues a rebuild operation of all WebViews that depend on this data.

These two caching mechanisms represent a trade-off between freshness and latency. Cached views are always fresh, as they are immediately invalidated when the underlying data changes. However, after being invalidated the next request will have to rebuild the view. This requires an expensive database access. On the other hand, materialized views are stale whenever there is a backlog of updates that have not been applied to the cache server. However, accessing materialized web views is always fast, as they are always served from cache.

The paper presents an algorithm, OVIS(Θ), that automatically determines a *materialization plan* – the list of WebViews that should be materialized vs cached. The parameter Θ controls the target average freshness for generated web pages. Freshness for a web page is defined as a weighted average (the weights being application-defined) of the freshness of all WebViews that make up the page. The freshness of a WebView is simply a binary value: 1 if the WebView incorporates the most recent updates (it is fresh), and 0 otherwise (it is stale).

Unlike definitions of staleness in other work, there is no temporal notion of staleness. A WebView is either stale, or fresh. The freshness of a page can vary continuously between 0 and 1, but only because it is a weighted average, and not as a result of elapsed time, number of updates, or magnitude of updates that have not been included in the view.

Temporal Notions of Synchronization and Consistency in Beehive

Another notion of staleness is used in Beehive [Singla et al., 1997], a distributed shared memory system. The motivating application for Beehive is distributed processing for interactive virtual worlds. The authors observe that completely up-to-date results are rarely needed in this context. They argue that using stale data is acceptable as long as the degree of staleness can be bounded by the application. Because Beehive targets interactive virtual worlds, the main goal is a notion of staleness that is tied to real time (as perceived by a human user). However, the system supports arbitrary *virtual time* defined by the application. The advancement of virtual time is controlled by the application, and may correspond to

artificial time such as the time in a simulation. In this case, virtual time may progress much more slowly (or quickly) than real time. E.g. it may take many hours to simulate a physical process that takes only a few seconds in reality.

In Beehive, each process maintains its own private virtual time, with the *global virtual time* tracking the minimum virtual time across all threads. Beehive enforces a staleness bound when threads try to advance their own virtual time: if the difference between a thread's new virtual time and the global virtual time exceeds a globally specified staleness bound ("*delta*"), the thread must wait for the global virtual time to catch up before it can continue. This is similar to the concept of soft synchronization that I will present in Chapter 4.

SCADS: Scale-Independent Storage for Social Computing Applications

The SCADS [Armbrust et al., 2009] database architecture is designed for clustered storage on cloud-based systems where resources can be frequently added and removed to adjust for demand. The design "...allows the developer to declaratively state application specific consistency requirements, takes advantage of utility computing to provide cost effective scale-up and scale-down, and will use machine learning models to introspectively anticipate performance problems and predict the resource requirements of new queries before execution."

SCADS introduces a declarative language that application developers can use to specify the trade-off between consistency (including freshness) and performance. Once specified, the SCADS system tries to automatically tune itself to meet these requirements. The requirements are made along 5 axes:

1. Performance. Affects latency and availability. E.g. "99.9% of requests succeed in less than 100ms"
2. Write consistency. Affects updates. E.g. "Writes must be serializable, last write wins"
3. Read consistency. Affects freshness. E.g. "Stale data updated within 10 minutes"
4. Session guarantees. Affects my actions. E.g. read-my-writes or monotonic reads.
5. Durability SLA. Affects data durability. E.g. "Data must persist with 5 nines of reliability"

These provide a flexible way for applications to control the trade-off between various consistency properties and freshness and performance. A later paper [Trushkowsky et al., 2011] describes an implementation of the automatic performance tuning portion of the design, but the implementation does not support different consistency configurations.

Speedy Transactions in Multicore In-Memory Databases

Silo [Tu et al., 2013] is an in-memory database designed for high performance and scalability to large numbers of cores (e.g. 64 cores). Silo provides two levels of freshness and allows read-only queries to choose the appropriate level. By default, all transactions in Silo – read-only or otherwise – are linearizable. This means that every successful transaction effectively runs on the most recent version of the data: transactions that see stale data must abort. However read-only transactions may opt for snapshot consistency, where they will operation on a stale but consistent snapshot of the database. In this case, "consistent" means that it was the state of the database at the end of some linearizable transaction. Queries that use snapshot consistency (i.e. stale data) do not abort, and as a result observe higher performance than running the same query with linearizable consistency (i.e. fresh data).

2.4 Consistency Models

The last body of work studies the trade-off between freshness and performance in the context of abstract consistency models. These models may be descriptive – attempting to classify existing systems based on an abstract model – or prescriptive – proposing a model that new systems should be built around.

Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services

> *"We argue that an important class of applications can tolerate relaxed consistency, but benefit from bounding the maximum rate of inconsistent access in an application-specific manner."* [Yu and Vahdat, 2002]

This paper [Yu and Vahdat, 2002] presents TACT, a consistency model that provides a flexible way of describing different levels of consistency using three metrics: numerical error, order error and staleness. TACT assumes that a system is constructed of many replicas that each store a copy of the shared data. These replicas may buffer writes locally before communicating them to other replicas.

The policies surrounding this buffering are affected by the three consistency metrics. Numerical error is defined to be the maximum weight of writes not seen by a replica. Order error is the maximum number (not weight) of not-yet- communicated local writes. Staleness is the maximum real time that a replica can stage writes before communicating them.

This definition of staleness measures the real time that the write can be buffered, unavailable for other hosts to read. Additionally, the definition of order error is similar to the

definition of staleness used in the LazyTables system, described in Chapter 4.

Eventual Consistency Explained Through Baseball

In this technical report [Terry, 2011b], Terry argues that different clients benefit from different consistency guarantees, noting that "eventual consistency is insufficient for most of the participants, but strong consistency is not needed either." This paper classifies consistency models as a grab-bag of individual consistency properties that can be mixed and matched to create a complete consistency model that is appropriate for the application. It notes that stronger consistency guarantees generally correspond to reduced performance and availability. Therefore, there is a trade-off between including more of these consistency properties and leaving them out to provide improved performance. Among the consistency properties presented is *bounded staleness*, a property that enforces a real-time bound on how stale the results of a read operation may be.

In addition to describing the trade-off between consistency (including freshness) and performance, the paper argues that this trade-off is application dependent. Multiple applications accessing the same data set may have vastly different consistency requirements, and therefore may prefer a different point in the trade-off. This is explained by analogy with the observers of a baseball game. For instance, the scorekeeper requires read-my-writes consistency to ensure that scores are updated correctly. A radio reporter does not need up-to-date results, but does require consistent prefix and monotonicity; that is, the score reported must have been the correct score at some point in the game, and cannot go backwards. On the other hand, the umpire needs up-to-date results in the middle of the ninth inning to determine if the home team needs to bat or can retire early because they have already won the game.

A Framework for Analysis of Data Freshness

In [Bouzeghoub, 2004] Bouzeghoub and Peralta give a survey of data freshness work in information systems. The survey "presents an analysis of these definitions and metrics [of data freshness] and proposes a taxonomy based upon the nature of the data, the type of application, and the synchronization policies underlying the multi-source information system." Specifically they are concerned with *Data Information Systems (DIS)*: systems that integrate data from different independent data sources and provide uniform access to the data. Among other examples, they talk about data warehouses and web portals.

This work puts freshness in the context of other *quality factors* used to measure the quality of information. It argues that the term "freshness" should remain imprecisely-defined, as an umbrella term for everything that intuitively seems like it falls under the heading of "data freshness". Under this heading, more specific terms with precise definitions are given:

Currency is the time elapsed since source changed without being reflected in materialized view. This is often approximated as difference between *extraction time* and *delivery time*. This is "staleness" in TACT.

Obsolescence is the number of updates to source that have not been applied to the materialized view. This is "numerical error" in TACT.

Freshness rate is the fraction of entries that are up-to-date.

Timeliness "measures the extent to which the age of the data is appropriate to the task at hand". Estimated as time elapsed since update vs update rate of data.

In addition to these types of freshness, the survey provides a list of synchronization models that can be used to classify existing systems. The models assume three components: a data source, a repository, and a data user. They introduce a notation that describes how data is moved from the repository to the user (either "push" or "pull"), how data is moved from the source to the repository (also "push" or "pull"), and whether the source-to-repository movement is triggered synchronously by a user request or asynchronously in the background. Synchrony is noted by a "-" and asynchrony by a "/".

For instance, the LazyBase system described in Chapter 4 is an example of a "pull/push" system. "pull/push" indicates that data moves from the repository to the user in response to a user request ("pull"). A background job on the data source asynchronously pushes data to the repository ("/push"). Similarly, "pull-pull" indicates that the user requests data from the repository, which synchronously queries the source. In this classification, a system with no central repository, where users make requests directly to the source, would also be classified as "pull-pull".

Obsolescent Materialized Views in Query Processing of Enterprise Information Systems

[Gal, 1999] presents a technique for analyzing the costs of using obsolete (i.e. stale) data in a database query. Stale data is introduced by way of materialized views that are precomputed from source data. Every time the source data is updated, the materialized view becomes stale. If fresh data is required, either the materialized view must be refreshed first, or the query must access the source data directly. Both of these may impose a significant computational cost on the query, which is modeled by traditional query optimizer cost models.

This work extends the query optimizer cost models with an *obsolescence cost*, which is charged to a query plan for using stale data. The parameters of the obsolescence cost control the trade-off between efficiency and freshness. If obsolescence is considered very costly, the query optimizer will prefer to do a substantial amount of computation to avoid

using stale data. On the other hand, if the obsolescence cost is low, the optimizer will prefer efficient queries that operate over stale data.

Probabilistically Bounded Staleness

This series of papers [Bailis et al., 2012, 2013, 2014] proposes – and uses – a consistency model based on Bounded Staleness. Probabilistically Bounded Staleness (PBS) specifies the probability of reading a write after Δ seconds ((Δ, p) semantics), the probability of reading one of the most recent K versions ((K, p) semantics), or a combination of the two ((K, Δ, p) semantics). The work focuses on quorum-replicated services like Dynamo [Hastorun et al., 2007], and describes a model for predicting PBS consistency in such systems based on the effects of latency and message reordering.

2.5 Summary

This chapter provided an overview of previous work that involves a trade-off between efficiency and data freshness. It classified this work into four categories: 1) where the trade-off exists but is not (or rarely) recognized, 2) where the trade-off is recognized but not controllable by the application, 3) where the trade-off is under application control, and 4) theoretical work in consistency models that mentions such a trade-off. The next two chapters (3 and 4) expand on this third body of work. They present two novel systems that exploit the trade-off between latency and freshness by giving the application direct control over it.

3 LazyBase: Exploting Staleness in an Analytical Database

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 trade-off 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.

3.1 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) [Chaudri et al., 2001]. 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 [Cassandra], HBase [HBase], CouchDB [CouchDB] and MongoDB [MongoDB], 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 trade-off 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 key space 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 chapter describes 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 trade-off.

3.2 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 3.6.

Data analytics applications. Insights and predictions extracted from large, ever-growing data corpora 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 3.1 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 [Henschen, 2011].

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 & network monitoring	automated problem detection and diagnosis	human-driven diagnosis, online performance charting	capacity planning, availability analyses, workload characterization

Table 3.1: Freshness requirements for application families from a variety of domains.

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 [Babcock, 2006]. In addition to transaction records from physical point-of-sale systems, modern retailers exploit data like click streams, 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 [Evans, 2009], flu/cold epidemics [Ginsberg et al., 2009], and even provide an early warning system for earthquakes and other events [Twitter Earthquake].

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.

Data ingest: Such applications rely on large data corpora that are updated rapidly, such as click streams, 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 [Vogels, 2009], are difficult to reason about. As a result, application developers generally seek stronger properties [Terry, 2011a, Vogels, 2009], 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 [Terry, 2011a]) 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 3.1.

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 trade-off between latency and result freshness.

3.3 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.

3.3.1 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 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. Simple, common update operations, such as incrementing or decrementing values, can be implemented as special cases.

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 3.3.2. 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 [Thrift] RPC calls, to provide the data values for each row update. For efficiency, queries are specified programmatically. Like MapReduce [Dean and Ghemawat, 2004], 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 3.3.7.

3.3.2 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 $\langle$A, B, C$\rangle$ and its primary view key could be $\langle$A, B$\rangle$, 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 host name 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.

3.3.3 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 3.1 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

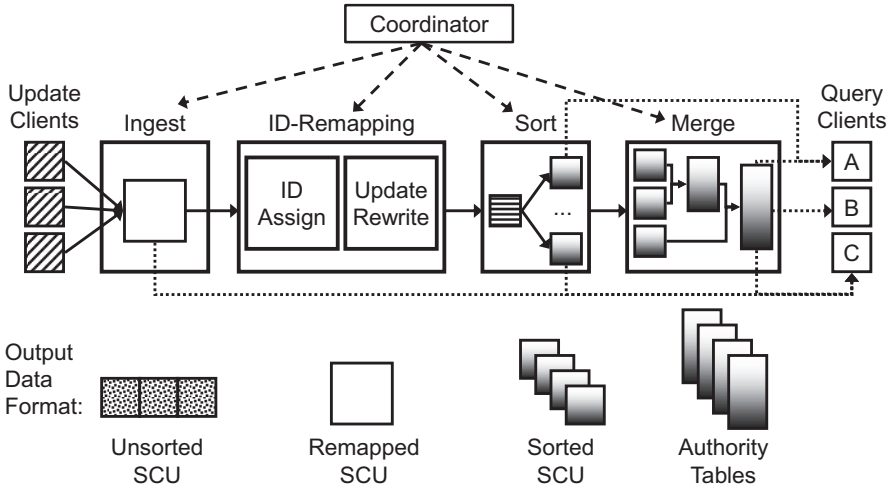


Figure 3.1: LazyBase pipeline.

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 3.3.6.

3.3.4 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.

3.3.5 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

[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.

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 [Cassandra].

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.

3.3.6 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 re-processing 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.

3.3.7 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 data flow: 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, similar to the "HAVING" clause in SQL. This design was chosen mainly for the simplicity of its implementation. There is ongoing work to implement a complete SQL query layer on top of LazyBase.

3.3.8 Data Storage

Our implementation makes heavy use of DataSeries [Anderson et al., 2009], an open-source compressed table-based storage layer designed for streaming throughput. DataSeries stores sets of 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.

3.4 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 [Cassandra], a popular scalable NoSQL database that is considered to be "write-optimized" in its design [Cooper et al., 2010]. 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.

3.4.1 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 [Campbell et al., 2009].

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 [Grier et al., 2010]. 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 3.4.5 describes this data set in more detail.

3.4.2 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.

Efficiency

LazyBase batches together updates into sizable SCUs to achieve high-throughput ingest with stronger semantics than eventually consistent systems. Figure 3.2 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

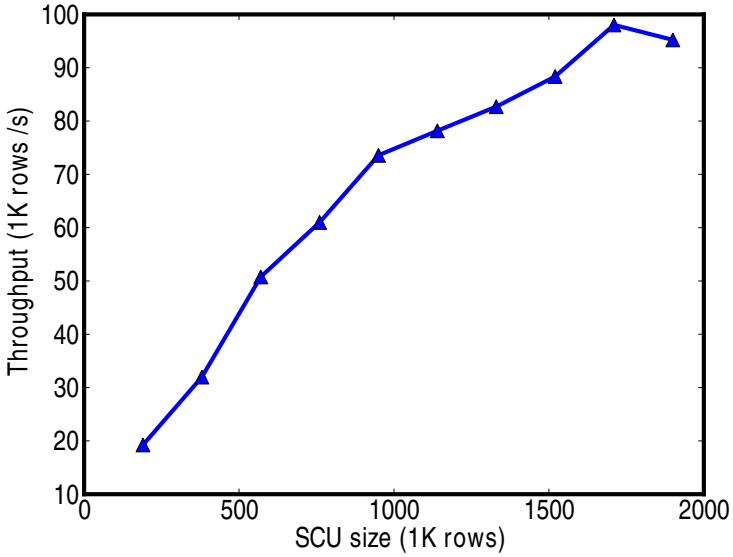


Figure 3.2: Inserts per second for different SCU sizes

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

choose a default SCU size of 1.95M rows for use in subsequent experiments.

Table 3.2 lists the average latency and throughput of each pipeline stage for our workload. Because the merge stage 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.

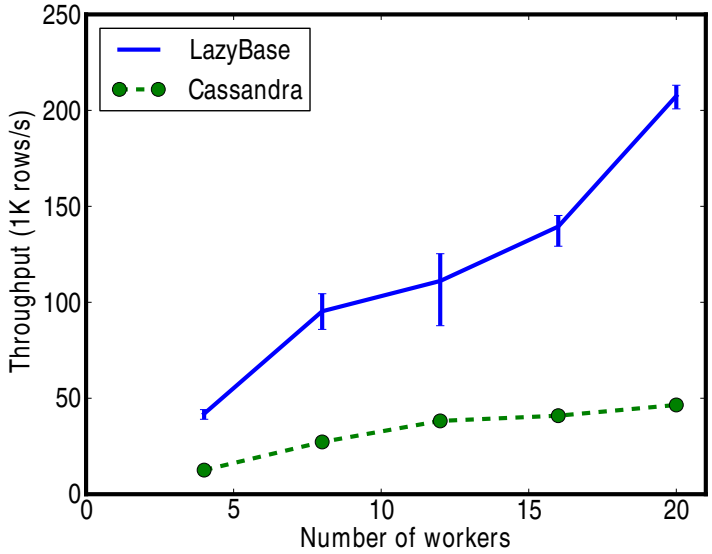


Figure 3.3: Inserts processed per second. Error bars represent the min and max of three runs, while the line plots the mean.

