Bulletin of the Technical Committee on

Data Engineering

June 2012 Vol. 35 No. 2

IEEE Computer Society

Letters

Special Issue on Big Data War Stories

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TC on Data Engineering web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

IEEE Computer Society News

In my role as TCDE Chair, I just attended a Computer Society Technical Activities Committee (TAC) meeting, this year held, very conveniently for me, in Seattle. TAC members consist of TC chairs (like I am for the TCDE) as well as a chair appointed from a higher level of the Computer Society, and some at-large members. The two noteworthy (and recurrent) items addressed at this meeting were (1) governance and(2) finances.

On governance, our governance subcommittee recommendation to have TC chairs elect the TAC chair had been rejected earlier. To address TAC governance, Elliot Chikofsky, our current (appointed) chair, proposed electing a TAC vice chair (an informal position). I was not initially enthusiastic, but I decided to support this after suggesting that the vice chair attend and participate in the meetings of the Technical and Conferences Board (T&C). The T&C is the TAC "parent board" where the TAC chair represents the TAC. This was approved, and a vice chair, Sven Dietrich of the TC on Security and Privacy, was elected. This is a very tentative step, and we will have to see how this plays out.

On the financial front, we heard a proposal suggesting that TC's like TCDE be able to carry over part of their fund balance from year to year. This is important for a few of reasons. (1) It removes the potential large volatility in our budget allocation each year, which depends on conference revenue. (2) It gives us an incentive to initiate profitable conferences. (3) It permits us to do longer range planning, which is difficult when finances are so uncertain. Importantly, however, little detail was provided, and the proposal is under study.

None of the above is final in any sense. But there is at least an indication that the Computer Society may be capable of change for the better.

TCDE Chair Election

I want to draw your attention to the "Nominations for Chair of TCDE" letter from Paul Larson and Masaru Kitsuregawa on page 2. The TCDE chair represents us at TAC meetings like the one described above. The chair also appoints our executive committee and, with their advice, decides which conferences we should sponsor. So there are issues of importance that require attention, and the TCDE Chair is the one who organizes and deals with them. Who serves as our Chair is thus important. I would urge you to think about whom you would like to have in this office and to participate in our electoral process.

TCDE Logo

Attentive Bulletin readers will notice a change in the cover of this Bulletin June issue. Computer Society TCs were asked to provide logos for the Society to use on their web pages and in their literature. We did not have an "official" logo, so we sought our people that we knew who had generated ICDE logos, and asked them to submit proposals. The winning proposal, as voted by the TCDE Executive Committee appears on the cover in place of the Computer Society logo.

This logo was produced by Jun Yang of Duke University. I want to thank Jun for his very successful artistic effort with the logo. It captures that the TCDE is part of the IEEE (the blue diamond), that we deal in data (the 1's and 0's), that we are engineers (the compass), and that this is the TC on DE (look for the "de" in the logo). This is a lot to capture in a small logo but Jun's design succeeds in all these aspects, as well as looking very stylish. Bravo and thanks, Jun!

The Current Issue

"Big data", however one might define that term, is a current catch phrase in our technical community. And for good reason. Commercial enterprises have found that they can extract sufficient "meaning" from huge stores of

data to enable them to have a real competitive advantage. Competitors see that happening, and the result is an arms race involving huge volumes of data and how to exploit it.

This data intensive activity has gone on for longer than the term "big data" has been commonly used. Long enough that there is now a wealth of knowledge about effective ways of operating on and using big data. Long enough that it is timely to take stock in this activity, and to try and understand what works and what does not.

With this in mind, Brian Cooper has brought to the bulletin a collection of "war stories" about how to support, how to mine, and how to exploit huge data volumes to win the marketplace competition. Not surprisingly, authors in this Bulletin issue are drawn from industry. Industrial focus, which I regard as a unique aspect of the Bulletin is hence especially strong in this issue. I want to thank Brian for bringing it to us. I think there is both much to enjoy in the issue, and much to learn.

David Lomet
Microsoft Corporation

Nominations for Chair of TCDE

Nominations for Chair of TCDE

The Chair of the IEEE Computer Society Technical Committee on Data Engineering (TCDE) is elected for a two-year period. The mandate of the current Chair, David Lomet, expires at the end of this year and the process of electing a Chair for the period 2013-2014 has begun. A Nominating Committee consisting of Paul Larson and Masaru Kitsuregawa has been struck. The Nominating Committee invites nominations for the position of Chair from all members of the TCDE. To submit a nomination, please contact any member of the Nominating Committee before August 15, 2012.

More information about TCDE can be found at http://tab.computer.org/tcde/.

Paul Larson and Masaru Kitsuregawa
Microsoft Corporation and University of Tokoyo

Letter from the Special Issue Editor

People are always trying to do more with their data than current technology allows. This leads to a lot of pain, as they fight a battle to get the software to do what they want, without unmanageable amounts of money, time, complexity, or general frustration. When enough people are fighting the same war against their data, new data models and architectures might emerge which promise to bring peace to the data wars. Thus, the rise of relational systems, object oriented systems, XML systems, and so on, each dealing with one or more specific pain points. Unfortunately, each new generation of systems does not end the battle, but merely transform it, as practitioners try to use their new weapons to do even more interesting things with their data.

The latest battle is being fought against "Big Data" - data that is large, complex or dynamic enough to be too painful to handle with existing systems. In the last decade or so, the ease of collecting a mind-numbing amount of data, often but not always from web applications, has led to a desire to do something with all that data. Although a variety of systems have been developed to store, query, analyze and transform big data, many practitioners still feel that they are fighting a war against their data - with newer, more powerful and sophisticated (and some may say, deadly) software tools, but a war nonetheless.

In this issue, we collect "Big Data War Stories" - examples of practitioners trying to use big data tools, finding that it was still painful, and having to adopt new strategies and tools to overcome their difficulties.

- From Facebook, we hear about experiences trying to get a new and promising but relatively untested technology (HBase) to manage a huge number of user messages.
- From Microsoft, we hear about efforts to take noisy maps and shopping data and clean it well enough to make it useful to users.
- From Yahoo, UC Santa Cruz and UC Irvine, we hear about a key limitation of big data storage systems: poor support for machine learning, and new strategies to deal with this limitation.
- From LinkedIn, we hear about how logs of activity data can be very useful in driving web applications, datacenter management and so on, but also how new techniques are needed just to move the right data to the right place to be analyzed.
- From Google, we hear about how just collecting and analyzing big data is not enough, if it cannot also be easily shared and visualized.

In some of these cases, the right tactic was to modify and extend an existing platform or technique. In other cases, a whole new tool was needed to overcome the limitations of big data software.

I hope these stories, and the lessons contained within them, are useful to other practitioners fighting similar battles against their data. I would like to thank all of the authors who agreed to share their experiences, as well as Dave Lomet who, as always, provided excellent advice during the process of putting together this issue.

Brian F. Cooper
Google
Mountain View, CA USA

Storage Infrastructure Behind Facebook Messages Using HBase at Scale

Amitanand Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania
Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan
Nicolas Spiegelberg, Liyin Tang, Madhuwanti Vaidya*
Facebook, Inc.

Abstract

Facebook Messages, which combines messages, chat and email into a real-time conversation, is the first application in Facebook to use HBase in production. In this article, we will discuss why we chose HBase for this use case, the early improvements we did to make HBase production ready, engineering and operational challenges encountered along the way, and the continued work we have had to do once in production, to improve HBase's efficiency and reliability. We will also describe some of the other use cases of HBase at Facebook.

1 Introduction

In 2009, discussion started around possible storage solutions for the next version of the Facebook Messages product. This version of the product would bring together chat, email and the earlier version of messages product into one umbrella. Chat messages were about 20 times the number of traditional messages, and email messages were likely to be much larger in terms of raw size. Previously, chat messages were held in memory and stored only for a small number of days, and the non-chat messages were stored in MySQL. The new product requirements meant that we needed a storage system that could efficiently support a 20x increase in the amount of writes, in addition to providing cheap and elastic storage for datasets that we were expecting to grow at 150TB+/month.

2 Why we picked HBase for Facebook Messages

HBase is a large-scale, distributed database built on top of the Hadoop Distributed File System (HDFS). It is an Apache open source project, and is modeled after BigTable. In this section, we'll describe our reasons for choosing HBase as the storage solution for the new Facebook messages.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*In alphabetical order

2.1 High write throughput

For a use case like Messages, clustering the message data on disk by userid as the primary dimension, and time or other properties such as thread id or word in the message (for search index purposes) as the secondary dimension is a reasonable schema choice since we want to able to retrieve the messages for a given user efficiently.

Traditional RDBMSs use a single mutating B-Tree as the on-disk data structure for each index. When backed by a traditional RDBMS, the Messages workload would cause a lot of random writes on the B-Tree index since the index is clustered by user id, and several users will receive messages at any given time. With ever increasing data sets, this approach incurs a substantial penalty as, over time, each write requires updating potentially a different block of the B-Tree. In addition, the in-place modification of the B-Tree incurs a read penalty on every write because the block being updated first needs to be read.

HBase's Log Structured Merge Tree (LSM) approach to updating its on-disk data makes it very efficient at handling random writes. HBase first collects all updates in a memory data structure and then periodically flushes the in-memory structure to disk creating an immutable index-organized data file (or HFile). Over time several immutable indices get created on disk, and they are merged in the background. In this approach, all writes to the disk are large sequential writes; and the saved disk iops can be in turn be used to serve reads!

2.2 Low latency random reads

Even though we planned to front the storage system with an application level caching layer for Facebook Messages, the storage system needed to be able to serve any misses with reasonably low latencies. HBase offered acceptable performance to begin with, and we knew that even the implementation wasn't as efficient out-of-the-box, we would be able to optimize reads a lot further– especially for read-recent type of workload using techniques like bloom filters, time range hints and lazy seeks.

2.3 Elasticity

We expected the Facebook Messages data sets to constantly grow over time. Being able to add incremental capacity to the system easily and without downtime was important. Both HBase and HDFS are systems built with elasticity as a fundamental design principle.

2.4 Cheap and fault tolerant

We expected the data sizes to reach several petabytes pretty quickly. Using cheap disks and commodity hardware would be the most cost effective way to scale. But failures are the norm with commodity hardware. Any solution must be resilient to node failures and should require no operational intervention. HBase over HDFS provides a very cheap, elastic and fault tolerant storage solution.