Update Scalability

To demonstrate LazyBase's update scalability, we compared equally sized LazyBase and Cassandra clusters. Figure 3.3 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.

3.4.3 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 3.5.

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 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 3.4 illustrates single-row query performance of both LazyBase and Cassandra. We exercise the query throughput of the two systems using increasing numbers of query

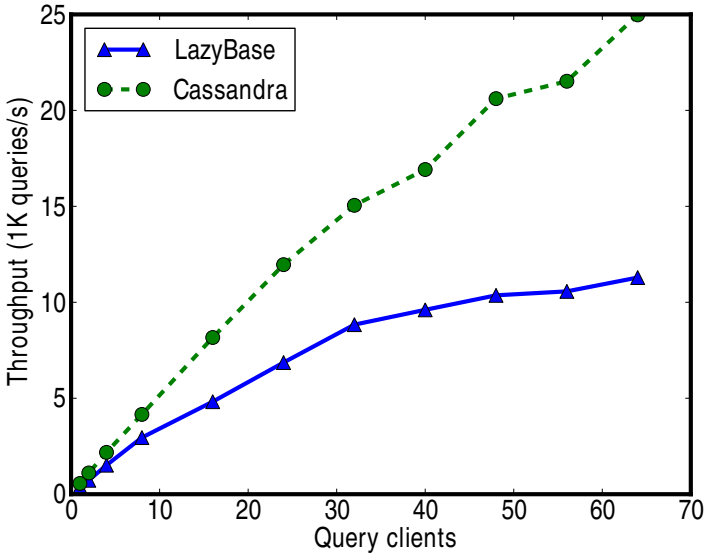


Figure 3.4: Random single row queries for LazyBase and Cassandra. Measured as queries per second with increasing numbers of query clients running 1000 queries per client.

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 3.5 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 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 3.6 illustrates the query latency when run with concurrent updates in LazyBase

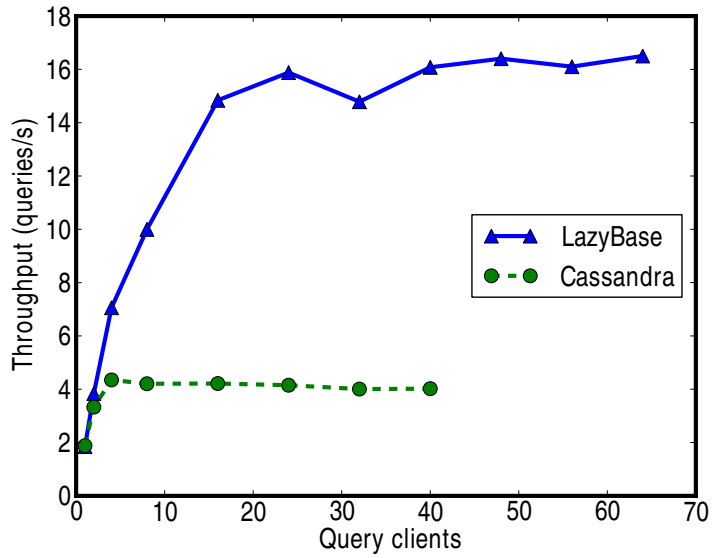


Figure 3.5: Random range queries for LazyBase and Cassandra with 0.1% selectivity. Measured as queries per second with increasing numbers of query clients running 10 queries per client.

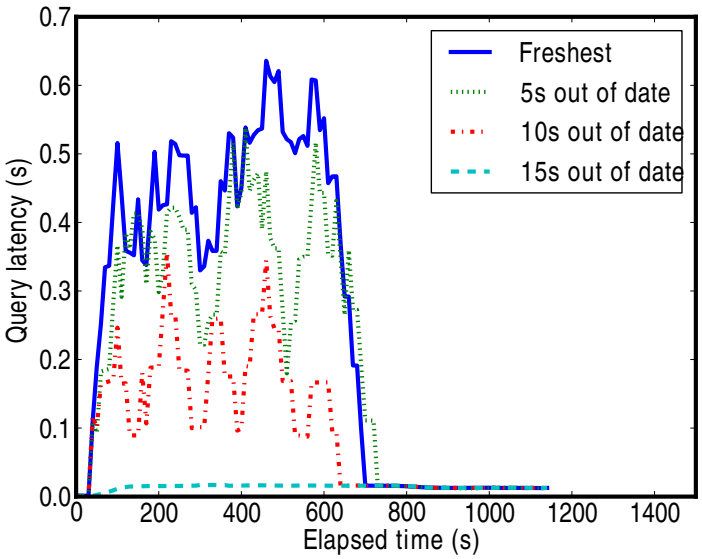


Figure 3.6: Query latency over time for steady-state workload. Sampled every 15s, and smoothed with a sliding window average over the past minute.

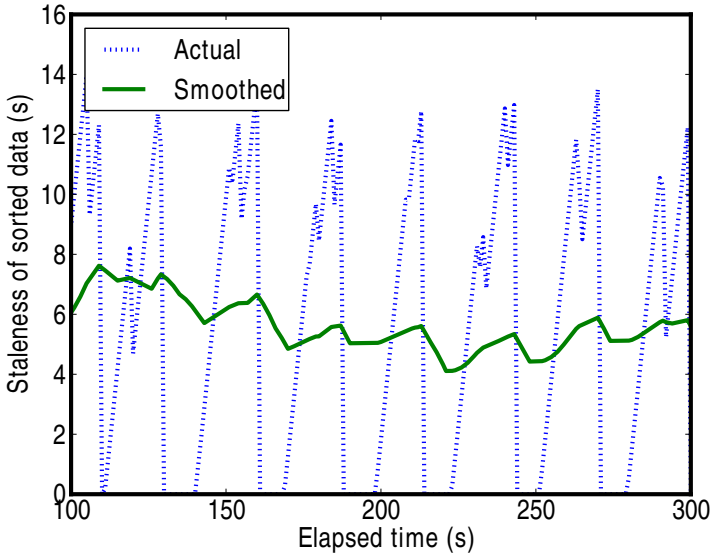


Figure 3.7: The staleness of the sorted data over a 200-second window of the steady-state workload. That is, at any point in time, the freshness limit that would have to be set to avoid expensive queries to unsorted data. The smoothed line is over a 60-second sliding window.

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 3.7 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 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 3.6 and 3.7 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 3.6 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.

3.4.4 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 3.8 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 3.9 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

[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.

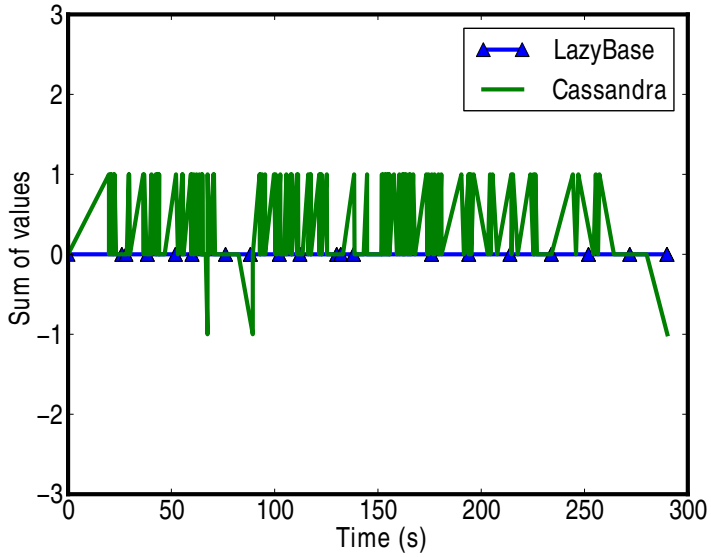


Figure 3.8: The sum of the values of two rows updated within a single client session. Non-zero values indicate an inconsistency in the view of the two rows.

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 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.

3.4.5 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

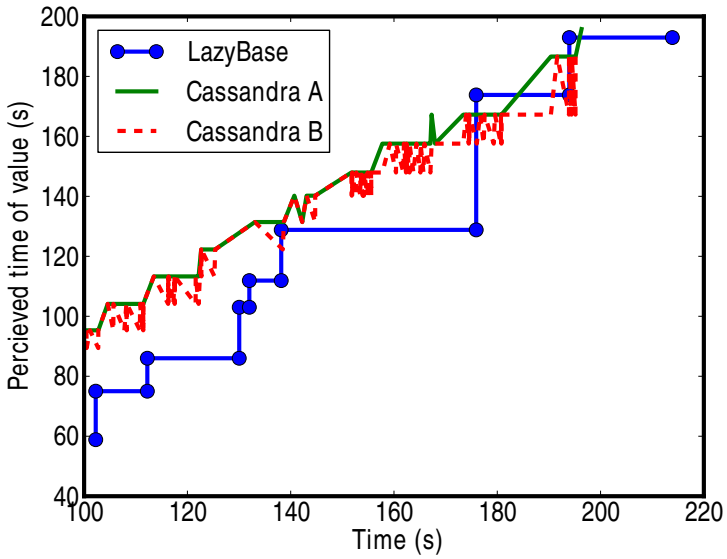


Figure 3.9: The effective timestamp of the returned rows at query time. Differences between Cassandra row A and Cassandra row B indicate inconsistencies between the rows at query time. Dips in the timestamps of a single row indicate violations in monotonic read consistency.

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.

Figure 3.10 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

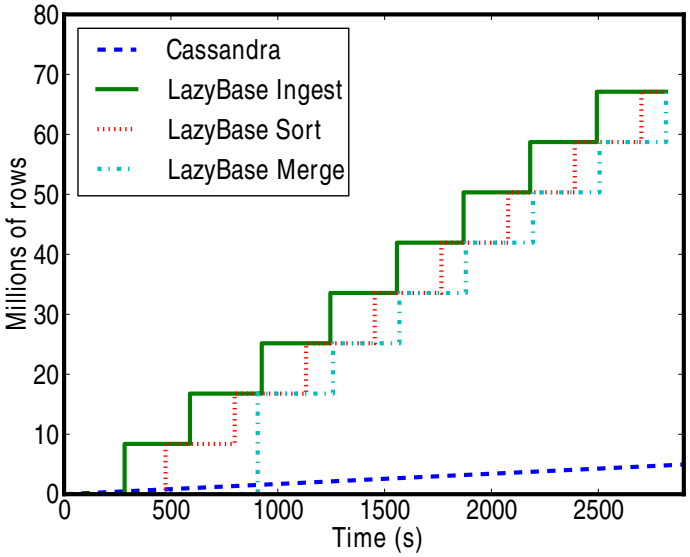


Figure 3.10: Ingest performance for the complex schema workload. The x-axis represents the elapsed time, and the y-axis represents the number of rows that are available in the database. For LazyBase each line represents the number of rows available at a particular stage of the pipeline.

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

Table 3.3: Performance of individual pipeline stages on the complex workload.

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 3.2. These results are

shown in Table 3.3. Compared to Table 3.2, 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.

3.4.6 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 analytics 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.

3.5 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.

3.6 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 [Hadoop] and Dremel [Melnik et al., 2010] 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. [Plattner, 2009, Schaffner et al., 2008]) have examined efficient means to support both data models from the same database and storage engine

using specialized in-memory layouts. RiTE [Thomsen et al., 2008] 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 [Vertica]. Both of these techniques provide up-to-date freshness, but require sufficient memory to cache the update stream. This becomes infeasible when the update workload experiences large bursts of activity that would overflow the in-memory portion of the database. Furthermore, in applications that require expensive queries to update materialized views from new data, it may be infeasible to perform all of the updates in memory. LazyBase separates the capture of the update stream from the transform step, reducing freshness but improving resource utilization and responsiveness to burst traffic. LazyBase's multiple sort orders are a solution to a subset of the materialized view problem. Different views of the data are created with different sort orders, and different subsets of the columns. Producing all of these, and merging with existing data, would not be feasible in an in-memory database.

FAS [Röhm et al., 2002] maintains a set of databases at different freshness levels, achieving a similar effect to LazyBase, but requiring significant data replication, reducing 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 [Adelberg et al., 1995, Salem et al., 2000]. Further, researchers have examined data structures to provide a spectrum between update and query performance [Graefe, 2004, 2006]. These structures provide a flexible trade-off 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 [Buttcher and Clarke, 2005, Lester et al., 2004]. 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 [Severance and Lohman, 1976]. More recent work for write-optimized databases limits queries to the base table, which is lazily updated [Hildenbrand, 2008]. Google's BigTable [Chang et al., 2006] 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 [Hastorun et al., 2007], SimpleDB [SimpleDB], SCADS [Armbrust et al., 2009], Cassandra [Cassandra], Greenplum [Greenplum] and HBase [HBase] 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 [Armbrust et al., 2009] 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 [Peng and Dabek, 2010], 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. [Sumbaly et al., 2012] 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 off-line. Unlike LazyBase, it does not permit queries to access the intermediate data for fresher results.

Some distributed "continuous query" systems, such as Flux [Shah et al., 2003] and River [Arpaci-Dusseau et al., 1999], 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.

3.7 Conclusions

We propose a new data storage system, LazyBase, that lets applications specify their *result freshness* requirements, allowing a dynamic trade-off 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.

4 LazyTables: Shared State for Parallel Machine Learning

LazyTables is a distributed *parameter server* for large-scale machine learning applications. It presents application programmers with a distributed shared data structure for storing the intermediate data ("parameters") for these algorithms. LazyTables maintains a set of freshness and consistency properties following the Stale *Synchronous Parallel model*, introduced in this chapter, providing application programmers a simple framework for controlling the trade-off between data freshness and performance. When the application instructs LazyTables that it can tolerate data staleness, LazyTables is able to increase its use of caching and operation logging to improve performance. However, when the application requires fresh data, LazyTables is able to provide it, bypassing mechanisms that may increase staleness.

4.1 Introduction

Machine learning (ML) algorithms are an important part of many applications, including document classification, movie recommendations, bioinformatics, and more. For instance, collaborative filtering algorithms are used to recommend movies, songs, and other products to users based on their previous taste, purchases, and browsing history. Sparse regression models are applied to genomes to identify the genes most likely to be responsible for certain traits (e.g., Alzheimer's). Page rank is used in search and social network analysis to find the most important items (web pages, users).

Many of these machine learning algorithms fall under the broad class of *iterative convergent algorithms*, which is the focus of this chapter. Table 4.1 provides examples of iterative convergent algorithms from machine learning, as well as some examples from outside ML.

As these applications become more ubiquitous, increasingly complex algorithms are being deployed on larger data sets, leading to performance problems. A state-of-the-art document topic modeling algorithm may take many hours to analyze a large corpus. For

Algorithm	Example applications
Latent Dirichlet Allocation (LDA)	News classification
Low-rank matrix factorization	Movie/music recommendations
Sparse regression	Genome-wide analysis
Conjugate gradient	Linear system solvers
Principal eigenvector	Web search/page rank
All-pairs shortest path	Mapping and route planning

Table 4.1: Examples of iterative convergent algorithms, and some of their applications.

instance, running single-threaded Latent Dirichlet Allocation [Blei et al., 2003] (LDA) over a corpus of 300,000 documents [Sandhaus, 2008] takes about 10 days [Newman et al., 2007].

To reduce computational time, application and algorithm designers are turning to parallel and distributed implementations running on clusters of servers [Newman et al., 2007]. While the diversity of these algorithms and applications makes it difficult to create a general-purpose method of parallelizing them, many of them share some important traits. This chapter focuses on *iterative convergent algorithms* – a class of algorithms that start with some guess as to the problem solution and proceed through multiple iterations that each improve this guess – with a particular focus on machine learning. The key property that makes this approach work is convergence, which allows such algorithms to find a good solution given any starting state.

Distributed implementations of iterative convergent algorithms tend to follow the Bulk Synchronous Parallel (BSP) computational model, described in detail in Section 4.2.2. In this model, all threads are always executing the same iteration at the same time, operating on a snapshot of the data produced by the previous iteration. While the BSP model can be easy to reason about, it is plagued by problems relating to communication overhead and straggler effects.

This chapter describes how staleness can be introduced into such systems in a controlled way to significantly improve the performance of distributed machine learning applications. This begins in Section 4.2 with a detailed description of iterative convergent algorithms and current techniques for running them in a distributed system. Section 4.3 introduces the Stale Synchronous Parallel computational model, which extends Bulk Synchronous Parallel with bounded staleness. Additionally, it describes Bounding Ball Staleness, a consistency model built on top of SSP that allows for more natural expressions of error bounds. Section 4.4 describes the design of LazyTables, a system that implements the SSP consistency model. Section 4.5 describes the example apps that are used in evaluating LazyTables, and Section 4.6 presents this evaluation. Lastly, Sections 4.7 and 4.8 discuss

open problems, and conclusions for the chapter.

4.2 Background

This section describes iterative convergent algorithms, the types of applications that use them, and current models for running these algorithms in parallel.

4.2.1 Iterative Convergent Algorithms

Iterative convergent algorithms typically search a space of potential solutions (e.g. N-dimensional vectors of real numbers) using an *objective function* that evaluates the goodness of a potential solution. Examples of objective functions include the mean squared error of an estimator, and the log-likelihood function in an inference problem. The goal of the algorithm is to find a solution with a large (or in the case of minimization, small) objective value. For some algorithms (e.g., eigenvector and shortest path), the objective function is not explicitly defined or evaluated. Rather, they continue to iterate until the solution does not change significantly from iteration to iteration.

These algorithms start with an initial state S_0 with some objective value $f(S_0)$. They proceed through a set of iterations, each one producing a new state S_{n+1} with a potentially improved solution (e.g. greater objective value $f(S_{n+1}) > f(S_n)$ in the case of a maximization problem). Eventually, they reach a stopping condition and output the best known state. Stopping conditions may include converging to a particular objective value, a slowed or stopped convergence rate, or simple conditions like running for fixed number of iterations or a fixed amount of real time.

A key property of these algorithms is that they will converge to a good state, even if there are minor errors in their intermediate calculations. This is because the algorithms will take any potential solution, and find an improved solution. As long as the errors are small enough, the improvement in each iteration will outweigh the regression due to errors, and there will still be a net improvement in the objective value. As we will see, this is what allows a system to introduce errors (via staleness) and maintain the correctness of the algorithm.

4.2.2 Bulk Synchronous Parallel

Iterative convergent algorithms are often parallelized with the Bulk Synchronous Parallel model (BSP), which I will use as a baseline when discussing other methods. As in the sequential version of the algorithm, BSP applications proceed through a series of iterations. In BSP, the algorithm state is stored in a shared data structure (often distributed among the threads) that all threads update during each iteration.

A single iteration of BSP consists of three steps. In the **computation** phase, threads produce the next value by operating on a snapshot of the previous iteration's value. In the **communication** phase, threads share this new value with each other, either by explicit communication or by writing to a shared data structure. Lastly, in the **synchronization** phase, threads execute a barrier to ensure that they don't begin the next computation step until all other threads have finished the communication step, and the new value is complete.

BSP provides a simple and easy-to-reason-about model for parallel computation, and can be easily applied to most iterative convergent algorithms. However, it introduces problems that can make it difficult to create efficient applications. Specifically, BSP is plagued by problems with communication overhead and straggler delays caused by "performance jitter" among threads.

4.2.3 Communication Overhead

Programs written using the Bulk Synchronous Parallel model must periodically stop computing to communicate updated values of all shared variables. In many applications, this communication time represents a significant portion of the overall execution time, thus reducing the efficiency of the application. Specifically, applications that tend to do very little computation before they have to share values – i.e. applications with many small iterations – may spend the bulk of their time in the communication phase.

Such applications, with many small iterations, are common in machine learning. Section 4.6 shows profiling results that separate computation time and communication time in a common ML algorithm. Depending on the configuration, this application may spend more than 80% of its time in communication and less than 20% of its time performing calculations.

4.2.4 Straggler Problem

Another problem with BSP is the *straggler problem*: because of the frequent and explicit synchronization, each iteration proceeds at the pace of the slowest thread in that iteration. The straggler problem refers to the phenomenon where a small number of slow threads have a severe impact on the entire computation. This problem only gets worse as the level of parallelism is increased. Because of seemingly random variations in execution time, as the number of threads increases, the probability that one of them will run unusually slowly in a given iteration increases. As a result, the entire application will be delayed in every iteration.

Stragglers can occur for a number of reasons including heterogeneity of hardware [Krevat et al., 2011], hardware failures [Ananthanarayanan et al., 2010], imbalanced data distribution among tasks, garbage collection in high-level languages, and even operating system effects [Beckman et al., 2006, Petrini et al., 2003]. Additionally, there are sometimes algorithmic reasons to introduce a straggler. Many algorithms periodically perform an expensive computation to evaluate their stopping criterion. Even if this computation is only run on a single thread, other threads will have to wait for it to finish before they can start the next iteration.

4.2.5 Existing Solutions

Existing systems address straggler problems in a number of ways. For consistent stragglers (e.g., less capable nodes), proper load distribution (e.g., via work stealing) or speculative execution (e.g., [Zaharia et al., 2008]) suffices. Speed variation due to unpredictable short-lived effects, such as intermittent "background" activities or contention for shared resources, is much more difficult.

The High Performance Computing community – who frequently run applications using the BSP model – has made much progress in eliminating stragglers caused by hardware or operating system effects [Dusseau et al., 1996, Ferreira et al., 2010, Petrini et al., 2003, Zajcew et al., 1993]. While these solutions are very effective at reducing "operating system jitter", they are not intended to solve the more general straggler problem. For instance, they are not applicable to programs written in garbage collected languages, nor do they handle algorithms that inherently cause stragglers during some iterations. Furthermore, the use of specialized hardware and software environments may not be applicable in all cases.

In large-scale networked systems, where variable node performance, unpredictable communication latencies and failures are the norm, researchers have explored relaxing

traditional barrier synchronization. For example, Albrecht, et al., describe partial barriers, which allow a fraction of threads to pass through a barrier by adapting the rate of entry and release from the barrier [Albrecht et al., 2006]. This is different from the Stale Synchronous Parallel model proposed here. Stale Synchronous Parallel puts a limit on the maximum difference in ”clocks” between two threads, while partial barriers allow a limited number of threads to fall arbitrarily far behind.

Another class of solutions attempts to reduce the need for synchronization by restricting the structure of the communication patterns. GraphLab [Low et al., 2010, 2012] programs structure the computation as a graph, where data can exist on nodes and edges. All communication occurs along the edges of this graph. If two nodes on the graph are sufficiently far apart they may be updated without synchronization. This model can significantly reduce synchronization in some cases. However, it requires the application programmer to specify the communication pattern explicitly. This may be difficult when the communication pattern is not known *a priori*, or changes over time. These techniques are also ineffective when the communication graph is very dense, and all threads need to communicate with all others.

Researchers have suggested making communication overhead a primary consideration in choosing which inference algorithm to use [Zhai et al., 2012]. They argue that, by using algorithms that perform a small number of long-running iterations, the overhead of communication and synchronization becomes negligible. While this does work in some cases, it often requires making other compromises so that a low-communication algorithm can be used. A systems-based solution to the communication problem gives application designers the flexibility to choose from a wider variety of inference algorithms.

Lastly, it is possible to ignore consistency and synchronization altogether, and rely on a best-effort model for updating shared data. Yahoo! LDA [Ahmed et al., 2012] and most solutions based around NoSQL databases rely on this model. While this approach can work well in some cases, it may require careful design to ensure that the algorithm is operating correctly. For instance, if a thread begins to fall too far behind other threads, the data the thread is operating over will be “undersampled”. In other words, that data will have less of an effect on the outcome of the computation, potentially causing a bias in the results.

Hogwild [Niu et al., 2011] is an extreme case of ignoring synchronization for shared memory systems. Threads do not lock shared data, and occasionally overwrite one another’s updates. The authors demonstrate that overwrites are infrequent enough that they do not hamper convergence. The Hogwild algorithm achieves the same solution quality with a 10x speedup relative to a lock-based approach. However, it assumes that all threads can sample from all data randomly, which may not be feasible in a distributed setting.

4.3 Using Staleness to Reduce Overhead

This section describes how data staleness can be used to reduce the communication and synchronization overhead of iterative convergent algorithms. Recall that in the context of this chapter, "data staleness" refers to staleness in the intermediate data that the algorithm is operating on, not to staleness of the input data. By introducing staleness, we can improve quantity of iterations – *iteration rate* – at the cost of decreasing the quality of the iterations – *convergence per iteration*. However, with the correct choice of staleness settings, overall convergence per time improves.

To further motivate the use of staleness for machine learning, consider a stochastic coordinate descent algorithm [Bradley et al., 2011]. This algorithm searches for a vector X that minimizes an objective function $f(X)$. The sequential version of the algorithm proceeds as follows:

1. given a set of parameters $X_1 ... X_n$, pick a coordinate i (i.e., an integer in the range $[1, n]$).
2. compute derivative of objective with respect to X_i: $\frac{df(X)}{dX_i}$.
3. update $X_i \leftarrow X_i - \gamma \frac{df(X)}{dX_i}$, where γ is a dampening factor between 0 and 1, used to adjust the step size.
4. repeat until converged.

In the Shotgun algorithm [Bradley et al., 2011] – a state-of-the-art parallel coordinate descent algorithm – each thread performs these operations independently, with no coordination or communication outside of updating the shared variable X_i. However, this shared variable must be updated very frequently, leading to very high communication costs.

Now suppose we introduce staleness into the Shotgun algorithm. Specifically, updates to the shared variable X_i may not be immediately visible to other threads. Instead, when a thread updates X_i, other threads may continue to see the old value for a number of iterations. By allowing this type of staleness, we can improve the iteration rate by reducing the need for otherwise frequent communication between threads. However, threads are now operating on (slightly) incorrect data, which means that it may take more iterations for the algorithm to converge. This exposes the trade-off between iteration quantity and quality, which will be explored in detail in this chapter.

Figure 4.1 depicts this trade-off graphically. Observe that increased staleness can significantly improve the iteration rate. However, there are diminishing returns: at some point, introducing more staleness provides very little improvement in iteration rate. Likewise, a

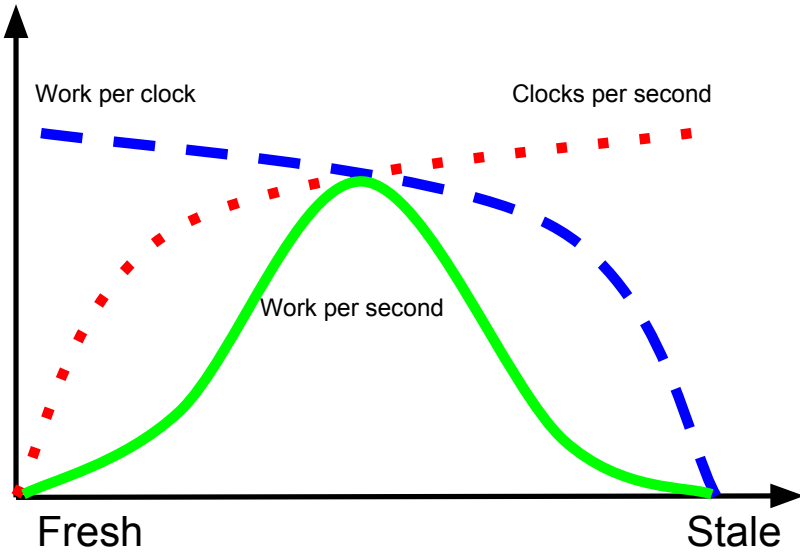


Figure 4.1: The trade-off between freshness and performance for iterative convergent algorithms. The blue dashed line represents the amount of useful work done in each clock period, i.e. the *quality* of iterations. The red dotted line represents the number of clock periods per second, i.e. the *quantity* of iterations. The green solid line is the combined effect: the amount of useful work done per second. At some point the improvement in clocks per second is balanced by the degradation in work per clock, and work per second is maximized.

little bit of staleness has little effect on the convergence per iteration, but too much staleness causes it to drop significantly. Generally, we observe that there is a sweet spot in staleness, where convergence over time – what we really care about – is maximized.

4.3.1 Minibatches and A-BSP

One approach to introduce staleness is to convert the algorithm to using the Bulk Synchronous Parallel model. Although not often discussed as such, BSP relies on the algorithm having a tolerance of staleness. During a given iteration, worker threads do not see the adjustments made by others; each of them determines these adjustments independently, and the adjustments are only aggregated at the end of the iteration. But, by coordinating only after a complete iteration, BSP reduces communication costs (by batching updates) and synchronization delays (by reducing their frequency). While most ML practitioners equate an iteration to a single pass over all input data, doing so fails to recognize staleness as a

parameter to be tuned for improved performance.

For applications using the Bulk Synchronous Parallel model, one can adjust the staleness of data by modifying the definition of an "iteration". To avoid a clash in terminology, the remainder of this chapter will use the word *clock period* (or simply "clock") to refer to an iteration of BSP, i.e. one round of compute-communicate-synchronize.

In the "textbook" version of BSP coordinate descent, each variable update must see all previous updates, so a clock can only include an update to a single coordinate. However, this would lead to unacceptably high overhead. Thus, in most parallel implementations, a standard "full batch" clock involves a complete pass over all data. Specifically, in "full batch" coordinate descent, an iteration is an update of all coordinates using the values from the previous clock period.

While this seems like a sensible way to define clock periods, it is likely that it does not fall on the optimal point in the staleness trade-off (Figure 4.1). If the data set being analyzed is very large, then a complete pass over the data involves a lot of computation and variable updates. This leads to a large degree of staleness compared to the sequential version of the algorithm, where updates are visible immediately. This corresponds to a point on the right side of Figure 4.1.

Similarly, with a small data set (relative to the number of processors) each iteration will take very little time, meaning that the ratio of computation to communication will be low. This means that the degree of staleness will be low. However, the relatively higher communication overhead will slow the iteration rate. This scenario corresponds to a point on the left side of Figure 4.1.

A common solution to the first problem (large data set, high staleness), is to start a new BSP clock after a partial pass over the data set – a technique known as *minibatches*. With, e.g, 10% minibatches, the first 10% of the data is processed, followed by a synchronization, then the next 10%, and so on. The increased frequency of synchronization allows the algorithm to use fresher data, at the cost of increased communication/synchronization overhead.

An approach to the second problem (small data set, low staleness) is to use *megabatches*, taking multiple passes over the data in a single clock period. This reduces the amount of communication, but increases staleness. For applications that started too far on the left of the freshness/staleness trade-off (Figure 4.1), this will improve convergence rate.

Together, these techniques are referred to as Arbitrarily-sized BSP (A-BSP) [Cui et al., 2014]. They provide a continuum of staleness settings within the framework of BSP, controllable by a parameter referred to as *work-per-clock* or *WPC*. While the unit of "work" depends on the particular algorithm, it is typically defined as a full pass over all input data

– what would be considered one iteration in the sequential version of the algorithm. For instance, in a coordinate descent algorithm, updating all coordinates is a single unit of work.

A-BSP provides a single parameter to control the use of minibatches and megabatches. However, this technique has a few shortcomings.

First, a static WPC setting is inflexible. The correct setting may depend on the progress of the algorithm. For some algorithms, staleness is more detrimental in later iterations. Section 4.6 shows examples of this, where a high staleness setting allows the algorithm to converge quickly at first, but it is unable to ”fine tune” the solution at the end. Some applications have the opposite property: they are more sensitive to stale results in early iterations. A goal of this work is to provide a framework that allows application developers to adjust the staleness settings dynamically. Heuristics for manually or automatically doing so is an area of future work.

Secondly, a programming framework designed to address staleness explicitly may be able to exploit it in various ways to improve convergence performance. Techniques such as caching, prefetching, and update batching can be implemented in such a system, exploiting specified levels of tolerable staleness. By designing the execution system with staleness in mind, it may be able to recognize opportunities to deliver fresher-than-necessary data with little to no communications overhead.

4.3.2 Stale Synchronous Parallel

To address the inefficiencies of BSP without giving up the benefits of synchronization, we propose a new computational model based on BSP, which we call *Stale Synchronous Parallel* (SSP). Like BSP, SSP assumes that the program consists of a number of threads, each proceeding through the same number of clock periods (iterations). During each clock every thread reads and updates some shared state. Updates can be simple overwrites (e.g. `x=5`) or accumulations (e.g. `x+=2`).

Unlike in BSP, programs using the SSP model must be aware of some crucial differences in the consistency model that can affect algorithm design as well as performance. These differences can be described in terms of the following properties [Terry, 2011b].

Bounded Staleness Data that is read by a thread may be more than one clock period of date. The application can put an upper bound on how stale the result of any given `read()` operation may be. In SSP, this bound is expressed in terms of the number of elapsed clock periods. This property is analogous to TACT’s order error [Yu and Vahdat, 2002].

Soft Synchronization When a thread reads a value, the bounded staleness property has the effect of a "soft barrier". Unlike a full barrier (as in BSP), which blocks until all threads are caught up, a soft barrier blocks the thread until all threads are within a specified range of the thread's current clock. For instance, a soft barrier with a parameter of "1" finishes when no thread is more than 1 clock behind the calling thread.