2.5 Strong consistency within a data center

Strong consistency guarantees, such as being able to read your own writes, or reading a consistent value irrespective of the replica you read from, was a requirement for the Messages product. In fact, for a wide class of applications, this model is much easier to work with. Strong consistency is another attractive property that HBase provides.

2.6 Experience with HDFS

HDFS, the underlying distributed filesystem used by HBase, provides several nice features such as replication, end-to-end checksums, and automatic rebalancing of data. Additionally, our technical teams already had a lot of development and operational expertise in HDFS from data processing with Hadoop.

3 Pre-flight preparations

HBase was already a reasonably functional and feature rich system when we started experimenting with it in 2009. For instance, it already had features like automatic load balancing and failover, compression, and map reduce integration. However, there was also a fair amount of correctness, reliability and performance work that we needed to do before we could deploy it at scale. In this section, we'll describe some of the early work we did at Facebook on HBase and HDFS in order to get HBase ready for production use.

3.1 Correctness Fixes

HBase keeps its transaction log in HDFS– but HDFS at the time didn't provide a way to guarantee that the contents appended to a file were flushed/synced to multiple nodes. This was one of the reasons why HBase wasn't popular as a storage solution for online user facing workloads with strict data durability requirements. Adding sync support to HDFS, in order to ensure durability of committed transactions, was one of the first features that we worked on.

HBase promises row-level transactional guarantees. An insert involving multiple column families should be done atomically as long as all the records belong to the same row– either all the changes must persist or none. However, the initial implementation of the commit log allowed this property to be violated. We enhanced the commit log implementation to support atomic multi-column family operations.

We also fixed several bugs in the recovery (commit log replay) code, such as a bug that could cause deleted items to be resurrected after a node failure, or cause transactions in the commit log to be skipped because of incorrect handling of transaction sequence ids.

3.2 New/Pluggable HDFS Block Placement Policy

HDFS, the underlying file system used by HBase, provides several nice features such as replication, end-to-end checksums, failover, and storage elasticity. However, the default block placement policy of HDFS, while rack aware, is minimally constrained. The non-local replicas of the different blocks of a file can pretty much end up spanning the entire cluster. To reduce the possibility of losing data if multiple nodes were to fail around the same time, we implemented a new block placement policy in HDFS that allowed us to constrain the placement of the replicas of the blocks of a file within configurable smaller sized node groups. This meant that there were a lot fewer combinations of multiple node failures that could result in data loss compared to before. Using the new block placement policy, we were able to reduce the theoretical data loss probability by about two orders of magnitude.

3.3 Availability

HBase handles node failures well. Except for the HDFS master (aka namenode) failure, the system automatically handles and recovers from failures of other components (such as the HBase master or regionservers, or HDFS datanodes). Given this, we expected the more common reasons for downtime to be maintenance. Since HBase software was still relatively new at that time, we anticipated that we would likely need to push critical bug fixes and improvements routinely to production clusters, and hence implemented rolling upgrades, whereby we could upgrade the software on the cluster in a rolling fashion without taking the entire database down. We also implemented the ability to alter a table schema (e.g., adding new column families or changing the settings of an existing column family) in an online manner. We also made HBase compactions interruptible so that restarts of a server or the entire cluster didn't have to wait for compactions to finish.

3.4 Performance

HBase's Log Structured Merge Tree (LSM) approach for maintaining on-disk data makes it very efficient at writes. On the read-side, however, because multiple, unmerged, immutable, index-organized data files aka HFiles need to be consulted before a read request can be served, in a naive implementation, the overhead on reads is likely to be higher in the general case compared to a traditional RDBMS.

We implemented several performance features to help reduce this read overhead.

- Bloom filters on keys– These are maintained at a per-HFile level, and help reduce the number of HFiles that need to be looked up on a read.
- Timerange hints– We added timerange hints to HFiles, so that timestamp supplied key lookups or time range queries only needed to look up HFiles that contained data for an overlapping time range.
- Seek optimizations– We also fixed some inefficiencies in the earlier implementation that were causing multi column lookups within a row to become full row scans, where instead an index could have been used to efficiently reseek to the desired columns.

3.5 Shadow Testing

Rather than speculating on the kind of issues we would run into when we put the system in production, or limiting ourselves to running synthetic load against test clusters, we decided to invest effort in shadowing/mirroring traffic for a portion of the users from the older generation of the messages and chat products while the new one was still under development. The shadow cluster setup gave us pretty good insights into what to expect under real traffic conditions, and it also served as a useful platform for testing failure scenarios, and evaluating alternate schema designs.

Once the rollout of the new version of the product reached a significant part of our user base, we switched to a new shadowing setup– one that mirrored the new version of the product. This setup continues to remain in place and helps us iterate on software changes rapidly and in a safe manner. All changes go through shadow testing before they are pushed to production.

3.6 Backups V1

HBase had not been used previously, at scale, for critical user facing applications. So during the initial stages, we were concerned about unknown/potential bugs in HBase, especially ones that could cause data corruptions. So we invested in a backup and recovery pipeline implemented outside of HBase. We used Scribe, Facebook's distributed framework for aggregating logs from thousands of servers, to log all actions against a user's mailbox (such as addition/deletion of message, change to read/unread status, etc.) to an HDFS cluster. But Scribe does buffered forwarding, and is not designed for guaranteed delivery. So to greatly minimize the probability of data loss in this pipeline, we double log to Scribe from both the application server and HBase tiers. The two log streams are then merged/de-duped and stored in HDFS clusters in a local as well as remote data center.

4 Mid-air refueling

Once in production, we had to rapidly iterate on several performance, stability and diagnosability improvements to HBase. Issues observed in production and shadow cluster testing helped us prioritize this work. In this section, we will describe some of these enhancements.

4.1 HFile V2

Few months into production, as datasets continued to grow, the growing size of the index started to become a problem. While the data portion in an HFile was chunked into blocks, lazily loaded into the block cache, and free to age out of the block cache if cold, the index portion, on the other hand, was a monolithic 1-level entity per HFile. The index, cold or otherwise, had to be loaded and pinned in memory before any requests could be served. For infrequently accessed column families or old HFiles, this was sub-optimal use of memory resources. And this also resulted in operations such as cluster restarts and region (shard) load balancing taking much longer than necessary. Bloom filters suffered from the same problem as well.

As a temporary measure, we doubled the data block sizes; this halved the size of the indices. In parallel, we started working on a new HFile format, HFile V2– one in which the index would be a multi-level data structure (much like in a traditional B-Tree) comprising of relatively fixed sized blocks. HFile V2 also splits the monolithic bloom filter per HFile into smaller blooms, each corresponding to a range of keys in an HFile. The bloom and index blocks are now interspersed with data blocks in the HFile, loaded on demand, and cached in the same in-memory LRU block cache that is used for data blocks. A single read operation could now result in paging in multiple index blocks, but in practice frequently accessed index blocks are almost always cached.

In traditional databases, such changes are almost impossible to make without a downtime. Thanks to compactions, in HBase, data format changes can be pushed to production and data gets gradually converted to the new format without any downtime, as long as the new code continues to support the old format. But it wasn't before HFile V2 had a lot of soak time in shadow cluster setup and significant compatibility testing that we felt comfortable pushing the code to production.

4.2 Compactions

Compaction is the process by which two or more sorted HFiles are merge-sorted into a single HFile. This helps colocate all the data for a given key on disk, as well as helps drop any deleted data. If a compaction takes all the existing HFiles on disk and merges them into a single HFile, then such a compaction is called a major compaction. All other compactions are "minor" compactions, or referred to below as just compactions.

We usually select 3 files of similar sizes and compact them. If the files are of different sizes, we do not consider them viable candidates for a minor compaction. The first issue we found was the case of inefficient cascaded minor compactions. Consider the case when we find 3 files of a similar size and compact them to a single file of three times the size (3x). It could so happen that there are 2 files existing in the store of a similar size (3x) and this would cause a new compaction. Now the result of that could trigger another compaction. These cascaded set of compactions are very inefficient because we have to read and re-write the same data over and over. Hence we tweaked the compaction algorithm to take into account the sum of the data.

The next issue to hit us was the parameter which allows a file to be unconditionally included in a compaction if its size was below a given threshold. This parameter initially was set to 128MB. What this resulted in was for some column families, which had small files flushed to disk, we would compact the same files over and over again until the new files resulted in a 128MB file. So, in order to get to 128MB, we would write a few GB of data, causing inefficient usage of the disk. So we changed this parameter to be a very small size.

Originally, compactions were single threaded. So, if the thread was performing a major compaction which would take a while to complete for a large data set, all the minor compactions would get delayed and this would bring down the performance of the system. And to make this worse, the major compactions would compact every column family in the region before moving on to any other compaction. In order to improve the efficiency here, we did a couple of things. We made the compactions multi-threaded, so one of the threads would work on a "large" compaction and the other would work on a "small" compaction. This improved things quite a bit. Also, we made the compaction queue take requests on a per-region, per column family granularity (instead of just per region) so that the column family with the most HFiles would get compacted first.

4.3 Backups V2

The V1 backup solution wrote the action logs for the user mailboxes as they arrived into Scribe. To restore a user's data, all the user's action logs had to be read out of scribe and replayed into the HBase cluster. Then, the secondary indices for the user's mailbox had to be regenerated.

As data sizes started growing, the existing backups V1 solution was getting less effective because the size of the action logs that would need to be read and replayed on recovery kept increasing. This made a V2 solution necessary.

The backups V2 solution is a pure HBase backup solution. The backups are split into two parts - the HFile backups (HFiles are HBase data files) and HLog backups (HLogs are HBase write ahead logs). With both of these backed up, and knowing the start and end time of the HFile backup process, it is possible to do point in time restores. Hence, we started taking HFile backups and increased the retention of the HLogs on the production cluster. These backups, which live on the production cluster, are our stage 1 backups. In order to survive cluster failures, we also have stage 2 backups where the data persists in another cluster in the same datacenter. For stage 3 backups, data is also copied to a remote datacenter.

One issue we were hit with pretty quickly while doing in-cluster stage 1 backups was that as the files got larger, we needed more disk iops on the production clusters to do the backups and it was taking longer to do the backups as well. To get around this issue, we modified HDFS to create hardlinks to an existing file's blocks, instead of reading and writing the data to make a copy.

4.4 Metrics

We had started out with a basic set of metrics to being with, for example - HBase get, put and delete ops count per second and latency of each, latencies of the resultant DFS read, write and sync calls, and compaction related metrics. We also tracked system level metrics such as CPU, memory and network bytes in and out. Over time, the need for more insight in order to fix bugs or improve performance made it necessary to add and track various other metrics.

In order to understand how our block cache was doing at a per column family level (so that we can decide not to cache some column families at all or look into possible application bugs), we started tracking per column family cache hit ratios. We also noticed that for some calls, the average times of various operations was low, but there were big outliers - this meant while most users had a good experience, some users or some operations were very slow. Hence we started tracking the N worst times per operation based on machine names so that we could look further into the logs and do an analysis. Similarly, we added a per-get result size so that we can track the size of responses being requested by the application servers.

Once we had HBase backups V2 in production, the total disk usage became an important metric, so we started tracking the DFS utilization as well. In addition, we found that there were quite a few rack switch failures and started tracking those.

These are just a handful of examples of how we added more metrics as we went along.

4.5 Schema Changes

The messages application has complex business logic. In the days leading up to its release and even for a short while afterwards, the product kept evolving. Hence an early decision was made to not invest in a complex schema to store the data in HBase. But loading all the messages into the application server's memory from HBase on a cache miss when a user accessed his mailbox was prohibitively expensive. So the initial solution was to store all the messages of the user in one column family, and maintain an index/snapshot of the in-memory structures required to display the most commonly accessed pages in another column family. This served us well in the days of evolving schema as the snapshot was versioned, and the application server could interpret changes to the snapshot format correctly.

However, as we store messages, chats, and emails, we found that the ”snapshot” column family got several megabytes in size for the active users - both reads and writes were large, and this was becoming a big bottleneck. Hence we had to invest in making the schema finer grained. We went to a ”hybrid” schema where we made the various structures stored in the column families more fine-grained. This helped us update just the more frequently varying data and reduce the number of updates to the infrequently changing data. This helped us make the writes smaller, though the reads were as large as before.

Finally, we looked into the usage patterns of the application and came up with a very fine-grained schema where the reads and writes were done in small units across all the calls. This improved the efficiency of things quite a bit. But since we were changing the entire disk layout of all the data, rolling this out at our scale with no noticeable downtime was a very big challenge. In addition, since the map-reduce jobs that were run to transform the data were very intensive and it was very risky to run the job on the production clusters, we decided to run the transformation jobs on a separate set of machines in a throttled fashion.

4.6 Distributed Log Splitting

To describe what this is briefly, consider the scenario when all or most of the regionservers die (for example, when the DFS namenode is unreachable because of a network partition). Upon restarting the clusters, all the regions would have some entries in the write-ahead logs (HLogs) that would have to be replayed before the regions could be brought online. To do this, all the HLogs belonging to all the regionservers would have to be split and regrouped into edits per region, and the new regionservers opening the regions would have to replay these edits before the regions can be opened to serve reads or take write traffic.

The implementation to achieve this was to have the HBase master list all the logs and split them using a few threads. The issue while running larger clusters - say with 100 regionservers - was that it would take more than 2 hours in order for the single master process to split the logs for all the servers. Since it was unacceptable to go into production in this state, the first and simple version of distributed log splitting was to have a script that would parallelize this across all the regionservers before the cluster was started. This did achieve the intended effect of making the log splitting times in the 5-10 minute range.

The issue with the script-based approach was that it was transparent to the HBase master. This required the cluster to be completely shutdown until the log splitting process was completed, so it was not suitable for the cases when a subset of regionservers (say, the ones on a rack) exited because they were partitioned from the DFS. Hence we implemented an automatic distributed log splitting feature into the HBase master itself - where the HBase master uses Zookeeper as a task queue and distributes the job of log splitting to all the regionservers. Once the log splitting is done the regions are opened as usual.

4.7 Data locality

To be available in spite of rack failures, we do not want all the replicas of a file to be on the same rack. For a very data intensive application, like messages, this causes a lot of network traffic between racks. As the data size grew larger, the network traffic kept increasing until it was causing the top-of-rack to become the bottleneck. So, to reduce network bandwidth during reads, locality of data became extremely important. In order to maintain locality of data, the regions have to assigned to the region servers that contained most of the data on the local disk. We solved this by computing the machine with the most data locality for each region at startup time and assigned the regions accordingly at the HBase startup time.

4.8 Data Block Encoding

In several applications, including Facebook messages, there is a lot of redundancy between adjacent records (KVs) within a data block. And hence it is desirable to have an in-memory format for blocks that takes advan-

tage of this; a format that allows us to pack more records per block, yet one in which records can be efficiently searched. We implemented a "data block encoding" feature in HBase, which is a lightweight, pluggable compression framework that's suited for in-memory use. It increases the holding capacity of the block cache, without much impact on performance. We have implemented a few of the encoding schemes, and we're seeing block cache capacity wins of up to 1.1x-5x. On disk too, now that we encode the blocks first and then compress, these schemes are able to add about 1.1x-2x savings on top of LZO.

4.9 Operational aspects and issues

Failure is the norm when running large-scale clusters. Hence disk failures, machine failures, per-node network partitions, per-rack network failures and rack switch reboots were all recurring events. We wanted to automatically detect and handle the more frequent of these. Early on, we had decided that we should automatically handle most of the single node issues. Hence we had monitoring scripts that would disabled and send to repair those machines which were not network reachable for an extended period of time, those which had root-disk failures, and so on. In fact, the process of machines being retired and reclaimed from the repair pool is a constant one, with about 2-3% of the machines always in the queue.

We had a lot of alerts that paged the relevant people on a variety of failure scenarios. For example, if the number of errors observed by the application server exceeded a certain steep threshold then it triggered alerts. In addition, if the error rate was not steep enough to trigger alerts but stayed persistently over a time window, then too an alert would get triggered (prolonged errors even if lower in volume). Similarly, the master nodes ("controller" nodes as we call them) would alert on issues, as they needed a higher availability than the other nodes. In addition, there is a HBase health checker that is run constantly and would alert if it fails the health checks.

In the light of issues we experienced, operator error is a non-trivial cause of some of the issues we faced. There was one such incident where upon accidentally unplugging a power cable caused about 60% of the nodes in the cluster to go down. The cluster was restored in a few minutes and there was no data loss.

5 New use cases

As HBase matures, its usage inside Facebook, for both serving and user-facing workloads as well as real-time and batch processing workloads, has grown. In this section, we will describe some of these new use cases.

5.1 Puma: Real-time Analytics

Puma is a stream processing system designed for real-time analytics. Using Puma, content publishers/advertisers can, for example, get real-time insights into the number of impressions, likes and unique visitors per domain/URL/ad, and a breakdown of these metrics by demographics. Such analytics was previously supported in a periodic, batch-oriented manner using Hadoop and Hive, and hence available for consumption only several hours later. With Puma, data is collected in real-time via Scribe, read via ptail and processed/aggregated by Puma nodes and written out to HBase. Puma nodes also act as a cache for serving the analytic queries, deferring any cache misses to HBase.

5.2 Operational Data Store (ODS)

ODS is Facebook's internal monitoring system which collects a variety of system and application level metrics from every server for real-time monitoring, anomaly detection, alerting and analysis. The workload, consisting of several tens of millions of timeseries data, is pretty write intensive. More than 100B individual data points are written per day. And the reads, both user-issued and from time rollup jobs, are mostly for recent data.

ODS was previously implemented on top of MySQL, but recently switched to an HBase backend implementation for manageability and efficiency reasons. The constant need for re-sharding and load balancing existing shards, as the data sets grew in unpredictable ways, caused a lot of operational inconvenience in the MySQL setup. Also, from an efficiency point of view, HBase is a really good fit for this write intensive, read-recent type of workload. HBase provides a much higher throughput because it converts random record updates into sequential writes at the disk subsystem level. And, because HBase naturally keeps old and newer data in separate HFiles, for applications like ODS, which mostly read recent data, the data can be cached a lot more effectively.

ODS makes heavy use of the HBase's tight integration with MapReduce for running the time-based rollup jobs periodically. ODS also uses HBase's TTL (time-to-live) feature to automatically purge raw data older than a certain number of days. There is no longer a need to implement and manage additional cron jobs for this purpose.

5.3 Multi-tenant key/value store

Several applications within Facebook have a need for a large-scale, persistent key/value store. A per-application key/value store backed by HBase is now offered as a service within Facebook and some of the applications now use this solution for their storage needs. A shared HBase tier provides controlled multi-tenancy for these applications, and relieves the application developers from the usual overheads associated with capacity planning, provisioning and infrastructure setup, and scaling as their data sets grow.

5.4 Site integrity

Facebook's site integrity systems gather data and perform real-time analytics to detect and prevent spam and other types of attacks. One of these systems now uses an HBase backed infrastructure to track new user registrations and surfaces several fake accounts every day. HBase was chosen because it provides high write throughput, elasticity, a data model that allows running efficient scans on lexicographically sorted row keys, and primitives that support aging out old data automatically.

5.5 Search Indexing

HBase is being used to implement a generic framework to build search indices across many verticals. Multiple document sources are fed into HBase constantly and asynchronously. HBase stores the document information, and is used to join the information coming from the various sources for each document. On demand, per-shard search indices are built using MapReduce to read the relevant data from HBase. The index servers then load these shards.

6 Future work

HBase is still in its early stages, and there is a lot of scope and need for improvement.

On the reliability and availability front, HDFS Namenode HA (making HDFS tolerate namenode failures), better resilience to network partitions, data locality aware node failover and load balancing of regions, rolling restarts for HBase/HDFS during software upgrades, and fast detecting single node failures are some of the current focus areas.

We are also working on cross data center replication and point-in-time restore from backups so that every application does not have to implement its own custom solution. Native support for hardlinks in HDFS is being implemented to enable super fast snapshotting of data in HBase. On the efficiency front, we are looking at keeping inlined checksums for data to reduce the disk overheads of verifying checksums, improving HBase's

ability to make optimum use of large memory machines, reducing HBase's reliance on OS page cache, adding support for compressed RPCs, optimizing scan performance, commit log compression and so on.

As new applications for HBase emerge, new requirements continue to arise. Better support for C++ clients, improved Hive/HBase integration, support for secondary indices and coprocessors and adapting HBase for use in mixed storage (flash+disk) or pure flash environments are also things on the radar.

7 Acknowledgements

We would like to thank our HDFS engineering, HBase operations and Messaging teams, as well as the Apache HBase community without whose help and involvement, a lot of this work would not have been possible.