The Stale Synchronous Parallel computational model can be thought of as BSP with the addition of bounded staleness and soft synchronization. It is important to note that reads in SSP are not historical queries: the system *may* return fresher data than the specified bound. In fact, it may return fresher updates from some threads and stale updates from others. For instance, the system may incorporate updates from the current thread because they are inexpensive to provide: they require no communication. As a result, the actual amount of error introduced by staleness is typically less than the bound set by the application.

Providing updates from the current thread is inexpensive, and can often be useful for application programmers. A system that guarantees that these updates will be visible is said to provide the *read-my-writes* consistency property:

Read-My-Writes If a thread updates a value, all subsequent `read()` operations by that thread will see the update (unless it is overwritten by a later update). In other words, threads see their own updates *immediately*, even if updates from other threads may be delayed (staleness).

While read-my-writes is not necessary in the Stale Synchronous Parallel model, it may make the system easier to use for certain problems. Anecdotally, we have found that the read-my-writes guarantee made it much easier to translate algorithms to use SSP. For instance, the implementation of Page Rank with read-my-writes is a straightforward translation of the published algorithm. Determining how to implement PageRank without read-my-writes sparked a long running discussion between three researchers on the project. In the end, the only solution that would clearly work without read-my-writes for shared data involved using per-thread local memory for some data. This effectively provides read-my-writes consistency for a small set of private data. We did not *prove* that it is impossible to implement Page Rank without read-my-writes, but it is clearly more more difficult to design and reason about algorithms without that property.

Furthermore, it is often possible to implement read-my-writes with little overhead. In our implementation (described in Section 4.4) we guarantee read-my-writes consistency.

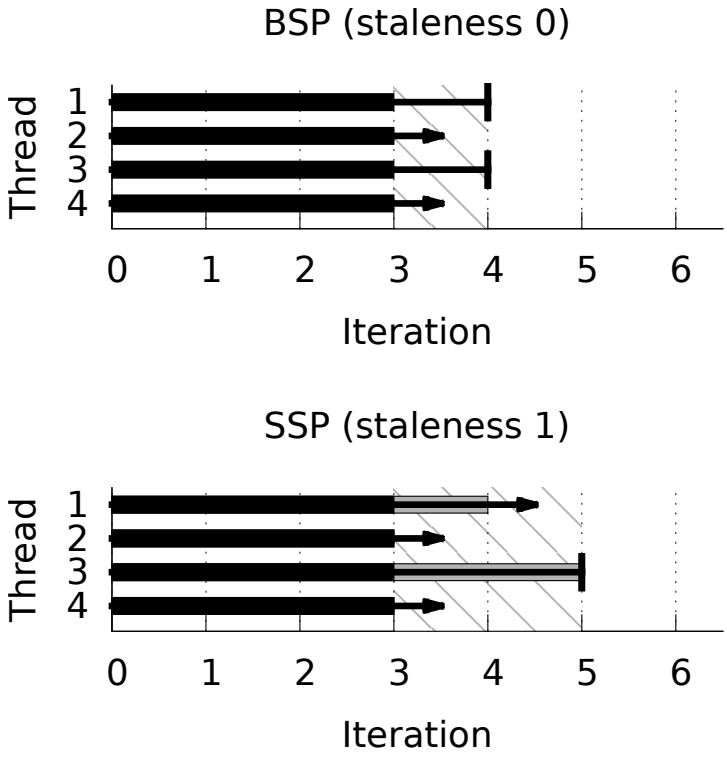


Figure 4.2: Diagram of Bulk Synchronous Parallel and Stale Synchronous Parallel execution state. The thick black bars indicate data that is visible to all threads. The gray bars indicate data that may be visible, but is not guaranteed. The lines indicate thread progress: arrows are runnable threads, while blocked threads are terminated with vertical lines.

Figure 4.2 illustrates these freshness properties. The BSP diagram represents an application with 4 threads using the BSP model. In this diagram, threads 2 and 4 are executing in iteration 3, while threads 1 and 3 are blocked waiting for them to finish that iteration. When these threads read data, they are guaranteed to see all updates up to the end of iteration 2, and none but their own in iteration 3.

The SSP diagram shows the same application, but using the SSP model with a fixed staleness of 1. In this diagram, threads 2 and 4 are still executing in iteration 3. However, because they are willing to use stale data, threads 1 and 3 did not have to wait for the other threads to complete that iteration. Thread 1 is currently executing in iteration 4. Thread 3 is blocked at the start of iteration 5 because it requires data from iteration 3 to continue.

The extra flexibility of the SSP model allows systems to reduce both the synchronization and communication overhead. Because of the soft-synchronization behavior, threads typically do not have to wait for other threads at the end of each iteration, eliminating the synchronization overhead. Furthermore, because threads can use stale data from other threads, they can communicate less frequently than they would in a BSP system.

4.3.3 Bounding Ball Staleness

So far we have defined staleness in terms of "iterations out of date". This model is easy to explain and reasonably simple to design systems around. However, tuning the desired freshness level may still be a burden for users. It is not always clear what a staleness of, for instance, 6 iterations actually means to an application.

Bounding Ball Staleness (BBS) is one potential solution for providing staleness bounds in terms of the numerical error, rather than number of iterations out-of-date. The result returned by a `read` operation in BBS is guaranteed to be within some bounding ball centered around the true value. The diameter of this bounding ball is tunable by the application designer. This provides a definition of staleness based on the geometry of the space being optimized.

One difficulty in BBS is defining what the "true" value should be. That is, where is the center of the ball? As Bulk Synchronous Parallel uses the previous iteration's value, we argue that the center should be the value that *will* be produced by the previous iteration, once all threads complete it. However, it's quite possible that an application would like to read a value before all threads have finished (or perhaps even started) the previous iteration.

Consider an application with 2 threads – A and B – that goes through a series of 5 iterations. For this example, assume all updates are increment operations, and let the updates done by each thread for each iteration be named A_1, B_1, A_2, B_2, etc.

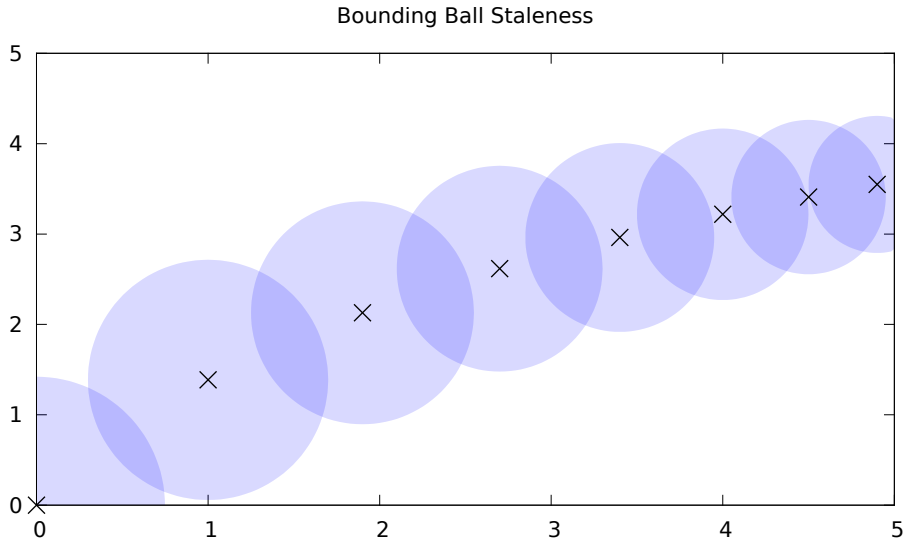


Figure 4.3: An example of an application running with Bounding Ball staleness. The x and y axes represent the optimization variables, i.e. the parameters of the space the application is exploring. The "X" markers represent the "true" snapshot locations at each iteration, starting from the lower left (0,0) point, and proceeding the upper right (5, 3.5) point. The shaded regions represent the bounding ball. If a thread is executing in iteration i and reads any parameter, the result of the read is guaranteed to fall withing the bounding ball surrounding that iteration's snapshot point.

No matter what the execution order is, we can define a series of *snapshots* $S_n = S_{n-1} + A_n + B_n$. In other words, S_n would be the shared state if both threads executed until iteration n, then stopped. In fact, this description of snapshots applies to ordinary SSP too. The bounding ball for a thread in iteration n is centered around S_n. Figure 4.3 shows an example of this.

However, we don't *actually* want to stop execution to find the snapshot state. Therefore the true snapshot value may not be known at the time. It may not even be knowable: some threads may not have executed as many iterations as others, so the values needed to compute the snapshot are not ready yet.

To solve this problem, we force threads to impose a limit on the *magnitude* of the updates that they submit in each iteration, for some distance function. Call this limit δ, thus $|A_i| \leq \delta$ and $|B_i| \leq \delta$. By limiting the magnitude of the update, we know that if thread

A is at iteration $n-2$, it can modify the global state by no more than 2δ before iteration n. Generally, given a vector clock V, the maximum difference between the known shared state, and snapshot n is $(\sum_i \delta \max(0, n - V_i))$. When threads adhere to this limit, there will be a direct relationship between SSP staleness and the bounding ball diameter. This limit can be thought of as another source of error for the algorithm. A thread *would have* taken a larger step, but could not because of the limit. However, limits like this are not unique to the BBS model. In general, gradient descent algorithms limit the step size to ensure convergence.

4.3.4 Implementing Bounding Ball Staleness

It is possible to implement BBS using a system that supports SSP as long as the application meets some requirements. Specifically, it's necessary to bound the magnitude of updates that each thread can make in any iteration. This bound could be an effect of the algorithm and data. For instance, in a coordinate descent algorithm, if we can determine analytically that the derivative with respect to any coordinate will never exceed d, and we adjust the step size by a factor of γ (with $0 \leq \gamma \leq 1$), then each thread will never submit an update greater than γd, so we can set $\delta = \gamma d$. Alternatively, an application designer can force a bound by artificially decreasing the step size for any iteration that would have exceeded the bound.

Given a per-thread bound of δ, and a system with t threads, the total magnitude of an update in any iteration cannot exceed $t\delta$. With a staleness of s, the size of the bounding ball is limited to $ts\delta$. Thus, any system that supports bounded staleness can also support bounding ball staleness.

However, when implementing BBS on top of SSP, this bound may be much looser than necessary, especially in systems with many threads. The reason for this is that staleness in SSP is measured as the difference between a current thread's clock and the *slowest* thread's clock. In other words, the size of the bounding ball is $(C_i - \min(C))t\delta$, where C is a vector of each thread's clocks, and C_i is the current thread's clock. However, if each thread's clock is known, a much tighter bound can be computed: $\sum_j abs(C_i - C_j)t\delta$.

Based on this result, an SSP system can be modified to support BBS more effectively. The BBS model imposes a limit on the total number of missing updates, rather than the number of missing updates from any particular thread. Therefore, rather than waiting for every thread to reach a given iteration, the system simply has to wait until the total iteration count (the sum of the vector clock) reaches a particular value. In this way, BBS is similar to the partial barriers described in [Albrecht et al., 2006].

4.4 LazyTables Design

We have built a prototype system called LazyTables that implements the Stale Synchronous Parallel model to support distributed machine learning applications. LazyTables provides the abstraction of a set of shared sparse matrices (tables) that all processes access as a *parameter server* [Power and Li, 2010, Smola and Narayanamurthy, 2010, Ahmed et al., 2012, Li et al., 2014] to store intermediate results. These matrices are stored in memory, distributed across a set of servers. Typically these server processes reside on the same physical hardware as the processes running the application, but this is not a requirement.

LazyTables is a form of distributed data structure (as opposed to general purpose distributed shared memory). It provides the programmer with the illusion of a single data structure shared between processes running across a cluster of machines. Specifically LazyTables appears to be a multilevel hash map from integer row indexes to rows, where each row is a map from integer column indexes to the floating point values used by the application (in C++ parlance: `map<int, map<int, double> >`), along with special operations for reading entire rows, incrementing values, and notifying the system that the thread has completed an iteration.

LazyTables provides an API similar to the Piccolo [Power and Li, 2010] system. It provides read and update operations including `get()`, `get_row()`, `put()` and `increment()`. While Piccolo provides support for a generic `update()` operation, LazyTables currently supports only `increment()`, which is sufficient for our test cases. Generic updates are necessary for some other algorithms such as all-pairs shortest-path, which uses `min` as the update operator. We do not see this as a design limitation: the LazyTables design can support any update operation that is commutative and associative with itself.

Non-commutative update operations could be supported, but would be another source of staleness errors: an update could be applied to a different value than what was intended. When the operation is not commutative, the result will be different. This problem is not unique to LazyTables. Any distributed system without transactional updates, including Piccolo, will have the same problem with non-commutative update operations. However, staleness may exacerbate the problem, since a thread may be trying to update a very out-of-date value.

Applications use LazyTables as a parameter server, to store the intermediate results of the computation. A typical application proceeds in three phases:

Loading Loading data from stable storage (e.g., file system or database) into LazyTables.

Iteration Iterating over data, modifying data in LazyTables with many small updates.

Output Writing output (subset of data in LazyTables) back to the file system.

The bulk of the application's time is spent in the iteration phase. Therefore, LazyTables is designed to make this as efficient as possible. The iteration phase for many machine learning applications is characterized by few row reads and many small updates (e.g., incrementing a single location in a matrix). The calculations on each data item are often very simple, such as drawing a random number from a distribution parameterized by the data. Therefore, the performance of the application is dependent on the performance of these table operations.

4.4.1 High-Level Design

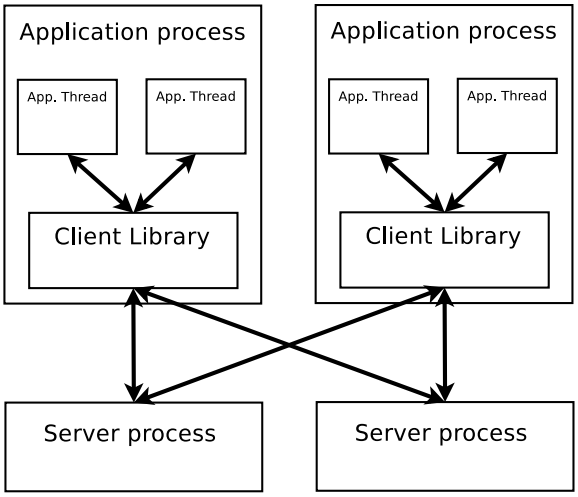


Figure 4.4: Component diagram of LazyTables running with two application processes, running two threads each. The parameter server is similarly split between two distributed processes. Each application thread calls its local Client Library, which forwards requests to the appropriate server.

LazyTables is composed of two principle components: a *client library* and a cluster of *parameter server* processes. This section will describe the high-level interactions between these components, while sections 4.4.2 and 4.4.3 will describe the individual components in detail.

Figure 4.4 depicts the relationship between client library instances and parameter server processes. The figure depicts two application processes each running two application threads. These four threads communicate with each other by accessing shared parameter

server data. They do this through the local instance of the client library.

Client Library

The client library runs in the same process as the application code and is responsible for maintaining caches and operation logs, as well as providing a simple interface to the application programmer. LazyTables is designed to primarily support applications that use one process per machine, with multiple worker threads in each process. As such, the client library is designed to handle multiple threads efficiently.

The client library translates the client-facing API calls to server commands. This involves determining which server is responsible for storing the relevant data, sending requests to the server, and handling responses. The client library also uses caching and update batching to improve efficiency without violating staleness bounds.

Parameter Servers

The cluster of parameter servers is made up of identical processes that are responsible for fulfilling read and write requests on the underlying data. Data is distributed among the servers based on row number so that all data for a single row resides on the same server.

For simplicity, the system is designed so that server processes never communicate with one another; in fact, they do not need to know how many other servers there are. Instead, when a client updates a value on a server, the server always accepts the update. Similarly, the server will respond to any read request: if the data is not present at the server it simply responds with an empty row. This design can support atomic updates to a single row, but update operations that involve multiple rows are not atomic.

4.4.2 Client

The LazyTables client library seeks to provide efficient access to the shared tables while maintaining the consistency properties of SSP. To do this, it includes a set of caches and logs. The major components of the client library are depicted in Figure 4.5.

To understand the interactions between the client library components, I will first describe a simplified implementation of the client library that consists of only the communication thread, the process cache (simply "the cache" for this example), and the process log (simply "the log" in this example). Then, I will describe the other components and how they improve the system.