References

[1] Apache HBase. http://hbase.apache.org.

[2] Apache HDFS. http://hadoop.apache.org/hdfs.

[3] Kannan Muthukkaruppan. The Underlying Technology of Messages. http://www.facebook.com/-note.php?note_id=454991608919, Nov. 2010.

[4] Dhruba Borthakur et al. Apache Hadoop Goes Realtime at Facebook. In *Proc. SIGMOD*, 2011.

[5] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. The Google File System. In *Proc. SOSP*, 2003.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, 2004.

[7] Fay Chang et al. BigTable: A Distributed Storage System for Structured Data In *Proc. OSDI*, 2006.

[8] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX*, 2010.

[9] Alex Himel. Building Realtime Insights. http://www.facebook.com/note.php?note_id=10150103900258920.

[10] Joel Seligstein. Facebook Messages http://www.facebook.com/blog.php?post=452288242130, 2010.

[11] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). In *Acta Inf.* 33(4): 351-385 (1996).

[12] Eugene Letuchy. Facebook Chat. https://www.facebook.com/note.php?note_id=14218138919.

[13] Scribe. https://github.com/facebook/scribe/wiki.

Experiences with using Data Cleaning Technology for Bing Services

Arvind Arasu, Surajit Chaudhuri, Zhimin Chen, Kris Ganjam, Raghav Kaushik, Vivek Narasayya
Microsoft Research
{arvinda,surajitc,zmchen,krisgan,skaushi,viveknar}@microsoft.com

Abstract

Over the past few years, our Data Management, Exploration and Mining (DMX) group at Microsoft Research has worked closely with the Bing team to address challenging data cleaning and approximate matching problems. In this article we describe some of the key Big Data challenges in the context of these Bing services primarily focusing on two key services: Bing Maps and Bing Shopping. We describe ideas that proved crucial in helping meet the quality, performance and scalability goals demanded by these services. We also briefly reflect on the lessons learned and comment on opportunities for future work in data cleaning technology for Big Data.

1 Introduction

The aspirational goal of our Data Cleaning project at Microsoft Research [6] (started in the year 2000) is to design and develop a domain independent horizontal platform for data cleaning, so that scalable vertical solutions such as address cleansing and product de-duplication can be developed over the platform with little programming effort. The basis for this goal is the observation that some fundamental concepts such as textual similarity and need to extract structured data from text are part of many data cleaning solutions. Our platform is designed to capture and support these concepts. The technologies that we have developed include a customizable notion of textual similarity along with an efficient lookup operation based on this similarity called Fuzzy Lookup, a clustering operation called Fuzzy Grouping, and an Attribute Extraction (aka Parsing) operation for extracting structured attributes from short text (e.g. extract attributes of the product item such the Brand, Product Line, Model from a product title).

Fuzzy Lookup and Fuzzy Grouping have shipped in Microsoft SQL Server 2005 [3]. Subsequently, as Bing started offering vertical services such as Bing Maps [4] and Bing Shopping [5], we became aware of several data cleaning challenges they faced. Over the past few years, we have worked closely with these teams in Bing to address some of the important data cleaning challenges that arise in their services. This engagement with Bing has helped shape the evolution of our Fuzzy Lookup and Attribute Extraction technologies and serves as an illustration that a domain independent horizontal platform for data cleaning can be leveraged relative easily by developers of vertical solutions. Two major principles underlying the design of our data cleaning technologies are customizability and performance at scale; and as we describe in this article, both are crucial in order to meet the ”Big Data” challenges of Bing. We also reflect briefly on the lessons learned and comment on potential opportunities for future work in data cleaning for Big Data.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Scalable Approximate String Matching for Bing

Scalable approximate string matching (fuzzy matching) is an essential problem for Bing. The need for approximate string matching in Bing arises in a variety of online and offline settings. For example, when a user submits a query to Bing Maps, locations (e.g. city, address, landmark) that are mentioned in the query need to be matched approximately against a large dataset of known locations. Need for offline fuzzy matching arises in many entity resolution scenarios involving comparing a big data set with a big entity reference table. In this section, using Bing Maps as the motivating scenario, we first describe the requirements we encountered for online fuzzy matching and key technical ideas that proved essential is meeting these requirements. Next, we discuss offline fuzzy matching that we implemented on top of Microsoft Cosmos/SCOPE [9] (a distributed data processing platform developed and used by Bing) and demonstrate the scalability of these techniques using de-duplication as the motivating scenario.

The Fuzzy Lookup technology what we developed provides efficient and scalable fuzzy matching for both these scenarios. For the online fuzzy matching scenario of Bing Maps, Fuzzy Lookup is implemented as a DLL that can be linked into the server code, whereas for the offline scenarios, it is implemented as a SCOPE script. Although these implementations differ, we note that in both cases, the notion of customizable similarity exposed to applications is the same.

2.1 Online Fuzzy Matching in Bing Maps

We begin with a brief overview of Bing Maps and its online fuzzy matching requirements; and then highlight a few key technical and architectural considerations that proved important.

2.2 Bing Maps Overview and Fuzzy Matching Requirements

Overview: Bing Maps is a web-based platform providing a variety of geospatial and location-based services including interactive maps, satellite imagery, real-time traffic, directions and task-oriented vertical portals. Users typically begin their interaction with these services by entering a text-based query for a specific point-of-interest (POI) such as a street address or named entity such as a business, landmark, city or region. Given a user query, the system matches it against a large data set of POIs and then returns information which it believes to be most relevant, for instance, repositioning the interactive map to display a specific location.

Customizable Similarity: There are several challenges involved in determining the intent of a user query and identifying the appropriate POI. Queries, such as a street address, are often complex strings containing substructure like street number, street name, city, state and zip code. Users often make spelling mistakes, use various conventions for abbreviating strings, or in general represent a given entity in a manner different from that present in the database of POI. Moreover, different countries or regions often require different sets of abbreviations etc. Finally, the query itself may be ambiguous and match many different reference entities with varying degrees of similarity.

Performance at scale: The worldwide POI database consists of upwards of 100 million entities. It continues to grow as new classes of people, places and other broad categories of entities are supported by the platform. Each entity has a primary textual representation, but may also include several alternative forms such as abbreviations or synonyms. The services of the system are localized for various languages and regions, so this data set is amenable to partitioning by region. Nevertheless, regions having upwards of 20 million entities are not uncommon, and thus any approximate string matching solution must be highly performant at this scale. Since, for a given query, fuzzy matching may be invoked multiple times (since a query may be parsed in different ways), and the query has strict overall latency SLAs, each invocation of fuzzy matching needs to complete in about *3 msec* on average with a worst-case latency not exceeding *15 msec*.

Low memory footprint: To achieve high availability and dynamic load balancing, the service is designed to be relatively stateless, with any necessary storage of user-data existing in a separate tier. This stateless model allows virtual machine (VM) images to be replicated across compute clusters to dynamically load balance against changes in user demand for the services. If demand picks up, new instances of the VM images can be created on additional machines. To allow efficient load balancing, the VMs are defined to be of fixed capabilities in terms of memory usage and CPU power. For example, a typical VM configuration is 4 GB or 8 GB putting tight constraints on the memory budget of any fuzzy matching solution. In particular, this requires the indexes used for fuzzy matching to be as compact as possible, while still allowing the performance goals to be met.

2.3 Key Ideas and Techniques

Transformation Rule based Set Similarity: In light of the system and design constraints above, it was important to find a suitable measure of string similarity that could provide sufficient precision/recall (quality) while at the same time being amenable to indexing techniques that would enable the performance and scalability goals to be met. The transformation rule based set similarity function (see [2] for details) that we implemented in Fuzzy Lookup enabled these rich kinds of domain knowledge to be expressed within our framework. The notion of transformation rules was particularly important for Bing Maps. It is common for street names to have alternate forms. For instance, in ZIP code 33027, *SW 154th Ave* is also known as *Lacosta Dr E*. In this case, it is important that the rule be contextual and apply only when we know that the record refers to a particular ZIP code.

The matching framework we designed is an extensible architecture which allows arbitrary transformation providers to be plugged in. A transformation provider can be based on static rules from a file or can be generated dynamically on-the-fly programmatically. In addition to nearly 100,000 pre-defined transformations such as the examples above, Bing Maps utilizes several additional dynamic transformation providers to provide spelling mistake correction and string splitting/merging. For instance, looking up a query substring against a dictionary of words known to be in the reference records and suggesting corrections on-the-fly. The example shown in Figure 1 illustrates transformation rule based set similarity. The two addresses being matched are: LHS: "*SW 154th Ave Florida 33027*" and RHS: "*Lacosta Dr E FL 33027*". Logically, all relevant transformation rules are applied to the LHS and the RHS, producing two variants on each side. Then among the cross-product of these variants, the pair of strings with the highest set similarity (i.e. *Jaccard* similarity) is chosen. In this example, the variants are in fact identical, so the overall similarity score between the two input strings in the presence of the transformation rules shown is 1.0.

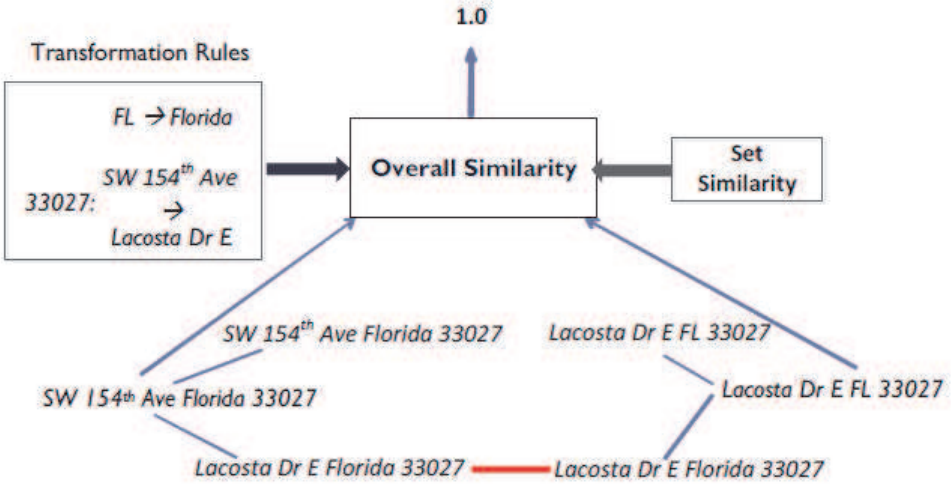

Figure 1: Transformation Rule based Set Similarity

Locality Sensitive Hashing (LSH): One important reason for having Jaccard similarity as the base similarity function is that we are able to very efficiently index this function. In particular, we exploit locality-sensitive hashing (LSH), which is a well-known signature scheme for Jaccard similarlity. Using this signature scheme we obtain a probabilistic guarantee of finding all the matches above a given similarity threshold. Another key advantage of this indexing technique compared to an inverted index based method is that we could precisely tune the space/time trade-off to put a strong bound on the average lookup time. One of the non-trivial challenges we addressed was to engineer LSH so that the blowup caused by transformation rules does not hurt performance.

Additional performance optimizations: While computation of similarity in the presence of transformations is efficient for a relatively small number of transformation rules, performance does become an issue when there are a very large number of rules. The scale and tight performance requirements of Bing Maps required the addition of a rule pruning step to select only a limited number of the best rules. An important technique used was a Bloom filter over adjacent token pairs in the reference table. Precedence is given to a transformation which transforms a pair of adjacent tokens into a pair which actually occurs in the reference table. We also found that, for certain transformation providers (e.g. Edit transformation provider that dynamically generates transformation rules), pre-computation and caching techniques helped avoid repeated expensive work, such as performing an edit distance lookup of a word against a very large dictionary of words, and brought down the end-to-end lookup time considerably. We also used lightweight compression of string data to help reduce the overall memory footprint of the matching service. Finally, for computing pairwise similarity between a pair of records in the presence of transformation rules, the bipartite matching based algorithm ([2]) greatly improved performance.

Trade-offs of managed code implementation: Our components were written fully in managed C# and .NET. This afforded rapid software development and reliable, easy-to-maintain code. We did have to be cognizant of the garbage collector to ensure that full collections did not take too long. This was achieved through allocation of objects having long lifetimes into more compact data structures having fewer object handles visible to the garbage collector. With improvements to asynchronous GC techniques in newer releases of the CLR, we believe that this will increasingly become less of a consideration.

2.4 Offline Fuzzy Matching on Cosmos (Bing's MapReduce Engine)

While online approximate string matching is a necessary component of modern web services, equally important are the offline pipelines which ensure that the underlying reference data is of high quality. Bing ingests data feeds from a variety of data providers and data sources which have varying degrees of completeness, consistency and overall data quality. The same entity may be represented in different ways across the different data sources and mentions must be resolved to remove duplicates. Thus in order to identify duplicates a necessary step is to find all pairs of records that exhibit a similarity above a pre-defined threshold, i.e. it is a self-join scenario.

There are many commonalities between the online and the offline matching processes. Instead of low latency driving the design choices, in the offline world, it is the scale of the data which requires that the matching solution to be very fast. In both cases the same desiderata of a domain-independent yet highly customizable framework still applies. We found that our techniques developed for the online world have many properties that make them equally suitable for the offline world and exhibit very nice scaling properties. Much of the offline processing in Bing is performed on a massively parallel and distributed MapReduce architecture known as Cosmos, with a higher level SQL like language on top referred to as SCOPE [9]. The data and computation for a given large processing pipeline is typically distributed across thousands of nodes. In this setting, the main challenges for the problem of entity resolution between data sources include reducing the number of comparisons between candidates, reducing the overall data movement and subdividing the problem so as to maximize the amount of parallelism.

A frequently used strategy for identifying similar entities between data sources is to derive one or more blocking keys for each entity and then perform pairwise comparisons between all entities which share a given

blocking key. This approach has the advantage that the work for each distinct key can be parallelized. In fact, the signature scheme developed in the online case uses this same principle except the input arrives one at a time instead of in batch. From a query processing perspective, we observe that what is needed is an algorithm for performing an equi-join of candidate records sharing a given signature. In a MapReduce system, this translates to mapping the records on their signatures and having each reducer output the pairs of records which truly satisfy the similarity predicate. LSH signature scheme approaches have the nice properties that, not only do they (probabilistically) guarantee that all matches will be found, the signature parameters can be tuned to control the size of each block, allowing effective means to optimize the match operation to the bandwidth and CPU characteristics of the underlying compute cluster. Using these techniques, we were able to find all approximate duplicates in a table of 1 billion organization name and address rows in less than an hour. Figures 2 and 3 show how our offline fuzzy matching technique scale with the the data size and number of nodes (machines). Finally, we also note that recent work on fuzzy joins on MapReduce ([7][8]) use similar techniques.

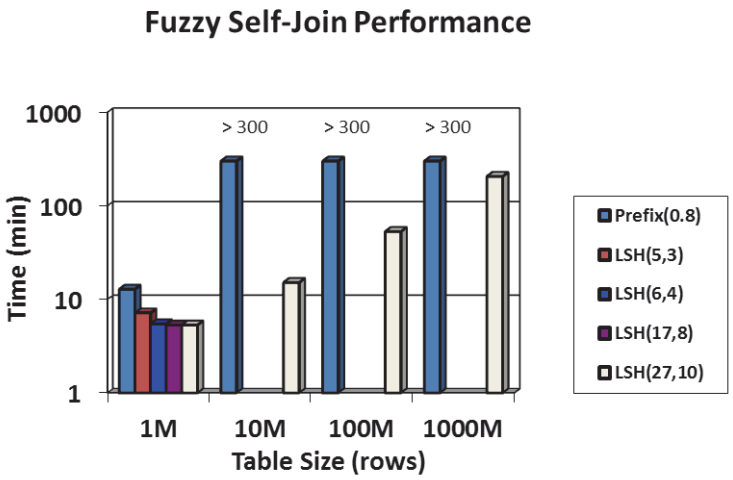

Figure 2: The figure above illustrates the performance of various signature scheme settings for the offline task of finding all approximate duplicates in tables of various sizes. We found that prefix filtering did not scale very well. LSH with 27 hash tables over 10 dimensional signatures performed best and exhibits good scaling characteristics.

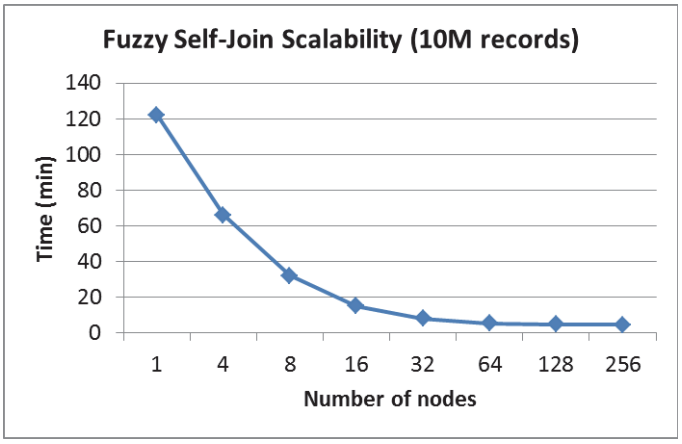

Figure 3: For the offline task of finding all pairs of fuzzy matches in a table using a fixed signature scheme setting we see significant speedup with number of nodes with diminishing returns beyond a point.

3 Attribute Extraction and Dictionary Augmentation in Bing Shopping

Parsing (or attribute extraction) is the process of extracting structured attribute values from short textual input. Attribute extraction is an important operation that is common in several data cleaning tasks. As mentioned earlier, we have developed an Attribute Extraction operator as part of our data cleaning platform. This operator relies on dictionaries for each attribute that defines what values are allowable for that attribute. For example for the Brand attribute of a Digital Camera product category, the dictionary may specify that the allowable values are: {Canon, Panasonic, Nikon, ...}. For numeric attributes (such as Megapixel), our Attribute extraction operators allows a set of regular expressions to define the allowable patterns. We use Bing Shopping as the motivating scenario in the description below, although our techniques are general and can be applied to other domains as well.

3.1 Bing Shopping Overview and Requirements

Overview: Bing Shopping is the commerce search vertical of the Bing search engine. It uses a referral model: when a user searches for a product, it suggests merchants with offers for that product. The service aggregates product catalogs, reviews and offers from external feeds and serves queries that are detected with commerce intent. One of the core components of the Bing Shopping backend is a master product catalog service that integrates product catalogs from different product data providers into a single master product catalog. The goal of the master product catalog is to provide a complete and clean reference of product information for other components in Bing Shopping such as offer matching, review matching, query rewriting, and to enable rich product search experience such as faceted search. Towards this end the master product catalog not only comprises all the products but also defines a taxonomy including a product category hierarchy, attributes for each category and their synonyms, and legitimate values (i.e. dictionaries) for each attribute and their synonyms. The architecture of the data processing pipeline of the master catalog service is shown in Figure 4. The three major processes are (1) Categorization to classify products into Bing Shopping's master category hierarchy (2) Enrichment, which in itself is also a pipeline including tasks such as mapping provider specific attributes to predefined master attributes, extracting attribute values from product titles, normalizing attribute values and filling nulls (3) De-duplication to group duplicates and to choose masters among duplicates.

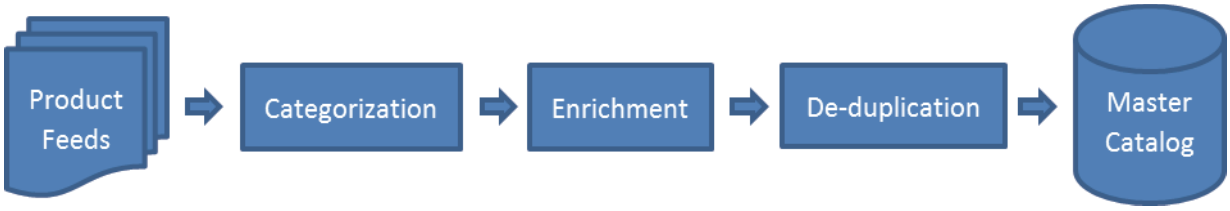

Figure 4: Data Processing Pipeline of Bing Shopping's Master Catalog Service

Need for accurate parsing: When an external data provider (e.g. a manufacturer of a product, a merchant who is selling a product) provides data to Bing Shopping, there is no standard data format. A significant number of products arrive into the pipeline as free text in a single Title attribute. Therefore a key challenge is to parse attribute values from free text for a given product item/offer.

Need for accurate and complete dictionaries: A challenge closely related to attribute extraction is to build complete and accurate dictionaries for each attribute. High quality dictionaries can not only improve accuracy of attribute extraction significantly, it can also help in other downstream uses such as query rewriting and faceted search.