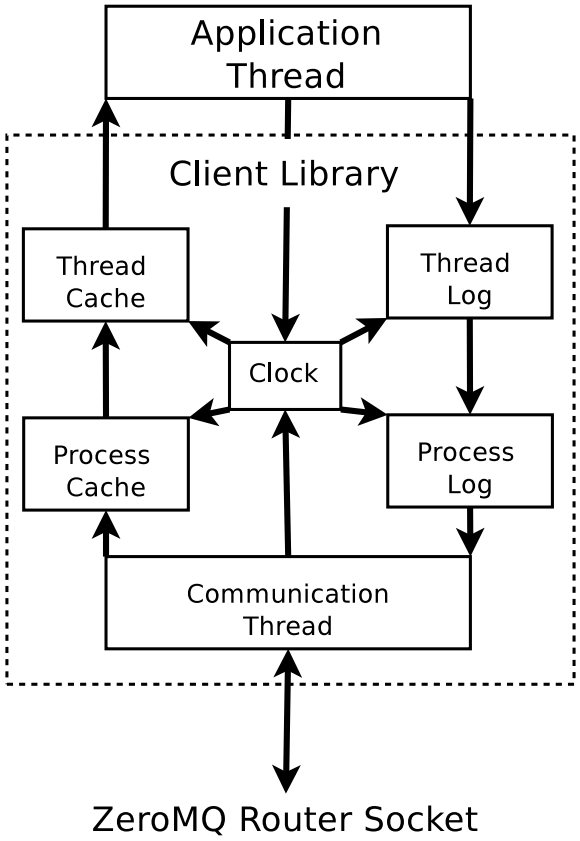


Figure 4.5: Detailed system diagram of the LazyTables client library. Arrows represent data flow: data produced by the client is sent through the logs to the communication thread and finally the network. Data read from the network passes through a series of caches and finally back up to the application threads. The clock is updated by both the application threads and the network, and controls the other data movement.

Communication Thread

As its name implies, the *communication thread* is a separate thread responsible for all network communications. It receives requests (such as `read_row`) from other threads in the process, encodes them as messages, and sends them to the appropriate server. It also receives messages from the servers, decodes them, and handles them appropriately.

When a `read_row` response, which is a message containing row data and a timestamp, arrives from the server, the communication thread is responsible for updating the cache. After updating the cache, all waiting threads are notified (via a condition variable) that the

cache has been updated.

Process Cache

The cache is responsible for handling `read` and `read_row` requests from the application, reducing the number of operations that require communication with the server. It caches entire rows and their respective timestamps. When a worker thread requests data for the row by calling a client library function, the cache timestamp is checked against the current iteration number and staleness limit for the request. If the cached data is fresh enough, it is returned. If it is not fresh enough, the request is forwarded to the communication thread, and the requesting worker thread waits for the cache to be updated.

Process Log

While the cache is responsible for reducing communication overhead for read requests, the log is responsible for reducing communication overhead for write requests. When the application calls a `put` or `update` operation, the operation and its parameters are logged in the process log. Later, when the application calls `clock` to advance its clock, the log sends all updates to the communication thread. These updates are sent to the servers as a single message per server.

An important job of the log is to coalesce similar updates to reduce the total amount of communication and the amount of work that the servers must do. For instance, if the application calls `increment` twice on the same data item from the same worker thread, or from different ones, these two operations can be converted to a single `increment`. Similarly, when the application calls `put`, any previous operations for that location can be deleted from the log, as they would be overwritten by the `put`.

Clock

The clock is responsible for keeping the client's view of the current iteration. It receives updates from the worker threads and the communication thread, and it propagates these updates to the other components as needed. Updates from the clock trigger action by the other components, such as flushing logs or communicating with servers. Internally, the clock keeps a few different notions of time:

Thread Clock The thread clock is a vector clock representing the current iteration for each worker thread in the process. It is incremented when a thread calls the `clock()` method. Incrementing the thread clock flushes the thread log to the process log.

Process Clock The process clock tracks the minimum value of all of the thread clocks. It is incremented when the last thread in an iteration calls `clock()`. Incrementing

the process clock causes the process to send a `clock` message to the server cluster, and flush pending updates created in previous iterations from the process log to the server cluster (by way of the communication thread).

Global Clock The global clock tracks the minimum process clock across the entire application. It is incremented when a `clock` message is sent from the server cluster to the application. This clock is used when determining which cache values are out-of-date relative to an operation's freshness requirement.

Thread-Local Storage

The process log and process cache are shared by all threads within an application process. While this is often acceptable for performance, there are cases where it is inefficient. Some applications require such frequent reads and writes that synchronization on the shared cache has a significant impact on performance. To overcome this, we introduce thread-specific caches and logs. These behave like the process cache and process log, except that they are local to each thread. Because of this, they do not require any synchronization on reads and writes.

Prefetcher

The Prefetcher (not depicted in the figure) tries to improve application performance by preemptively issuing `read` requests to the server cluster. By issuing a `read` request before the row is needed, the application can avoid waiting for data when it does need the row. There are many ways to determine when to prefetch a row. Two example strategies are outlined in Section 4.4.4.

4.4.3 Parameter Server Cluster

The main data structure in the server is a set of *tables*, which store rows of data. Additionally, the server cluster keeps queues of pending reads and writes, and a clock. These components are depicted in Figure 4.6 and are described in detail in this section.

Table

The tables store the current "master" view of all rows that are stored on this server. Read requests are serviced from this data. Each table is given an ID number that is used by the client library to specify which table an operation should affect. Additionally, tables are given text-based names by the application programmer. The programmer uses these names when initializing the client library, allowing them to appear as named variables. For

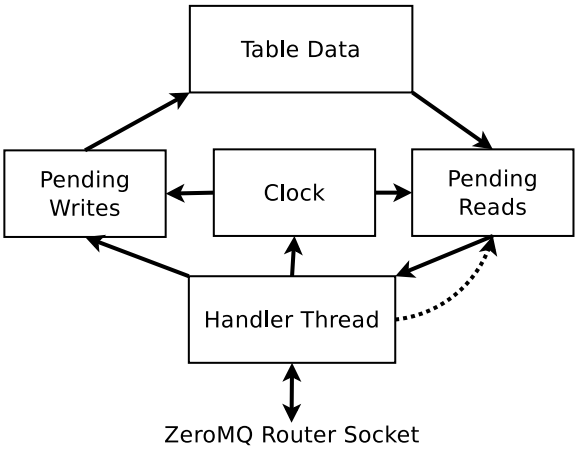


Figure 4.6: Detailed system diagram of the LazyTables server process. The dotted arrow represents the movement of read requests to the pending reads queue. All other arrows correspond to data movement: data is written through the pending writes queue, to the table, and read through the pending reads queue.

instance, the following code is used to get a handle to the topic-word table:

```
LazyTable tw_table = client_lib.table("topic_word_table");
```

Clock

The clock provides a similar service as the clocks in the client library. It is comprised of two parts: a vector of process clocks and a single global clock. The process clocks track the clock value of every client process that is connected to the server. When the client library sends a `clock` message, it is updated in this vector clock.

The global clock tracks the minimum value of the process clocks. When the global clock changes, it sends a `clock` message to all clients, moves updates from the pending writes queue to the table, and responds to the pending reads that are now serviceable.

Pending Writes

The pending writes queue is not necessary for the SSP model, but we use it both to simplify mimicry of BSP and to simplify the implementation of read-my-writes. It can be discarded when neither is needed, and could be replaced with a per-thread strategy in a future implementation.

When it is used, updates from future iterations (iterations beyond the current global clock) are queued in the pending writes queue. When the clock advances, those pending

updates are written to the table, but updates from later iterations remain in the queue. In this way, no updates are reflected in the global table until all threads finish an iteration. This simplifies the implementation of read-my-writes by making it easier to track which updates have been applied to the global table, and which have not. It allows mimicry of BSP, because writes in BSP cannot be visible until all threads have finished an iteration.

Pending Reads

The pending reads queue keeps track of read requests that could not be immediately serviced. This occurs when the staleness bound of the request is beyond the current global clock value. In other words, when a thread has advanced more than s iterations beyond the slowest thread and is requesting data with a staleness bound of s.

Snapshots and Fault Tolerance

It is possible to configure the LazyTables parameter servers to take periodic snapshots of their state. These snapshots are taken at fixed clock period intervals, e.g. every 10 clocks. That ensures that all servers will take the snapshot at the same virtual time, allowing clients to determine which updates are included in the snapshot.

In our current prototype, the main use of snapshots is to enable performance experiments. Determining the global objective value can be a computationally expensive operation that we do not want to run in parallel with a performance test. Snapshotting is relatively lightweight, and allows us to determine objective values after the experiment has finished.

Despite the fact that we mainly use it for experimental purposes, the snapshotting mechanism could be used as the basis for a fault tolerance system for LazyTables. In the event of a failure, the servers could reload their state from the last complete snapshot, and clients could be instructed to begin processing from a later iteration. Currently there is no mechanism for loading snapshots in the servers, but it would not be difficult to implement. However, the client side changes may be application-specific: If an application uses local memory in addition to LazyTables, it will be responsible for recording that state at snapshot boundaries, and reloading it when restoring a snapshot. Clients that do no store crucial state in local memory would not have such problems.

4.4.4 Design Trade-Offs

The design of a parameter server implementing the SSP model involves a number of trade-offs. The goal of LazyTables is demonstrate the utility of SSP, as such, exploring these trade-offs in detail is outside the scope of this work. Nevertheless, it is useful to discuss

some of the issues that arise in designing such a system.

Read-My-Writes

The read-my-writes consistency property says that when a client reads data, it will always see the result of its own previous updates, provided that they have not been overwritten by other updates. This is often one of a number of properties grouped together as *session guarantees*. In many applications, the read-my-writes property makes it much easier to reason about how to write the application. Based on that insight, we designed LazyTables to support read-my-writes. This decision carries with it a number of complications.

The main difficulty in implementing read-my-writes is that there is no guarantee that an update sent to the server will be reflected in the global table immediately. In SSP, this delay could be as long as the staleness limit for the read operation. To implement read-my-writes, the client library must retain information from the update log after the log has been sent to the server. When data is read from the server, the library must selectively reapply the logged updates to the data. But, the client must not reapply updates that the server has already applied.

This requirement is the reason that the server keeps a pending writes queue. Writes are queued and marked with the clock of the thread that submitted them. They are only applied to the global table once the global clock matches the pending write clock.

By doing this, the server can guarantee that a row contains all data up to the global clock, and no more. The client then knows which updates need to be reapplied. Without this design, the client would need to keep separate per-thread write logs, as well as vector timestamps on each row.

However, by adopting this design, the data returned to the client is not as fresh as it could be. In fact, it is almost as stale as it could possibly be.

Another option would be to forgo read-my-writes. Such a system would be substantially simpler, eliminating a major component of the server process, as well a source of complication for the client update logs. Furthermore, it would be able to deliver *fresher* data to the clients. However, some algorithms would be more complicated to implement. For instance, a simple implementation of PageRank requires read-my-writes to properly track the node weights. Without read-my-writes, the algorithm has to be modified to ensure correctness. Specifically, threads have to use local memory to track changes they have made, effectively providing read-my-writes for the subset of the data they are responsible for.

Prefetching

Another design decision is the use of prefetching for rows. Prefetching can be automatic, with no input from the application programmer, or application-guided, where the application provides hints to control the prefetching behavior. A basic automatic prefetching heuristic is described below. However, many machine learning algorithms have a predictable structure. At the beginning of every iteration, it is possible to predict which rows will be needed. This information can be used in an *application-guided* prefetching system.

The trade-off between automatic prefetching and application-guided prefetching is one of ease-of-use vs. efficiency. Early LazyTables experiments tried a number of automatic prefetching heuristics. Most of them actually degraded performance: the extra overhead caused by the prefetching requests outweighed the benefit of having fresher data. This becomes especially true in cases where the current `read` operation is not a good predictor of future reads.

Automatic Prefetching

This section outlines a simple automatic prefetching heuristic that we have implemented in LazyTables. Prefetching is triggered when the application issues a `read_row` command. If the row is not in cache, or the cached row is too old, an ordinary `read_row()` request is sent to the server. If the row is in cache, the prefetcher may issue a background `refresh_row` command to the server, while returning row data to the application immediately. A `refresh_row()` request is sent if all of the following conditions are met:

- We do not have an outstanding request for this row, either `refresh` or `read`.
- The total number of outstanding requests is below `REQUEST_LIMIT`.
- The cached data is "getting old". Specifically, $staleness\_limit - cache\_age <$ `PREFETCH_STALENESS_LIMIT`.

The parameter `REQUEST_LIMIT` simply prevents the prefetching system from flooding the servers with requests. By limiting the total number of outstanding requests, the prefetcher can limit its impact on performance. However, if the `REQUEST_LIMIT` is too low, prefetching will be ineffective.

The `PREFETCH_STALENESS_LIMIT` controls the "aggressiveness" of prefetching. If set it to 1 it means "only prefetch if we're going to have to do a blocking read next iteration". If set it to `MAX_INT`, it means "always prefetch (if all other conditions are met)". The `PREFETCH_STALENESS_LIMIT` parameter also limits prefetching requests, increasing the chances that "important" requests – those for which the current cache is very stale – are executed.

Application-Guided Prefetching

Automatic prefetching removes the burden of prefetching decisions from the programmer, but may make poor decisions if assumptions about temporal locality are wrong. Fortunately, many machine learning algorithms exhibit regular and predictable access patterns that can be exposed to the prefetcher, to guide the prefetching policy. Typically, a thread will process the same set of input data in each iteration. In some cases, such as the matrix factorization example, the rows used by a thread are completely determined by the input data. A thread can predict its access pattern completely, and pass all of this information to the prefetcher. In other cases, such as topic modeling, some of the rows can be predicted at the start of an iteration, but others depend on the results of the iteration's computation. In this case, application-guided prefetching can only prefetch some of the rows that will be needed.

To support application-guided prefetching, the LazyTables API exposes the `refresh` operation to application programmers. The parameters to `refresh` are the same as those for `read`, except that `refresh` never blocks and never returns results. The behavior of `refresh` depends on the specific prefetching policy in use. LazyTables currently supports two policies, conservative and aggressive. In *conservative prefetching*, a row is only refreshed when necessary, given the staleness bound: if $cache\_age < timestamp - staleness - 1$. With *aggressive prefetching*, a row is always refreshed unless it is completely up-to-date.

The choice between conservative and aggressive prefetching is one of resource use vs freshness. Conservative prefetching uses less resources, both on the client and server side, but will provide staler data to the application. We find that the extra cost of aggressive prefetching is almost always worth it.

Thread Caches and Logs

By using thread-local storage, the client library can avoid synchronization operations – mutexes and condition variables – on read and update operations from the application. However, these benefits come at a cost. First, there are the extra memory requirements. For large working sets, the extra thread caches can take up a lot of precious memory space. If the cache is too small – below the working set size – it will be ineffective. Too many operations will have to access the shared process cache.

Another issue is the overhead of moving data between the different layers of caches and logs. In applications where a row is used many times by the same thread, this overhead is amortized over many operations. However, other applications access rows in a random pattern. In these cases, a row is typically accessed once before being evicted from the thread cache. This means that the overhead of moving data to the thread cache is wasted.

4.5 Example Applications

The design of LazyTables and the implementation of the prototype were motivated by the needs of three example machine learning applications. To demonstrate the generality of LazyTables, these applications are drawn from different fields of machine learning, and they employ different algorithms representing three of the main machine learning approaches.

4.5.1 LDA: Topic Modeling

The first application is Topic Modeling. Generally, topic modeling is a class of problems that assign *topic vectors* to documents. The topic vector represents how well a document is represented by a topic. For instance, a document about a financial scandal may be classified as "50% politics, 30% finance, 20% legal", while another document about the professional baseball draft may be "80% sports, 20% finance".

The specific model that the test case uses is known as Latent Dirichlet Allocation [Blei et al., 2003] (LDA). LDA is a generative model. It assumes that each document can be summarized as a multinomial distribution over topics. Similarly, each topic is simply a multinomial distribution over words. So, to generate a document, words can be generated one-by-one: first a topic is chosen from the document's topic distribution, then a word is chosen from the topic's word distribution. This procedure continues, generating an infinite stream of words for each document.

The goal of LDA is to learn the parameters of the document-topic and topic-word distributions that best explain the data (a corpus of documents). While there are a number of ways to do this, the example application uses collapsed Gibbs sampling [Ahmed et al., 2012], a Monte Carlo technique.

Gibbs sampling is a technique to sample from a multivariate probability distribution in cases where direct sampling is difficult. It requires that the algorithm designer specify how to sample from any conditional distribution. That is, given fixed values for all variables except one, sample from the distribution of the remaining variable. Gibbs sampling iterates through all variables, updating the current value of each variable by sampling in this way. In a parallel iteration, different variables can be sampled simultaneously by different threads.

In the case of LDA, the fixed parameters are the topic assignments for all words in all documents, except for the single word that is being examined. With all other word assignments fixed, the current document-topic probabilities and topic-word probabilities are known. These are used to generate a random topic for the current word

4.5.2 Matrix Factorization: Collaborative Filtering

The second application is an example of collaborative filtering. Collaborative filtering attempts to predict user's preferences based on the known preferences of similar users. A familiar example of this is found in online movie services, where users rate movies they have seen. The service uses these ratings to predict the users' ratings for other movies. These predictions are used to present the user with a personalized list of movie recommendations.

One technique to perform such recommendations is by low-rank matrix factorization. In the movie example, this technique assumes that there is a large matrix, the *user-movie matrix*, with a row for each user and a column for each movie. The values in the matrix represent the users' ratings for each movie. However, only some values are known, and our goal is to fill in the rest of the matrix.