Scale: Scale of data presents another dimension of challenge. Bing Shopping's master catalog contains more than 30 million products and is growing very fast. Thus, operations such as Attribute extraction and De-

duplication must run at scale. A second scale challenge arises with the taxonomy that contains more than 3000 categories and more than 44 attributes per category on average at the leaf level. Some attributes such as the Model Number attribute contains thousands of legitimate values per category. Building complete and accurate dictionaries for so many attributes presents a huge challenge.

The rest of section 3 will discuss attribute extraction and dictionary creation in further details because of their particular importance.

3.2 Attribute Extraction

Overview: Attribute extraction is the process to extract attributes of product from its title. Table 1 shows some examples of product titles. The Bing Shopping team defines important attributes and their properties such as whether it is a numeric attribute or not for each category. For example, Brand, Product Line, Model, Resolution, Color, Optical Zoom, Digital Zoom, etc. are defined for Digital Camera category. Attribute extraction also takes each attribute's dictionary as input. The output is attribute value pairs for each product.

Product Titles
Fuji FinePix Z200FD 10.0 MegaPixel Digital Camera - Silver
Panasonic Lumix DMC-FX33W 8.1 MegaPixel 3.6X Optical/4X Digital Zoom Digital Camera (White)
Casio Exilim Zoom EX-Z300 Digital Camera, 10.1 MegaPixel, 4X Optical Zoom, 4X Digital Zoom, 3" LCD, Pink
...

Table 1: Sample Product Titles of Digital Camera Category

Dictionary/Pattern Match: Attribute extraction behaves slightly differently for categorical and numeric attributes. For a categorical attribute it relies upon dictionary matching. For example, assuming the dictionary for the Brand attribute is Fuji, Panasonic, Casio, matching this dictionary will output Fuji as Brand for the first product in Table 1. For numeric attributes it is pattern matching. For example, assuming the pattern for the Resolution attribute is \d+[\.\d+] MegaPixel (Section 3.3.1 will discuss how to generate these patterns), matching it will output 10.0 MegaPixel as Resolution for the first product in Table 1.

There are two points worth noting with the above approach. First, it assumes the dictionaries have good coverage. For example, if Panasonic is not in the Brand dictionary, then extraction of Brand attribute fails for the second product in Table 1. Section 3.3 will discuss a solution to this problem. Second, it assumes the dictionaries are mostly unambiguous, i.e., dictionaries do not overlap and when there is a dictionary match, it is a true positive match. While this assumption can lead to false positives in a general setting of information extraction from free text, it is relatively safe in this context, especially when applying the dictionaries on a per category basis. For example, while there are a lot of false positives to extract every "Apple" as a laptop brand in general, it is acceptable to extract every "Apple" as Brand in the titles of products of laptop category.

Disambiguation: When dictionaries overlap, there is a disambiguation phase to resolve the conflicts. We currently employ a rule based approach for disambiguation. For example, if one extraction is contained in the other, then the longer extraction wins. If one extraction is next to its attribute name while the other is not, the extraction next to its attribute name wins. For instance, token 4X in the second product in Table 1 matches the pattern for attribute Optical Zoom and attribute Digital Zoom, but since it is next to attribute name Digital Zoom, so 4X is extracted as Digital Zoom. If two numeric attributes have distinct range, then the extraction that is closer to the mean value prevails. For example, using this rule correctly extracts 4GB as the attribute for Main Memory and 500GB as Hard Disk Capacity from product title "Toshiba L645DS4037 14" NB/AMD Phenom P820/4GB/500GB/Red/Case". If extraction involves Brand/Product Line/Model, we prefer extractions that are consistent: i.e., it conforms to the dictionary hierarchy. For example, FinePix under Fuji and Z200FD under FinePix. In the future, we plan to investigate the use of some sequential model like Conditional Random Fields (CRFs) to incorporate dictionary matching and the disambiguation rules together.

3.3 Dictionary Augmentation

Motivation: As discussed in section 3.2 attribute extraction assumes that dictionaries of good coverage are available, but in reality these dictionaries can be rather incomplete, e.g. most of the tail Product Lines and Models may be missing. Since complete dictionaries can be large (e.g. there are thousands of digital camera models), in practice it is very expensive to create dictionaries entirely manually. On the other hand as evidenced in Table 1, there exist a lot of regularities and repetitions in product titles that can be leveraged to auto-suggest additional dictionary entries.

Kinds of dictionaries: We observe that there are three types of attributes: hierarchical, flat (numeric) and flat (categorical). The most notable attribute hierarchy is formed by Brand, Product Line and Model. They are also the most important attributes except for a few categories. Thus, obtaining an accurate and complete list of Brands, Product Lines and Models, and their hierarchical relationships can be very valuable. They also have notable regularities: they tend to have strong correlation between parent and child by the nature of hierarchy relationship; they tend to locate together in the product titles because merchants/vendors are accustomed to using the brand-line-model path to denote a product; if models under the same product line are composed of alpha-numeric characters, they tend to have similar alpha-numeric patterns, for example, Canon EOS 450D, Canon EOS 500D, etc. Flat numeric attributes also have regularity that can be easily leveraged: they form similar alpha-numeric patterns, for example, 8.1 MegaPixel, 10.1 MegaPixel; if its attribute name is also present in product title, they tend to locate next to each other, for example, 4X Digital Zoom. Flat categorical attributes have regularity too but they are less reliable than the other two cases: if both attribute name and value are present in product title they appear next to each other; sometimes data provider put attributes in the titles in a fixed order, but often they do not (we do not discuss this kind of dictionary in more details in this article).

3.3.1 Key Ideas

Semi-supersived approach: Our dictionary augmentation tool uses a semi-supervised learning approach. The Bing Shopping team provides a few example values for each flat attribute and the top level attribute of a hierarchy, and optionally the partial path or full path of the hierarchy. Table 2 and Table 3 show some example inputs. The tool generates new values from these examples. The Bing Shopping team can then review the generated dictionaries, confirm or reject some of the suggested entries or adds new examples and re-run the tool if needed.

Example	Attribute
Fuji	Brand
10.0 MEGAPIXEL	Resolution
White	Color
4X	Digital Zoom

Table 2: Examples of flat attributes and hierarchy top attributes

Brand	Product Line	Model
Fuji	FinePix	Z200FD

Table 3: Examples of hierarchical attributes

Generating candidate regular expressions for numeric attributes: The tool generates new values for numeric attributes by iteratively generalizing the numeric parts and the context part in the example values. It first generates patterns from examples by generalizing the numeric parts, for example, 10.0 MegaPixel becomes pattern \d+[\.\d+] MegaPixel. Matching the titles with the pattern results in additional numeric values, for example, 8.1 MegaPixel. In turn, we then use high support numeric values, for example, 8.1 MegaPixel to find

additional fragments of title with high similarity. For example, if 8.1 MegaPixels and 8.1 MP are both frequent, and if their contexts also agree frequently enough, then a new pattern \d+[\.\d+] MP is generated.

Generating candidate hierarchical dictionary entries by correlation analysis: We could use correlation of tokens to identify potential candidate entries, but the challenge is that dictionary entries are often not single tokens. On the other hand, simplistic techniques like finding maximal windows of correlated tokens are inadequate since such windows could span multiple attributes. We use the following ideas to address these issues. We use the Brand-ProductLine-Model hierarchy as example. First we rely on a set expansion technique such as SEISA [12] to generate new brands from the given examples. We partition the product items by Category and then by Brand, and look for regularities only within each Brand; this significantly improves precision of ProductLine, Model dictionaries. Another key idea is the notion of anchor tokens, i.e. tokens that are both frequent and highly correlated with Brand in each Brand partition. For example, if 474 titles have brand Panasonic and 503 titles includes token Lumix in the Digital Camera category, then $correlation(Lumix) = 474/503 = 0.94$. If both 503 and 0.94 are higher than predefined thresholds then Lumix is used as an anchor token for partition Panasonic. We use correlation among neighboring tokens as the basis to build larger candidate fragments from anchor tokens. For example, if correlation(FinePix, Z200FD) is high and significantly larger than correlation(Z200FD, 10.0), we merge them to a single candidate FinePix Z200FD for Line-Model. We also found that further filtering of these candidates was crucial for obtaining high precision, e.g. filtering based on proximity to the Brand token. Finally, these candidates need to be split into ProductLine and Model. An important observation that drives this step is that shared prefix/suffix usually denotes a boundary between attributes, for example, EOS 450D and EOS 500D.

We evaluated the dictionaries generated for the Brand-ProductLine-Model attribute hierarchy for the a few categories. Table 4 shows the result. The precision is estimated by randomly sampling paths from the output dictionary hierarchy and counting how many of them are correct. The recall is estimated by randomly sampling product titles and counting how many brands, product lines and models are output by the algorithm. This tool is used by the Bing Shopping team and it has greatly reduced the manual effort necessary to build these dictionaries.

Category	Number of Brand-Line-Models Generated	Precision	Recall
Digital Camera	2350	90%	70%
Shoes	27225	80%	50%
Mattress	920	75%	60%

Table 4: Precision and Recall of Generated Dictionaries

4 Conclusion and Future Work

The need for data cleaning arises in a multitude of scenarios beyond those traditionally cited in the enterprise data integration context. In this article, we described our experiences with both online and offline data cleaning challenges in Microsoft's Bing services such as Bing Maps and Bing Shopping. Our data cleaning technologies are today incorporated into these services and has significantly improved their data quality. Since Bing uses COSMOS (a MapReduce engine) for its offline data processing needs, our offline data cleaning technologies have been implemented on this platform and engineered for large scale. Our online fuzzy matching technology required significant new algorithmic and engineering effort in order to meet the performance, memory and scale requirements of Bing Maps.

Finally, we briefly comment on opportunities for future work. One of the issues that any horizontal platform for data cleaning faces is to obtain the necessary domain knowledge for a particular vertical solution. By default, this burden falls on application developers. However manually generating (or purchasing) this knowledge can be expensive, tedious and error-prone. One important example of this is the need for synonyms for fuzzy matching. Therefore tools that can auto-suggest synonyms for entities (such as locations or products) would be valuable.