This is done by assuming that the matrix has low rank. That is, it is the product of a tall and skinny matrix and a short wide matrix. Intuitively, the tall and skinny matrix can be thought of as the *user-genre matrix*, where each row represents a user, and each column represents how much that user likes that genre of movies. Similarly, the short wide matrix is the *genre-movie matrix*, representing how much fans of a given genre like a particular movie. [1]

Once this factorization is found, any rating can be predicted by simply computing the dot product of the user's row in the user-genre matrix, and the movie's column in the genre-movie matrix.

The (incomplete) user-movie matrix is factored into the user-genre and genre-movie matrices by a stochastic gradient descent algorithm [Gemulla et al., 2011]. This algorithm iterates over known values in the user-genre matrix, performing small adjustments to user-genre and genre-movie matrices for each value.

4.5.3 Sparse Regression: Genomics

The last application is an algorithm used to determine which genes are responsible for some trait of interest, e.g. Celiac disease. It uses a sparse regression model for genome-wide analysis. This problem involves a large number of potential predictive variables (e.g. nucleotides or pairs of nucleotides), but a relatively small number of samples. In linear algebra terms, the problem is *underdetermined*, meaning that there are many solutions.

[1] There is a difference between "how much fans of a genre like the movie" and "how much the movie belongs to the genre". A movie could be, without a doubt, an action movie. However, if it is a bad action movie, action fans will still dislike it.

However, geneticists know that these solutions are not equally good. Specifically, it is most likely that a trait is affected by a small number of genes. In other words, the solution should be sparse: most coefficients should be 0.

To find sparse solutions to these problems, the application uses a least squares objective function with a Lasso regularizer [Bradley et al., 2011]. Lasso is a general term for the technique of penalizing the objective function with the $L1$ norm of the optimization variable. Doing so encourages sparse solutions. The regularization term has a coefficient that can be used to increase or decrease the penalty, thus increasing or decreasing the preference for sparse solutions.

After constructing the objective function, the application uses a standard gradient descent algorithm to find the most predictive variables.

4.6 Evaluation

This section evaluates the Stale Synchronous Parallel model using our LazyTables prototype. Results demonstrate that systems using A-BSP or SSP are more efficient than strict BSP. Specifically:

1. Introducing staleness (via A-BSP or SSP) controls a trade-off between iteration speed (efficiency) and iteration effectiveness.
2. The best point on this trade-off is not always the one used by the default BSP setup, and depends on factors including the algorithm being parallelized and the hardware setup.
3. The flexible nature of SSP is better suited to handling intermittent disruptions in performance (temporary stragglers).

Additionally, the results include an exploration of the overheads introduced by – and mitigated by – staleness. Specifically the CPU overhead of extra communication and the reduction of network wait times.

4.6.1 Implementation Details

In these experiments, we used the LazyTables implementation for BSP, A-BSP, and SSP. Note that SSP with staleness=0 is equivalent to BSP, so the same implementation can test BSP, A-BSP and SSP. Additionally there are results for an asynchronous system based on the same caching and communication architecture as LazyTables, but configured so that

clients never wait for fresher updates, and so that the server sends updates to the client whenever they are available.

The LazyTables prototype is written in C++, using ZeroMQ [Hintjens, 2010] for asynchronous communication. ZeroMQ was chosen because it provides a flexible low latency communication layer, and, importantly, will automatically batch messages that are sent to the same destination into larger frames. It uses an opportunistic batching mechanism. All pending messages are sent at once, but it will not wait for more messages to be added to the queue. This improves the performance of the system when many small messages are being sent at once.

Data, on both the servers and the client caches, is stored in RAM using C++ container objects. Data can be stored sparsely using Boost `unordered_map` types (a hash table), or densely with STL `vector` types. Data is transmitted in a densely packed format using ZeroMQ multipart messages.

This section includes results from multiple studies with different experimental setups. Relevant details of the hardware and setup will be provided with the results.

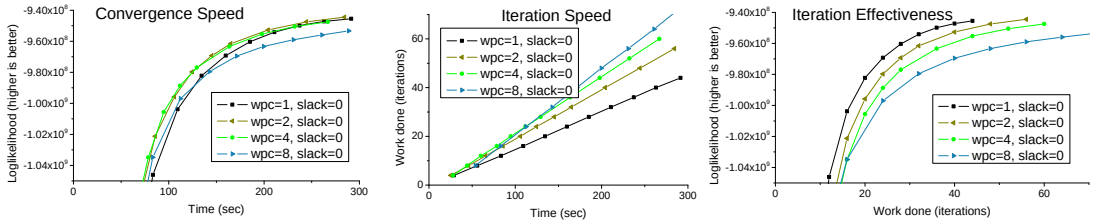
4.6.2 Exploiting Staleness for Efficiency

The first set of experiments (Figure 4.7) shows how the work-per-clock (WPC) and slack parameters affect the iteration rate (quantity) and iteration quality. WPC and slack control the staleness of A-BSP and SSP, respectively. Together, these experiments are meant to demonstrate points 1 and 2 from the above list: (1) staleness controls a trade-off between iteration quantity and quality and (2) there is an ideal point on that trade-off that is often not the ”default” value.

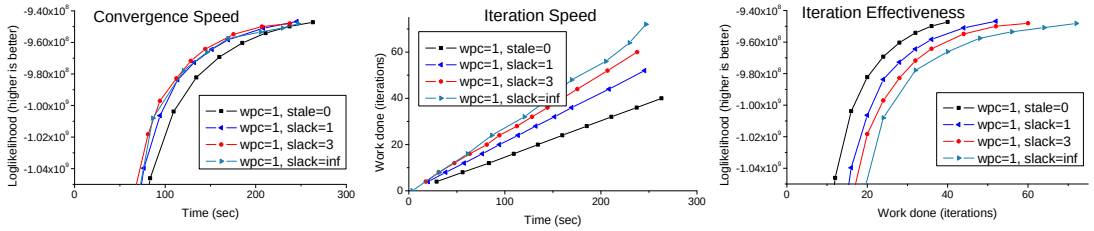
These experiments were run on an 8-node cluster of 64-core machines. Each node has four 2.1 GHz 16-core Opteron 6272s and 128 GB of RAM. The nodes run Ubuntu 12.04 and are connected via an Infiniband network interface (40Gbps spec; $\approx$ 13Gbps observed).

A-BSP

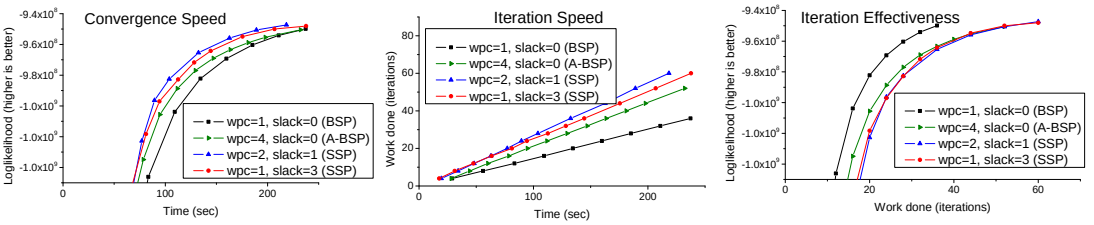
Figure 4.7a shows the effects of tuning work-per-clock in an A-BSP system. These results were generated using the Topic Modeling application. The leftmost graph shows overall *convergence speed*, which is the result quality as a function of time. One way to look at these results is to draw a horizontal line and compare the time (x-axis) needed to reach a fixed log-likelihood value (y-axis). For instance, to reach a log-likelihood of -9.5e8, WPC=1 takes 236 seconds, while WPC=2 takes only 206 seconds: a 13% improvement. These results demonstrate the second point: the ideal staleness setting is not necessarily the



(a) Tuning work-per-clock in A-BSP



(b) Tuning slack in SSP



(c) Direct comparison of SSP with A-BSP

Figure 4.7: Different ways of exploiting data staleness in Topic Modeling. A-BSP adjusts the amount of work done in each clock period, while SSP adjust the ”slack”, i.e. how many clocks a thread can get ahead of others before it will be blocked.

one implicitly chosen by BSP (WPC=1).

The middle and rightmost graphs explain this behavior in the context of the first point – iteration quantity vs quality. The middle graph plots iteration speed (quantity). The graph shows that increasing the staleness (WPC) increases the iteration rate, but with diminishing returns. The rightmost graph shows the log likelihood as a function of iteration number, i.e. the iteration quality. In this plot, higher points indicate that the system was able to get more work done in the same number of iterations, representing improved iteration quality. This plot shows that increased staleness hurts iteration quality. Because of the diminishing returns in iteration quantity, these two graphs together explain why there is a ”sweet spot” in staleness.

SSP

The next set of results (Figure 4.7b) shows the effect of changing the staleness by adjusting the slack in SSP. Again, these results were generated using the topic modeling application. In addition to two SSP settings (slack=1 and slack=3), the results include the fully asynchronous system (slack=inf). These results are qualitatively similar to the A-BSP results from the previous section.

One important difference is the behavior of the asynchronous system. Its performance is not noticeably different than SSP for most of the experiment. However, it struggles during the "fine tuning" portion at the end of the experiment. Its behavior begins to lag behind both SSP configurations, and even the BSP configuration catches up with it and converges faster. This is evidence that freshness requirements should be dynamic. In this case, stale data is acceptable at the start of the algorithm, but near the end the algorithm requires fresh data in order to converge. Additionally it is interesting to note that, while BSP, A-BSP, and SSP all provide convergence guarantees, asynchronous execution does not.

Comparing A-BSP and SSP

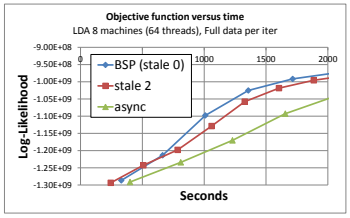
Figure 4.7c compares the results of 4 configurations: BSP, the best performing A-BSP (WPC=4), the best performing SSP (slack=3), and the best overall performer: a hybrid configuration with WPC=2 and slack=1. Interestingly, the three "best in class" configurations use the same staleness bound of 4 (recall that staleness is $wpc \times (slack + 1)$). These results show that SSP outperforms both A-BSP and BSP, and that the best configuration is the hybrid. To reach -9.5e8, A-BSP takes 237 seconds, while the pure SSP approach takes 222 seconds (7% improvement over A-BSP), and the hybrid takes 194 seconds (18% improvement over A-BSP).

Effect of Cluster Size

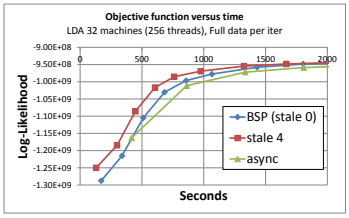
Though powerful many-core servers are now available, it is still often necessary to scale out to large clusters. The last round of experiments in this section demonstrates the effect of staleness as cluster size increases. While the previous two sections used a relatively small cluster of powerful servers (8 nodes of 64 cores each), this sections shows results for two cluster sizes (8 nodes and 32 nodes) of smaller servers (8 cores and 16GB of RAM each).

Figure 4.8 shows the results of this experiment for three configurations: BSP, SSP, and Asynchronous. The leftmost plot are the results on a small cluster of only 8 servers, while the middle plot shows the larger 32 server cluster. Observe that the performance of SSP was actually *worse* than BSP on the small cluster. This is because the relatively weak nodes in that cluster take a long time to finish the computation phase of an iteration, and spend proportionally less time in the communication phase. Therefore the improved iteration

Topic Modeling 8 VMs



Topic Modeling 32 VMs



Topic Modeling 32 VMs, 10% minibatches

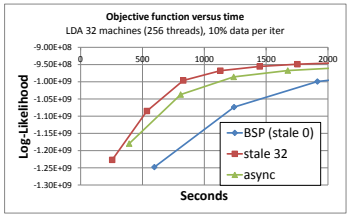


Figure 4.8: Effect of cluster size on BSP and SSP. BSP is more efficient on a small cluster, while SSP's reduced communication overhead makes it faster on a larger cluster. In the case of 10% minibatches – meant to approximate the computation/communication ratio of a very large cluster – SSP does even better.

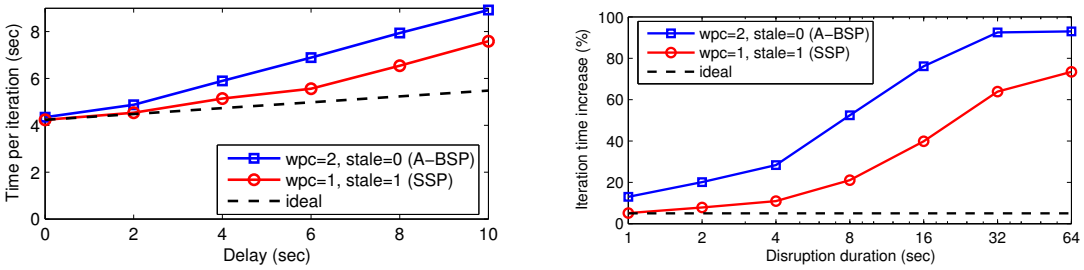
quality of BSP wins out over the reduced communication of SSP.

Running the experiment on a larger cluster (the middle graph) shows a different story. In this case the data set is spread out across 4 times as many machines, shortening the iteration time and causing the application to spend relatively more time in communication and synchronization. These results show that the performance of SSP overtakes BSP at this cluster size, as it is able to reduce time spent in communication and synchronization. We hypothesize that increasing staleness becomes more important on ever larger clusters.

Though we didn't have a larger cluster available, we tested this hypothesis by using minibatches. The graph on the right shows the performance on a cluster of 32 servers where each iteration only processed 10% of the input data. Thus, in this case "BSP" is actually A-BSP with WPC=0.1, and SSP is a hybrid model with WPC=0.1 and slack=32. We believe that this approximates the effect of running on a larger cluster because the application spends relatively less time in computation and more time in communication and synchronization. These results show a substantial advantage to using SSP, and for the first time in this series of experiments, even Asynchronous outperforms BSP.

4.6.3 Flexibility and Disruptions

This section explores the robustness of SSP in the face of intermittent straggler effects from delayed threads and background work. These results use the Topic Modeling application with the same 8 node $\times$ 64 core cluster as the previous section.



(a) Delayed threads

(b) Background disrupting work

Figure 4.9: Influence of delays and background work. A-BSP performance quickly degrades as the delay or disruption duration increases. SSP is able to mask delays and disruptions up to the length of one iteration (about 4s), after which it behaves similarly to A-BSP.

Delayed Threads

The first experiment examines how slack in SSP can mask the effects of delayed threads. In a real-world setting, delay might occur as a result of a stop-the-world garbage collector or single thread calculating a stopping condition. In these experiments, threads are delayed by forcing one thread to sleep at the end of each iteration on a round-robin schedule: thread 1 sleeps in iteration 1, thread 2 sleeps in iteration 2, etc. The length of the sleep d is varied to simulate different levels of interruption. With only 1 thread sleeping in each iteration, the sleep time should average out to $\frac{d}{N}$ seconds per iteration, where N is the total number of threads.

Figure 4.9a shows the average time per iteration as a function of the sleep time. Curves are shown for SSP, an equivalent A-BSP setting with the same staleness, and an ideal system where delays are completely spread out across all servers. Note the baseline time per iteration without delays was 4 seconds. For A-BSP the average time per iteration increases linearly with slope 0.5, because threads synchronize every 2 iterations (WPC=2), and experience a delay of d every time they synchronize. For SSP, the effect of the delayed threads is almost completely masked when the delay is less than the iteration time. With delays up to and slightly exceeding the baseline iteration time (up to about 6s) the iteration time is close to ideal. However, after that SSP is no longer able to mask the delay and the slope matches that of A-BSP, but the absolute iteration time remains 1.5s less than A-BSP.

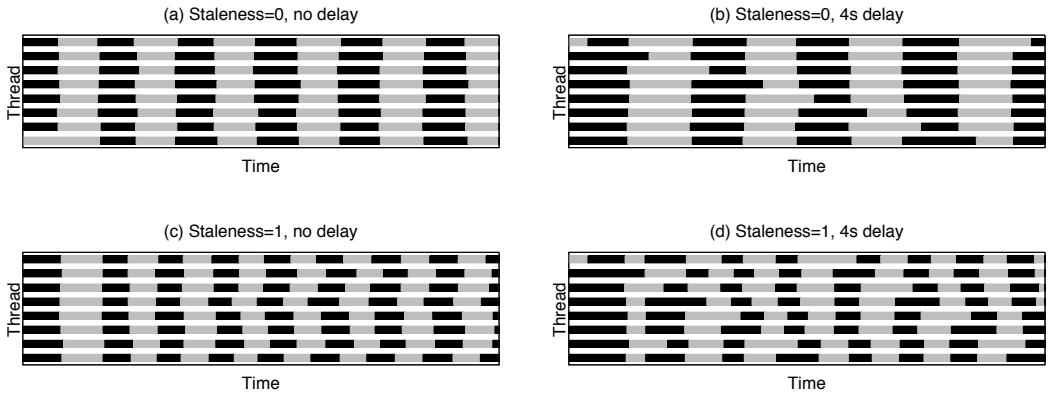


Figure 4.10: Swimlane diagram of the first 100s of execution. Alternating gray and black bars indicate progress from one iteration to the next. The frequency of stripes indicates iteration speed, so more stripes corresponds to faster iteration execution.

Background Work

The second experiment induces stragglers with a competing computation. In addition to the Topic Modeling application, each machine is configured to run a "disruptor" process. The disruptor creates 1 thread per core that performs CPU-intensive work when activated; so, the CPU scheduler will give half of the CPU resources to the disruptor when it is active. Using time slots of size t, each machine's disruptor is active in a time slot with a probability of 10%, independently determined.

Figure 4.9b shows the increase in iteration time as a function of t, the time of a single disruption. Ideally the increase would be just 5%, because the disruptor processes take away half of the CPU for 10% of the time. However, because threads must frequently synchronize, actual performance is worse. The results are similar to the previous experiment. The iteration time of A-BSP increases steadily as the length of the disruptions increases. SSP is able to mitigate disruptions that are less than an iteration length. Once the disruption is longer than a single iteration, SSP follows the same degradation curve as A-BSP, albeit with much better absolute performance.

Visualizing Iteration Progress

Figure 4.10 shows a "swimlane diagram" of the first 100 seconds of the Topic Modeling application for four different configurations of slack and round-robin delay. Each row of the diagram represents a thread. The alternating gray and black bars represent the different

iterations.

Diagram (a) shows a BSP execution (staleness 0) with no delay. The vertical alignment of the bars is caused by threads executing the same iteration at the same time. (b) represents a BSP execution with a 4 second delay round-robined between the threads. Like in the previous diagram, all threads begin executing an iteration at the same time. However, one thread is delayed in each iteration, visible by the diagonal (upper-left to lower-right) pattern of elongated bars. Other threads must wait for the delayed thread to finish before they can start their next iteration. As a result, fewer iterations are completed in the 100s window shown, as seen in the smaller number of stripes.

The bottom two diagrams show the SSP case (staleness 1). (c) is the case with no delay. Unlike in the BSP case, threads are not waiting for one another to finish before starting their next iteration. This is visible in the raggedness of the vertical lines in the diagram. Even without the artificial staleness, the reduced synchronization improves iteration speed. (d) shows the BSP execution with a 4 second delay. Here, each thread can proceed at its own pace, within the staleness bounds, significantly reducing the impact of the stragglers.

Slack Settings

The experiments in section 4.6.2 found that the best overall staleness setting has a slack of 1, and a WPC of 2. Indeed, across many experiments with all applications, we found that it is generally best to set slack=1 and adjust WPC to find the best overall staleness bound. The results from this section explain this phenomenon: in our experimental setup, a slack of 1 clock is sufficient to overcome straggler effects, after which increasing WPC, rather than slack, reduces communication costs more effectively. We expect that systems experiencing longer disruptions would benefit from increased slack.

4.6.4 Microbenchmarks

The last section of experiments presents microbenchmark results that help explain the results in the previous section. Here we explore the amount of time the application spends in computation vs synchronization, and how that is affected by cluster size. Furthermore, we quantify the amount of data sent and received under different configurations of A-BSP and SSP. Lastly, we show why SSP can be less efficient than an equivalent A-BSP configuration by quantifying the CPU overhead of our communication system.

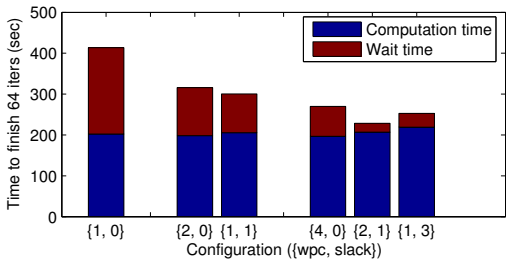


Figure 4.11: Computation time vs wait time for the Topic modeling application. Each bar represents a different staleness setting. The notation {X,Y} indicates WPC=X, slack=Y. Each group of bars has the same effective staleness.

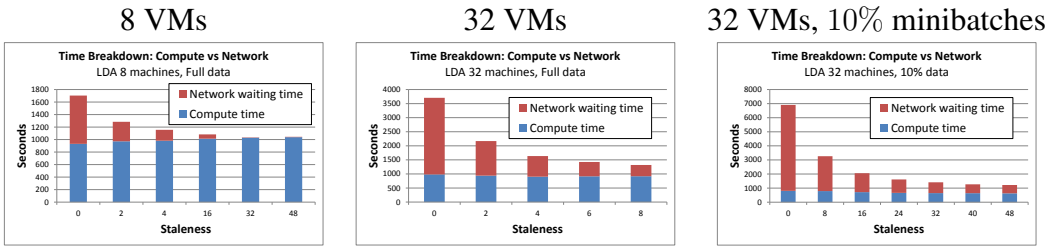


Figure 4.12: Computation time vs network wait time for different cluster sizes. Like before, the 10% minibatches approximate an even larger cluster. The time spent in network wait (thus the importance of staleness) increases with cluster size.

Computation vs Network Wait Time

Figure 4.11 shows the time to complete 64 iterations of Topic Modeling divided into two parts: computation time and network wait time. In this context, network wait time is the time that the application is blocked in the `read` operation, waiting for fresh data to arrive from the parameter server. As we saw in section 4.6.2, iteration speed increases with increased staleness. The plot shows that this improvement in iteration speed comes from a reduction in wait time. Intuitively, this makes sense: if an application will accept staler data, then it will spend less time waiting for fresher data. Section 4.6.2 observed diminishing returns when increasing staleness. These graphs explain that phenomenon. As the network wait time becomes a smaller fraction of the total time, there is less to be gained by reducing wait time.

Computation vs Network Wait Time with Varying Cluster Sizes

Figure 4.12 presents a similar breakdown, but in the context of changing cluster sizes. These measurements came from the same experimental setup as section 4.6.2 (32 nodes $\times$ 8 cores). The results in that section showed an increasing importance of staleness on larger cluster sizes. This plot explains why that is. When using BSP on 8 nodes, the application spends a bit over half of its time in computation. On 32 nodes that number drops to less than a third on computation, and with 10% minibatches (to simulate a larger cluster), it spends less than 15% of its time on computation. As the cluster size increases, the fraction of time spent in communication also increases, therefore there is a greater payoff to strategies like staleness that will reduce communication.

Data Sent/Received

Config.	Bytes sent	Bytes received
wpc=4, slack=0	33.0 M	29.7 M
wpc=2, slack=1	61.9 M	51.0 M
wpc=1, slack=3	119.4 M	81.5 M

Table 4.2: Bytes sent/received per client per iteration of TM. Because SSP configurations can use stale data, the bytes received (i.e. traffic from server to client) does not double with each decrease in WPC.

Table 4.2 provides measurements of data transmitted during a single iteration (pass over all data) of the Topic Modeling application. The three configurations presented all have equivalent staleness (4), but with different WPC and slack. “Sent” denotes traffic sent from the client to the server, while “received” denotes traffic from the server to the client. Most of the sent traffic is the result of “update” operations, while most of the ”received” traffic is from read operations.

Reducing work-per-clock from 4 to 2 means that the application requires twice as many clocks to complete a single iteration. In a configuration without any slack, this would result in a doubling of both the bytes sent and bytes received. However, because the slack is increased and the application uses stale data, the bytes received (quantity of read traffic) does not double. If it doubled each time, we would expect it to be about 60 MB at WPC=2, and 120 MB at WPC=1. However, the actual measurement at WPC=1 is only about 80 MB, indicating about 30% savings with a slack of 3.

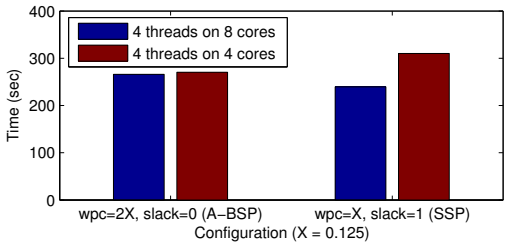


Figure 4.13: CPU overhead of communication: the blue bar represents a configuration with where all threads, including communication threads, run on their own core. The red bar represents a CPU-constrained configuration, where communication threads share cores with computation threads. The relative increase between blue and red indicates the overhead of communication. This overhead is negligible for A-BSP, but significant for SSP.

CPU Overhead of Communication

The extra communication costs of SSP relative to A-BSP can degrade performance even with equivalent stalenesses. Given the small amounts of data transferred in an iteration, most of this extra overhead is the CPU cost of communication: serializing and deserializing data structures, locking shared data, allocating memory for new data, etc. To illustrate this, we performed an experiment with A-BSP and SSP in a CPU-constrained configuration and a non-CPU-constrained configuration. In both cases, the application was configured to use 4 threads. In addition to these threads, there are two communication threads in the client process, and 2 threads used by the server process that runs on the same machine as the client. Collectively, we consider these 4 threads to be ”communication threads”. In the CPU-constrained configuration, the entire virtual machine was given only 4 cores, so that the 4 application threads would be forced to share CPU resources with the communication threads. In the unconstrained configuration it was given 8 cores, so that these threads would not compete with each other for CPU resources. Figure 4.13 shows the results of this experiment. Observe that there is very little difference in the performance of A-BSP in a constrained or unconstrained environment. However, the time to complete an iteration in SSP increases by about 25% when CPU resources are constrained.

Prefetching

Figure 4.14 shows the effect of prefetching on our example applications. The time per iteration is partitioned into computation time and wait time. In each case, prefetching significantly reduces wait time. The speedup for matrix factorization is the most dramatic for two reasons. First, there is less data sharing between threads in this algorithm than in

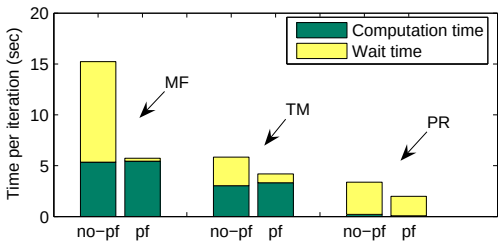


Figure 4.14: Effect of prefetching on computation time and wait time.

the other two examples. With LazyTables' shared process cache, data sharing between threads can act as a form of prefetching: when one thread gets new data for a row, that data is available for all other threads on the same host. Secondly, matrix factorization iterations are particularly long, so the useful lifetime of a prefetched row is extended.

4.6.5 Summary of Results

The results in this section demonstrate that A-BSP and SSP (as well as hybrids of the two), are viable techniques to introduce staleness into iterative convergent algorithms. In both cases, increasing the staleness improves iteration rate while degrading iteration quality. Furthermore, there is a "sweet spot" in the configuration space where the improvement in rate overwhelms the degradation in quality. The precise configuration that matches this sweet spot depends on factors including the behavior of the algorithm, the presence of stragglers and performance jitter in the environment, the number of servers and CPUs, and the cost of communication between nodes.

4.7 Open Problems

The evaluation of our LazyTables prototype demonstrates many promising results using the Arbitrarily-size BSP and Stale Synchronous Parallel. However, we believe that there are still many open problems in staleness-aware consistency models and the design of systems that use them. This section outlines some of these open problems.

Prefetching Policies

Section 4.4.2 demonstrates how LazyTables can use knowledge about applications' freshness requirements to improve performance by prefetching data. The results show two

policies for controlling prefetching: an application-controlled policy and an automatic policy. While the application-driven approach provides excellent performance, it requires the programmer to be aware of their access patterns and provide those hints to the system. An effective automatic policy is desirable for ease-of-use by application programmers.

The automated policy presented in Section 4.4.2 and evaluated in Section 4.6.4 shows that automated policies can be effective for some workloads. While an in-depth study of automated prefetching is outside of the scope of this thesis, the initial results are promising, and suggest that it could be an important area of further study. Generally, we expect the access patterns to be cyclical: as a thread loops over the same data many times, it tends to access the same set of parameters. We believe that it would be useful to design prefetching strategies for exploiting this cyclical behavior.

Dynamic Staleness

Section 4.6.2 suggests that some problems may benefit from changing the staleness setting as the algorithm progresses. There are a number of ways this could potentially improve some algorithms, and the strategies are likely to be algorithm-dependent. For instance, some algorithms begin with a phase where all threads need to "get in sync". For instance, the first few iterations of LDA are often called "topic alignment". In these iterations, threads must come to agreement on which numbered topics correspond to which categories of words. Communication during this phase is important, but once threads have roughly the same topics, they can proceed mostly independently. This suggests that LDA is an example where fresh data is important in early iterations, and more staleness can be tolerated in the later iterations.

On the other hand, algorithms like gradient descent end with a "fine tuning" phase, where interactions between different threads may become more important for the algorithm to continue making progress. For these algorithms, it may be beneficial to start with a high staleness to quickly find an approximate solution. Once the algorithm finds the approximate solution, the staleness tolerance is reduced to improve the accuracy of the solution. One possible way of doing this is to periodically compute the global objective value. When the system detects that improvement in the objective value is slowing down, it can increase the freshness setting to compensate.

Generally, algorithms that benefit from dynamic staleness are those that are less sensitive to error during particular phases of their execution. Another strategy to improve the performance of these algorithms is to undersample data during periods where they are less sensitive to errors. While this would allow them to iterate faster, it could increase the ratio of communication to computation, hurting efficiency.

Classifying Staleness-Tolerance

Using staleness to improve performance is based on the observation that many iterative convergent algorithms can tolerate small errors (and therefore staleness). In our experiments, we have noticed that some algorithms (e.g. matrix factorization), respond well to some degree of staleness, but if the staleness setting is too high, the algorithm will diverge. A more complete classification of staleness-tolerant algorithms would be a useful contribution. This would involve classifying which algorithms can tolerate any staleness at all, and determining what factors of the algorithm and input data will affect what staleness setting causes the algorithm to diverge.

4.8 Conclusions

The trade-off between freshness and performance is important in the design of parameter servers for distributed machine learning applications. Existing systems using Bulk Synchronous Parallel are already operating on stale data produced by the previous iteration (clock in our terminology). The notion of Arbitrarily-sized BSP (A-BSP) makes this trade-off explicit. By adjusting the amount of data processed in each clock, the application designer can choose between fresher data and reduced communication and synchronization overhead.

Building on this idea, the Stale Synchronous Parallel model provides a different way of tuning freshness. In the SSP model, threads do not wait for one another to finish a particular clock before proceeding. Each time a thread reads data it specifies how stale the result of the read operation may be. For instance, by giving a staleness bound of 2, the read operation must incorporate all updates as of 2 iterations ago. This has the effect of a "soft barrier" – if another thread has not finished that iteration yet, the reading thread will have to block until it does. This staleness bound – called the *slack* – provides another parameter that application designers can use to tune staleness.

Results in section 4.6 show that the optimal staleness setting uses both the work-per-clock and slack parameters. Results show that increasing WPC reduces communication overhead more than slack , but it does not improve the performance of systems suffering from straggler effects. The slack parameter is able to reduce synchronization delays, which is particularly important for mitigating straggler effects. In a simple test case, on a small cluster with little performance variation, SSP improved performance over a baseline BSP configuration by 22%. However, when we introduced significant performance variation, the SSP configuration had double the iteration rate of the A-BSP system.

Furthermore, the results in section 4.6 show that staleness becomes more important on larger clusters. On a small cluster of 8 8-core servers, a standard BSP configuration was able to converge faster than the SSP configuration by up to 20% in some cases. However, on a cluster of 32 servers the situation reversed, with SSP improving performance by more than 20%. In an experiment using a 10% batch size – providing similar communication overheads as a 320 server cluster – SSP reduced convergence time by a factor of 2 compared to standard BSP.

5 Conclusions

There is an inherent trade-off in system design, between data freshness and system efficiency. Techniques that improve efficiency – such as caching reads, batching and buffering updates, and creating specialized update pipelines – often reduce data freshness. It is important to recognize this when designing systems, to ensure that the data used by applications is sufficiently fresh. Whenever possible, systems should be designed to delay this decision. Allowing the application to specify its freshness requirements for each individual query ensures that fresh data is available when necessary, while queries that do not need such fresh data can use all of the techniques available to improve performance.

I have classified work that exploits the freshness/efficiency trade-off classified into 4 groups, depending on how it allows applications to configure the trade-off. These groups are:

1. Systems that exploit a freshness/efficiency trade-off without explicitly acknowledging it.
2. Systems that acknowledge the trade-off, but do not provide fine-grained control over it.
3. Systems – like LazyBase and LazyTables – that allow fine-grained application control over the trade-off.
4. Theoretical work on consistency models that incorporates a discussion of freshness and the trade-offs involved.

To demonstrate the advantages of acknowledging this trade-off, this dissertation expands on this body of work by describing two novel systems – LazyBase and LazyTables – that are designed to make the trade-off between freshness and efficiency a first-class concern. These two systems are designed for very different domains and application types. LazyBase is an analytical database. It was originally conceived for meta-data management in large file archives, but is applicable to a wide array of data collection and analysis tasks. LazyTables is distributed programming environment for Machine Learning calculations on large clusters.