Examples of early work in this direction include [10][11]. A second issue that came up in our Bing Maps experience was that tuning the parameters of LSH to find the right balance between performance and memory was manual and based on trial-and-error. A tool that could assist developers in tuning these parameters automatically could have saved us significant effort. Last but not the least, performance and scalability challenges arise in each of these data cleaning problems for Big Data, and new techniques may be necessary depending on the desired performance/scale characteristics of the specific problem.

5 Acknowledgments

We thank Jai Chakrapani and Brad Snow from the product team who collaborated closely with us on leveraging our Fuzzy Lookup technology for the Bing Maps service. They helped characterize the requirements for fuzzy matching and were instrumental in integrating the technology into production code. We are grateful to Meera Mahabala, Prakash Sikchi, Shankar Raghavan, David Talby, Bodin Dresevic, Eduardo Laureano, Farshid Sedghi, Tsheko Mutungu, Nikola Todic and James Russell from the Bing Shopping team for their involvement in the attribute extraction, de-duplication and dictionary augmentation efforts. Finally, we take this opportunity to thank Venky Ganti and Dong Xin who have contributed significantly to the data cleaning technology developed at Microsoft Research.

References

[1] Arvind Arasu, Surajit Chaudhuri, Zhimin Chen, Kris Ganjam, Raghav Kaushik, Vivek R. Narasayya: *Towards a Domain Independent Platform for Data Cleaning*. IEEE Data Eng. Bull. 34(3): 43-50 (2011)

[2] Arvind Arasu, Surajit Chaudhuri, Raghav Kaushik: *Transformation-based Framework for Record Matching*. ICDE 2008: 40-49

[3] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, Rahul Kapoor, Vivek R. Narasayya, Theo Vassilakis: *Data cleaning in microsoft SQL server 2005*. SIGMOD Conference 2005: 918-920

[4] Bing Maps. `http://www.bing.com/maps`

[5] Bing Shopping. `http://www.bing.com/shopping`

[6] Data Cleaning Project at Microsoft Research. `http://research.microsoft.com/en-us/projects/datacleaning/default.aspx`

[7] Foto Afrati, Anish Das Sarma, David Menestrina, Aditya Parameswaran, Jeffrey Ullman: *Fuzzy Joins Using MapReduce*. In Proceedings of the conference on International Conference on Data Engineering (ICDE), Washington, USA, April 2012.

[8] Rares Vernica, Michael J. Carey, Chen Li: *Efficient parallel set-similarity joins using MapReduce*. SIGMOD Conference 2010: 495-506

[9] R. Chaiken, B. Jenkins, P.A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. *Scope: Easy and efficient parallel processing of massive data sets*. In Proceedings of VLDB Conference, 2008

[10] Kaushik Chakrabarti; Surajit Chaudhuri; Tao Cheng; Dong Xin, *A Framework for Robust Discovery of Entity Synonyms*, ACM SIGKDD 2012.

[11] Arvind Arasu, Surajit Chaudhuri, Raghav Kaushik: *Learning String Transformations From Examples*. PVLDB 2(1): 514-525 (2009)

[12] Yeye He, Dong Xin: *SEISA: set expansion by iterative similarity aggregation*. WWW 2011: 427-436

Declarative Systems for Large-Scale Machine Learning

Vinayak Borkar[1], Yingyi Bu[1], Michael J. Carey[1], Joshua Rosen[2],
Neoklis Polyzotis[2], Tyson Condie[3], Markus Weimer[3] and Raghu Ramakrishnan[3]
[1]University of California, Irvine, [2]University of California, Santa Cruz,
[3]Yahoo! Research

Abstract

In this article, we make the case for a declarative foundation for data-intensive machine learning systems. Instead of creating a new system for each specific flavor of machine learning task, or hard-coding new optimizations, we argue for the use of recursive queries to program a variety of machine learning algorithms. By taking this approach, database query optimization techniques can be utilized to identify effective execution plans, and the resulting runtime plans can be executed on a single unified data-parallel query processing engine.

1 Introduction

Supported by the proliferation of "Big Data" platforms such as Apache Hadoop, organizations are collecting and analyzing ever larger datasets. Increasingly, machine learning (ML) is at the core of data analysis for actionable business insights and optimizations. Today, machine learning is deployed widely: recommender systems drive the sales of most online shops; classifiers help keep spam out of our email accounts; computational advertising systems drive revenues; content recommenders provide targeted user experiences; machine-learned models suggest new friends, new jobs, and a variety of activities relevant to our profiles in social networks. Machine learning is also enabling scientists to interpret and draw new insights from massive datasets in many domains, including such fields as astronomy, high-energy physics, and computational biology.

The availability of powerful distributed data platforms and the widespread success of machine learning has led to a virtuous cycle wherein organizations are now investing in gathering a wider range of (even bigger!) datasets and addressing an even broader range of tasks. Unfortunately, the basic MapReduce framework commonly provided by first-generation "Big Data analytics" platforms like Hadoop lacks an essential feature for machine learning: MapReduce does not support iteration (or equivalently, recursion) or certain key features required to efficiently iterate "around" a MapReduce program. Programmers building ML models on such systems are forced to implement looping in ad-hoc ways outside the core MapReduce framework; this makes their programming task much harder, and it often also yields inefficient programs in the end. This lack of support has motivated the recent development of various specialized approaches or libraries to support iterative programming on large clusters. Examples include Pregel, Spark, and Mahout, each of which aims to support a particular family of tasks, e.g., graph analysis or certain types of ML models, efficiently. Meanwhile, recent MapReduce

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

extensions such as HaLoop, Twister, and PrItr aim at directly addressing the iteration outage in MapReduce; they do so at the physical level, however.

In our proposed approach to scalable machine learning, programmers need not learn to operate and use a plethora of distinct platforms. In the database world, relational database systems separate the conceptual, logical and physical schemas in order to achieve *logical* and *physical data independence*. Similarly, we aim to open the door in the machine learning world to principled optimizations by achieving a separation between:

- The user's program (in a domain-specific language that provides a library of available templates and user-definable functions) and the underlying *logical query*, expressed in Datalog. This shields the user from any changes in the logical framework, e.g., how the Datalog program is optimized.
- The logical Datalog query and an optimized *physical runtime plan*, reflecting details related to caching, storage, indexing, the logic for incremental evaluation and re-execution in the face of failures, etc. This ensures that any enhancements to the plan execution engine will automatically translate to more efficient execution, without requiring users to re-program their tasks to take advantage of the enhancements.

In essence, the separation identifies "modules" (such as the plan execution engine or the optimization of the logical Datalog program) where localized enhancements lead to higher overall efficiency. To illustrate our approach, we will describe a new high-level programming language called ScalOps, in which the data scientist encodes their machine learning task. We will then describe the ScalOps translation into Datalog for the example of a specific class of supervised machine learning algorithms. Lastly, we will demonstrate that an appropriately chosen data-intensive computing substrate, namely Hyracks [3], is able to handle the computational requirements of such programs through the application of dataflow processing techniques like those used in parallel databases [9].

2 High-level Abstractions for Machine Learning

A large class of machine learning algorithms are expressible in the statistical query model [11]. Statistical queries (e.g. max, min, sum, ...) themselves decompose into a data-local function and subsequent aggregation [6]. The MapReduce [8] programming model has been successful at performing this decomposition over a cluster of machines while abstracting away runtime concerns—like parallelism, fault-tolerance, and scheduling—from the programmer. However, for tasks in the machine learning domain, MapReduce is missing a key ingredient: iteration. Furthermore, the MapReduce programming model contains a group-by key component that is not needed for many machine learning tasks. Therefore, we have defined an alternative programming model called Iterative Map-Reduce-Update, which consists of the following three UDFs:

`map` receives read-only global state value (i.e., the model) as side information and is applied to all training data points in parallel.

`reduce` aggregates the `map`-output into a single aggregate value. This function is commutative and associative.

`update` receives the combined aggregate value and produces a new global state value for the next iteration or indicates that no additional iteration is necessary.

An Iterative Map-Reduce-Update runtime executes a series of iterations, each of which first calls `map` with the required arguments, then performs a global `reduce` aggregation, and lastly makes a call to `update`.

Example: Convex Optimization A large class of machine learning—including Support Vector Machines, Linear and Logistic Regression and structured prediction tasks such as machine translation—can be cast as convex optimization problems, which in turn can be solved efficiently using an Iterative Map-Reduce-Update approach [14]. The objective is to minimize the sum over all data points of the divergences (the loss) between the model's prediction and the known data. Usually, the loss function is convex and differentiable in the model,

Listing 1: Batch Gradient Descent in ScalOps

```
1 def bgd(xy: Table[Example], g:(Example, Vector)=>Vector, e:Double, l:Double) =
2   loop(zeros(1000), 0 until 100) {
3     w => (w - (xy.map(x => g(x, w)).reduce(_+_) * e) * (1.0 - e * l)
4   }
```

and therefore the *gradient* of the loss function can be used in iterative optimization algorithms such as Batch Gradient Descent. [1] Each model update step is a single Map-Reduce-Update iteration. The `map` UDF computes (loss, gradient) tuples for all data points, using the current model as side input. The `reduce` UDF sums those up and `update` computes a new model, which becomes the input to the next iteration of the optimization algorithm.

2.1 ScalOps: A Domain-Specific Language for Machine Learning

The Iterative Map-Reduce-Update programming model can be naturally represented in a high-level language. ScalOps is an internal domain-specific language (DSL) that we are developing for "Big Data" analytics [13]. ScalOps moves beyond single pass data analytics (i.e., MapReduce) to include multi-pass workloads, supporting iteration over algorithms expressed as relational queries on the training and model data. As a concrete example, consider the update rule for learning an L2-regularized linear regression model through batch gradient descent (BGD).[2] In each iteration of BGD, the gradient of the loss function with respect to the model w is computed for each data point and summed up:

$$w_{t+1} = \left(w_t - \eta \sum_{x,y} \delta_{w_t}(l(y, \langle w_t, x \rangle)) \right) (1 - \eta\lambda) \tag{1}$$

Here, λ and η are the regularization constant and learning rate, and δ_{w_t} denotes the derivative with respect to w_t. The sum in (1) decomposes per data point (x, y) and can therefore be trivially parallelized over the data set. Furthermore, the application of the gradient computation can be phrased as `map` in a MapReduce framework, while the sum itself can be phrased as the `reduce` step, possibly implemented via an aggregation tree for additional performance.