Together, they show the wide applicability of the freshness/latency trade-off. Individually, they are important contributions in their own right. LazyBase was the subject of a 2012

EuroSys paper [Cipar et al., 2012], and is currently being used in a commercial archival storage product from HP: StoreAll Express Query [Johnson et al., 2014]. Results from the LazyTables project have been published in HotOS [Cipar et al., 2013], NIPS [Ho et al., 2013], and Usenix ATC [Cui et al., 2014]. The CMU BigLearning research group is continuing to study the design and use of parameter servers with configurable freshness properties, with multiple contributions to the research and open-source communities.

LazyBase is a novel database designed with data freshness as a first-class concern. LazyBase is used for analytics applications that operate over large rapidly-changing data sets. As an example, it is a component of a commercial product [Johnson et al., 2014], being used for meta-data management of very large file archives. To support these types of applications, LazyBase collects updates in large batches and processes them in a pipelined fashion. The pipeline stages implement data transformations that must occur before the data can be applied to the main data repository (the Authority Table). These transformations include sorting, mapping foreign keys, and a tree-based merge.

The use of large batches and a pipelined architecture allows LazyBase to scalably ingest large volumes of updates. Results in Chapter 3 show that it scales linearly up to 20 servers, with update rates typically 4 times faster than a well-known NoSQL database. This efficiency comes at the expense of freshness. Even with a simple database schema, experiments demonstrate that data in the authority table can be 10 to 20 seconds stale, relative to the ingest servers. In some configurations we have observed data staleness on the order of minutes or hours.

Part of LazyBase's novel design is a query engine that is able to access intermediate data produced by the pipeline stages. When an application issues a query with a freshness requirement, it is possible that the data in the Authority Table will be too stale to satisfy the query. By accessing data that is still being processed by the ingest pipeline, the query engine can retrieve fresher results to satisfy the query. However, doing so reduces the efficiency of the query. Intermediate pipeline data may be unsorted, exist in many files, or use local (rather than global) row IDs.

LazyTables is a distributed *parameter server*: a system for storing and computing the parameters of statistical models used in Machine Learning. These parameters are typically large, sparse vectors or matrices. Typical distributed ML applications follow the Bulk Synchronous Parallel (BSP) model, where individual threads work together in iterations, operating on shared data produced by the previous iteration. As part of the LazyTables work, we frame BSP as an example of using stale data to improve efficiency. Specifically, it is an example of the first class of application described above: it exploits the trade-off between freshness and efficiency without explicitly recognizing it.

The Arbitrarily-sized BSP model (A-BSP) makes this trade-off explicit. It is not a truly different model from BSP. Instead, it is a technique for writing BSP applications so that the amount of staleness can be controlled by the application, based on a single parameter called *work per clock* (WPC). Increasing the work per clock allows the application to iterate more quickly, but it will make less progress per iteration. The correct setting for work per clock can improve the convergence time by 13%.

The Stale Synchronous Parallel (SSP) model is a generalization of A-BSP with a per-read-operation *slack* parameter. The default slack is 1, which gives the same behavior as A-BSP: read operations return data from the previous clock. Setting a higher slack value allows the operation to return staler data. A slack of 3 indicates that the data may be up to 3 iterations old. By issuing all reads with an increased slack value, some threads may temporarily proceed a few iterations ahead of others. This increased flexibility in scheduling makes the system more resilient to intermittent performance degradation of some threads or machines. Even our tightly-controlled environment, setting an increased slack value along with a good work per clock improved performance by 22% relative to BSP.

Outside of our experimental environment, we expect these effects to be magnified. Staleness becomes more important with increased cluster sizes. This is because the time spent processing becomes relatively small compared to the communication costs, so there is more benefit to reducing communication overhead. Slack, in particular, becomes more important with increased performance variability. Running on larger clusters, especially those allocated in a cloud environment, increases performance variability. On our simulated "large cluster" experiment, SSP outperforms BSP by more than a factor of 2.

These two systems, LazyBase and LazyTables both provide mechanisms to allow the programmer to specify application-specific freshness requirements, but neither one dictates how to determine the correct freshness setting. In the case of LazyBase, where freshness requirements are expressed in real time, it is possible to guess what a reasonable freshness requirement might be. For instance, user-to-user messages in a social network require very fresh data, no staler than a few seconds. On the other hand, recommendation engines, e.g. determining what video a user may want to watch next, can use data that is minutes old.

LazyTables presents a bigger challenge, because there is no intuitive notion of "correct" freshness. For applications that are run repeatedly with different input data, the application designer or user can spend time tuning the parameters for their particual system. In general, setting these parameters automatically would be easier for the user, and more robust to changes in behavior with different data or hardware. Algorithms for automatically tuning staleness parameters are an interesting domain for future work.

Another area of future work is in the debuggability of programs that use stale data. The

use of staleness for intermediate data, as is done is LazyTables, may make it more difficult to debug parallel programs. It also may expose bugs that would not have been a problem in a system without staleness. One strategy for dealing with the former issue is to configure the system to avoid stalness for debugging purposes (in LazyTables, set slack to 0). It may also be useful to have a ”strict” mode for debugging, where the system never returns fresher-than-necessary data, ensuring that the computation is deterministic. The second problem, where stalness introduces new bugs or exposes otherwise harmless bugs, is more difficult. Indeed, it is harder to reason about systems where there is more than one correct output. In experimenting with LazyTables, we found a case of this problem. If the slack was set too high, the Matrix Factorization application would fail to converge to a solution. However, it was clear from the output that it was failing to converge, and we were able to find appropriate slack settings by trial and error.

There are many trade-offs that must be made when designing a software system. The trade-off between data freshness and system efficiency is present in many system designs, and has a significant effect on how the system can be used. To support a wide range of applications, systems should be designed to make this trade-off dynamically. Results from LazyBase, LazyTables, and examples from chapter 2 demonstrate the value of allowing each query to the system to configure this trade-off independently.

Bibliography

B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proc. SIGMOD*, 1995. 2.2, 3.6

Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012. 4.2.5, 4.4, 4.5.1

Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Loose synchronization for large-scale networked systems. In *USENIX Annual Tech*, pages 301–314, 2006. 4.2.5, 4.3.4

Nick Roussopoulos Alexandros Labrinidis. Balancing performance and data freshness in web database servers. pages pp. 393 – 404, September 2003. 1, 2.3

Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, pages 1–16, 2010. URL `http://dl.acm.org/citation.cfm?id=1924943.1924962`. 4.2.4

Eric Anderson, Martin Arlitt, Charles B. Morrey III, and Alistair Veitch. DataSeries: An efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1):70–75, January 2009. 3.3.8

Apache Mahout. Apache Mahout, http://mahout.apache.org. URL `http://mahout.apache.org`.

Michael Armbrust, Armando Fox, David A. Patterson, Nick Lanham, Beth Trushkowsky, Jesse Trutna, and Haruki Oh. SCADS: Scale-independent storage for social computing applications. In *Proc. CIDR*, January 2009. 2.3, 3.6

Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, Dave Patterson, and Kathy 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. 3.6

Charles Babcock. Data, data, everywhere. *Information Week*, January 2006. 3.2

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212359. URL http://dx.doi.org/10.14778/2212351.2212359. 2.4

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Pbs at work: Advancing data management with consistency metrics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1113–1116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465260. URL http://doi.acm.org/10.1145/2463676.2465260. 2.4

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with pbs. *Commun. ACM*, 57(8):93–102, August 2014. ISSN 0001-0782. doi: 10.1145/2632792. URL http://doi.acm.org/10.1145/2632792. 2.4

P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. pages 1–12, 2006. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4100352. 4.2.4

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=944919.944937. 4.1, 4.5.1

Mokrane Bouzeghoub. A framework for analysis of data freshness. In *Proceedings of the 2004 international workshop on Information quality in information systems*, IQIS '04, pages 59–67, New York, NY, USA, 2004. ACM. ISBN 1-58113-902-0. doi: 10.1145/1012453.1012464. URL http://doi.acm.org/10.1145/1012453.1012464. 2.4

Joseph Bradley, Aapo Kyrola, Daniel Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 321–328, June 2011. ISBN 978-1-4503-0619-5. 4.3, 4.3, 4.5.3

Laura Bright and Louiqa Raschid. Using latency-recency profiles for data delivery on the web. In *VLDB*, pages 550–561. Morgan Kaufmann, 2002. URL http://dblp.uni-trier.de/db/conf/vldb/vldb2002.html#BrightR02. 1, 2.3

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. 3.6

R. Campbell, I. Gupta, M. Heath, S. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H.Y. Lee, M. Lyons, D. Milojicic, D. O'Hallaron, and Y.C. Soh. Open cirrus cloud computing testbed: Federated data centers for open source systems and services research. In *Proc. of USENIX HotCloud*, June 2009. 3.4.1

Cassandra. Apache Cassandra, http://cassandra.apache.org/. 2.3, 3.1, 3.3.5, 3.4, 3.6

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A distributed storage system for structured data. In *Proc. OSDI*, November 2006. 3.6

S. Chaudri, U. Dayal, and V. Ganti. Database technology for decision support systems. *Computer*, 34(12):48–55, December 2001. 3.1

James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 169–182, 2012. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168854. URL http://doi.acm.org/10.1145/2168836.2168854. (document), 5

James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, 2013. USENIX. URL https://www.usenix.org/conference/hotos13/solving-straggler-problem-bounded-staleness. 5

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. 3.4

CouchDB. CouchDB, http://couchdb.apacheorg/. 3.1

Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 37–48, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `http://dl.acm.org/citation.cfm?id=2643634.2643639`. 4.3.1, 5

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004. 3.3.1

Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, 1996. ISBN 0-89791-793-6. doi: 10.1145/233013.233020. URL `http://doi.acm.org/10.1145/233013.233020`. 4.2.5

Mark Evans. A look at Twitter in Iran . http://blog.sysomos.com/2009/06/21/a-look-at-twitter-in-iran/, June 2009. 3.2

Kurt B. Ferreira, Patrick G. Bridges, Ron Brightwell, and Kevin T. Pedretti. The impact of system design parameters on application noise sensitivity. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4220-1. doi: 10.1109/CLUSTER.2010.41. URL `http://dx.doi.org/10.1109/CLUSTER.2010.41`. 4.2.5

Avigdor Gal. Obsolescent materialized views in query processing of enterprise information systems. In *Proceedings of the eighth international conference on Information and knowledge management*, CIKM '99, pages 367–374, New York, NY, USA, 1999. ACM. ISBN 1-58113-146-1. doi: 10.1145/319950.320029. URL `http://doi.acm.org/10.1145/319950.320029`. 2.4

Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pages 69–77. ACM, 2011. 4.5.2

Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 2013. (document)

Jeremy Ginsberg, Matthew H. Mohebbi, Rajan S. Patel, Lynnette Brammer, Mark S. Smolinski, and Larry Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, pages 1012–1014, February 2009. 3.2

Goetz Graefe. Write-optimized B-trees. In *Proc. VLDB*, pages 672–683, 2004. 3.6

Goetz Graefe. B-tree indexes for high update rates. *ACM SIGMOD Record*, 35(1):39–44, March 2006. 3.6

Greenplum. EMC Greenplum, http://www.greenplum.com/. 3.6

Chris Grier, Kurt Thomas, Vern Paxson, and Michael Zhang. @spam: The underground on 140 characters or less. In *Proc. of ACM Conf. on Computer and Communications Security*, October 2010. 3.4.1

Hadoop. http:// hadoop.apache.org/. 3.6

Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, pages 205–220, 2007. 2.3, 2.4, 3.6

HBase. Apache HBase, http://hbase.apache.org/. 3.1, 3.6

Doug Henschen. 3 big data challenges: Expert advice. *Information Week*, October 2011. 3.2

Stefan Hildenbrand. Performance tradeoffs in write-optimized databases. Technical report, Eidgenossiche Technische Hochschule Zurich (ETHZ), 2008. 2.2, 3.6

Pieter Hintjens. ZeroMQ: The Guide, 2010. URL `http://zguide.zeromq.org/page:all`. 4.6.1

Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013. 5

Charles Johnson, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, Alistair Veitch, Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer,

Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro. From research to practice: Experiences engineering a production metadata database for a scale out file system. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 191–198, Santa Clara, CA, 2014. USENIX. ISBN ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/johnson. 5

James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, February 1992. ISSN 0734-2071. doi: 10.1145/146941.146942. URL http://doi.acm.org/10.1145/146941.146942. 1

Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks are like snowflakes: no two are alike. In *Proceedings of the USENIX conference on Hot topics in operating systems*, pages 14–14, 2011. URL http://dl.acm.org/citation.cfm?id=1991596.1991615. 4.2.4

Lena Lau and Keith Gabrielski. Personal conversation with Lena Lau and Keith Gabrielski – "Big Data" engineers at Fiksu. 2.1

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. 3.6

Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu. 4.4

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010. 1, 4.2.5

Yucheng Low, Gonzalez Joseph, Kyrola Aapo, Danny Bickson, Carlos Guestrin, and M. Hellerstein, Joseph. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012. 4.2.5

Sergey Melnik, Andrey Gubarev, Jing Jong Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB*, pages 330–339, September 2010. 3.6

MongoDB. MongoDB, http://www.mongodb.org/. 3.1

David Newman, Arthur U. Asuncion, Padhraic Smyth, and Max Welling. Distributed inference for latent dirichlet allocation. In *NIPS*, 2007. 4.1

Feng Niu, Benjamin Recht, Christopher R, and Stephen J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *In NIPS*, 2011. 4.2.5

Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. OSDI*, pages 1–15, 2010. URL `http://dl.acm.org/citation.cfm?id=1924943.1924961`. 3.6

Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 55–, 2003. ISBN 1-58113-695-1. doi: 10.1145/1048935.1050204. URL `http://doi.acm.org/10.1145/1048935.1050204`. 4.2.4, 4.2.5

Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proc. SIGMOD*, July 2009. 3.6

Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, pages 1–14, 2010. URL `http://dl.acm.org/citation.cfm?id=1924943.1924964`. 4.4

Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS - a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proc. VLDB*, pages 754–765, 2002. 3.6

Kenneth Salem, Kevin Beyer, and Bruce Lindsay. How to roll a join: Asynchronous incremental view maintenance. In *Proc. SIGMOD*, 2000. 2.2, 3.6

Evan Sandhaus. The new york times annotated corpus. https://catalog.ldc.upenn.edu/LDC2008T19, 2008. 4.1

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. 3.6

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. 3.6

Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. *Proc. ICDE*, pages 25–36, 2003. 3.6

SimpleDB. Amazon SimpleDB, http://aws.amazon.com/simpledb/. 3.6

Aman Singla, Umakishore Ramachandran, and Jessica Hodgins. Temporal notions of synchronization and consistency in beehive. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 211–220, New York, NY, USA, 1997. ACM. ISBN 0-89791-890-8. doi: 10.1145/258492.258513. URL `http://doi.acm.org/10.1145/258492.258513`. 2.3

Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920931. URL `http://dx.doi.org/10.14778/1920841.1920931`. 4.4

SQLite. SQLite database speed comparison. http://www.sqlite.org/speed.html. 2.1

Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proc. 10th USENIX Conf. on File and Storage Technologies (FAST)*, 2012. 3.6

Douglas Terry. Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research, October 2011a. 3.2

Douglas Terry. Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research, October 2011b. 1, 2.4, 4.3.2

Christian Thomsen, Torben Bach Pedersen, and Wolfgang Lehner. RiTE: Providing on-demand data for right-time data warehousing. In *Proc. ICDE*, 2008. 3.6

Thrift. Apache Thrift, http://thrift.apache.org/. 3.3.1

Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association. ISBN 978-1-931971-82-9. URL `http://dl.acm.org/citation.cfm?id=1960475.1960487`. 2.3

Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522713. URL http://doi.acm.org/10.1145/2517349.2522713. 2.3

Twitter Earthquake. Twitter Earthquake Detector, http://recovery.doi.gov/press/us-geological-survey-twitter-earthquake-detector-ted/. 3.2

Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8): 103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL http://doi.acm.org/10.1145/79173.79181. 1

Vertica. HP Vertica, http://www.vertica.com/. 3.6

Werner Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, January 2009. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1435417.1435432. URL http://doi.acm.org/10.1145/1435417.1435432. 3.2

Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3): 239–282, August 2002. 1, 2.4, 4.3.2

Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 29–42, December 2008. URL http://www.cs.berkeley.edu/~matei/papers/2008/osdi_late.pdf. 4.2.5

Roman Zajcew, Paul Roy, David L. Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala. An osf/1 unix for massively parallel multicomputers. In *USENIX Winter*, pages 449–468, 1993. 4.2.5

Ke Zhai, Jordan Boyd-Graber, Nima Asadi, and Mohamad L. Alkhouja. Mr. lda: A flexible large scale topic modeling package using variational inference in mapreduce. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 879–888, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1229-5. doi: 10.1145/2187836.2187955. URL http://doi.acm.org/10.1145/2187836.2187955. 4.2.5