Listing 1 shows an implementation of this parallelized batch gradient descent in ScalOps. Line 1 defines a function `bgd` with the following parameters: the input data `xy`, the gradient function `g` (δ_{w_t} in (1)), the learning rate `e` (η) and the regularization constant `l` (λ). Line 2 defines the body of the function to be a `loop`, which is ScalOps' looping construct. It accepts three parameters: (a) Input state—here, we assume 1000 dimensions in the weight vector—(b) a "while" condition that (in this case [3]) specifies a range (`0 until 100`) and (c) a loop body. The loop body is a function from input state to output state, which in this case is from a Vector `w` to its new value after the iteration. In the loop body, the gradient is computed for each data point via a `map` call and then summed up in a `reduce` call. Note that `_+_` is a Scala shorthand for the function literal `(x,y) => x+y`.

ScalOps bridges the gap between imperative and declarative code by using data structures and language constructs familiar to the data scientist. This is evident in Listing 1, which is nearly a 1:1 translation of Equation 1. ScalOps query plans capture the semantics of the user's code in a form that can be reasoned over and optimized. For instance, since the query plan contains the loop information we can recognize the need to cache as much of `xy` in memory as possible, without explicit user hints. Furthermore, the expression inside of the loop body itself

[1] A more detailed discussion can be found in the Appendix of our technical report [5].

[2] Batch Gradient Descent, while not a state-of-the-art optimization algorithm, exemplifies the general structure of a larger class of convex optimization algorithms.

[3] We also support boolean expressions on the state object i.e., the weight vector.

is also captured. [4] This could offer further optimization opportunities reminiscent to traditional compilers i.e., dead-code elimination and constant-propagation.

3 A Declarative Approach to Machine Learning

The goal of machine learning (ML) is to turn observational data into a *model* that can be used to predict for or explain yet unseen data. While the range of machine learning techniques is broad, most can be understood in terms of three complementary perspectives:

- **ML as Search:** The process of training a model can be viewed as a *search problem*. A domain expert writes a program with an objective function that contains, possibly millions of, unknown parameters, which together form the *model*. A runtime program *searches* for a good set of parameters based on the objective function. A *search strategy* enumerates the parameter space to find a model that can correctly capture the known data and accurately predict unknown instances.
- **ML as Iterative Refinement:** Training a model can be viewed as iteratively closing the gap between the model and underlying reality being modeled, necessitating iterative/recursive programming.
- **ML as Graph Computation:** Often, the interdependence between the model parameters is expressible as a graph, where nodes are the parameters (e.g., statistical/random variables) and edges encode interdependence. This view gives rise to the notion of *graphical models*, and algorithms often reflect the graph structure closely (e.g., propagating refinements of parameter estimates iteratively along the edges, aggregating the inputs at each vertex).

Interestingly, each of these perspectives lends itself readily to expression as a Datalog program, and thereby to efficient execution by applying a rich array of optimization techniques from the database literature. The fact that Datalog is well-suited for iterative computations and for graph-centric programming is well-known [12], and it has also been demonstrated that Datalog is well-suited to search problems [7]. The natural fit between Datalog and ML programming has also been recognized by others [2, 10], but not at "Big Data" scale. It is our goal to make the optimizations and theory behind Datalog available to large-scale machine learning while facilitating the use of established programming models. To that end, we advocate *compilation* from higher order programming models to Datalog and subsequently physical execution plans.

3.1 Iterative Map-Reduce-Update in Datalog

We begin with the predicates and functions that form the building blocks of the Datalog program.

training_data(*Id,Datum****)*** is an extensional predicate corresponding to the training dataset.

model(*J,M****)*** is an intensional predicate recording the global model M at iteration J. Initialization is performed through the function predicate *init_model(M)* that returns an initial global model.

map**(M, R,** S**)** is a UDF predicate that receives as input the current model M and a data element R, and generates a statistic S as output.

reduce is an aggregation UDF function that aggregates several per-record statistics into one statistic.

update**(J, M, AggrS,** *NewM****)*** is a UDF predicate that receives as input the current iteration J, the current model M and the aggregated statistic $AggrS$, and generates a new model $NewM$.

[4] We currently capture basic math operators (+, -, *, /) over primitive types (int, float, double, vector, etc.).

Listing 2: Datalog runtime for the Iterative Map-Reduce-Update programming model. The temporal argument is defined by the *J* variable.

```
% Initialize the global model
G1: model(0, M) :- init_model(M).

% Compute and aggregate all outbound messages
G2: collect(J, reduce<S>) :- model(J, M),
  training_data(Id, R), map(R, M, S).

% Compute the new model
G3: model(J+1, NewM) :-
   collect(J, AggrS), model(J, M),
   update(J, M, AggrS, NewM), M != NewM.
```

The Datalog program in Listing 2 is a specification for the Iterative Map-Reduce-Update programming model. A special temporal variable (`J`) is used to track the iteration number. Rule *G1* performs initialization of the global model at iteration 0 using the function predicate `init_model`, which takes no arguments and returns the initial model in the (`M`) variable. Rules *G2* and *G3* implement the logic of a single iteration. Let us consider first rule *G2*. The evaluation of `model(J, M)` and `training_data(Id, R)` binds (`M`) and (`R`) to the current global model and a data record respectively. Subsequently, the evaluation of `map(M, R, S)` invokes the UDF that generates a data statistic (`S`) based on the input bindings. Finally, the statistics from all records are aggregated in the head predicate using the `reduce` UDF.

Rule *G3* updates the global data model using the aggregated statistics. The first two body predicates simply bind (`M`) to the current global model and (`AggrS`) to the aggregated statistics respectively. The subsequent function predicate `update(J, M, AggrS, NewM)` calls the `update` UDF; accepting (`J, M, AggrS`) as input and producing an updated global model in the (`NewM`) variable. The head predicate records the updated global model at time-step `J+1`.

Program termination is handled in rule *G3*. Specifically, `update` is assumed to return the same model when convergence is achieved. In that case, the predicate `M != NewM` in the body of *G3* becomes false and we can prove that the program terminates. Typically, `update` achieves this convergence property by placing a bound on the number of iterations and/or a threshold on the difference between the current and the new model.

3.2 Semantics and Correctness

The Datalog program for Iterative Map-Reduce-Update contains recursion that involves an aggregation, and hence we need to show that it has a well-defined output. Subsequently, we have to prove that this output corresponds to the output of the target programming model. In this section, we summarize a more formal analysis of these two properties from our technical report [5], which is based on the following theorem.

Theorem 1: The Datalog program in Listing 2 is XY-stratified [16].

The proof can be found in the Appendix of our technical report [5] and is based on the machinery developed in [16]. XY-stratified Datalog is a more general class than stratified Datalog. In a nutshell, it includes programs whose evaluation can be stratified based on data dependencies even though the rules are not stratified.

Following XY-stratification, we can prove that the output of the program in Listing 2 is computed from an initialization step that fires *G1*, followed by several iterations where each iteration fires *G2* and then *G3*. By translating the body of each rule to the corresponding extended relational algebra expression, and taking into account the data dependencies between rules, it is straightforward to arrive at the logical plan shown in Figure 1.

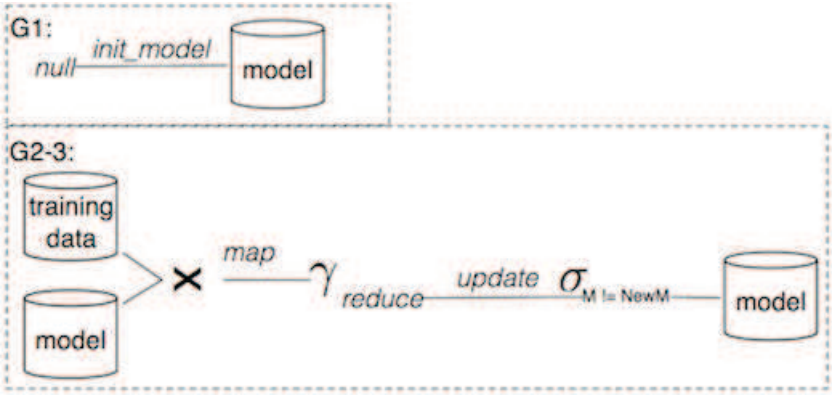

Figure 1: Logical query plan for Iterative Map-Reduce-Update.

The plan is divided into two separate dataflows, each labeled by the rules they implement in Listing 2. The dataflow labeled *G*1 initializes the global model using the `init_model` UDF, which takes no input, and produces the initial model. The *G*2–3 dataflow executes one iteration. The `cross-product` operator combines the model with each tuple in the training dataset, and then calls the `map` UDF on each result. (This part corresponds to the body of rule *G2*.) The mapped output is passed to a `group-all` operator, which uses the `reduce` aggregate (e.g., sum) to produce a scalar value. (This part corresponds to the head of rule *G2*.) The aggregate value, together with the model, is then passed to the `update` UDF, the result of which is checked for a new model. (This part corresponds to the body of *G3*.) A new model triggers a subsequent iteration, otherwise the `update` output is dropped and the computation terminates. (This part corresponds to the head of *G3*.)

4 A Runtime for Machine Learning

This section presents a high-level description of the physical plan that executes our Iterative Map-Reduce-Update programming model on a cluster of machines. A more detailed description can be found in our technical report [5]. Our physical plan consists of dataflow processing elements (operators) that execute in the Hyracks runtime [3]. Hyracks splits each operator into multiple tasks that execute in parallel on a distributed set of machines. Similar to parallel database systems and Hadoop, each task operates on a single partition of the input data. In Section 4.1, we describe the structure of the physical plan template and discuss its tunable parameters. Section 4.2 then explores the space of choices that can be made when executing this physical plan on an arbitrary cluster with given resources and input data.

4.1 Iterative Map-Reduce-Update Physical Plan

Figure 2 depicts the physical plan for the Iterative Map-Reduce-Update programming model. The top dataflow loads the input data from HDFS, parses it into an internal representation (e.g., binary formated features), and partitions it over N cluster machines, each of which caches as much of the data as possible, spilling to disk if necessary. The bottom dataflow executes the computation associated with the iteration step. The **Driver** of the iteration is responsible for seeding the initial global model and scheduling each iteration. The `map` step is parallelized across some number of nodes in the cluster determined by the optimizer. [5] Each `map` task sends a data statistic (e.g., a loss and gradient in a BGD computation) to a task participating in the leaf-level of an aggregation tree. This aggregation tree is balanced with a parameterized fan-in f (e.g., $f = 2$ yields a binary

[5] Reflected in the partitioning strategy chosen for the data loading step.

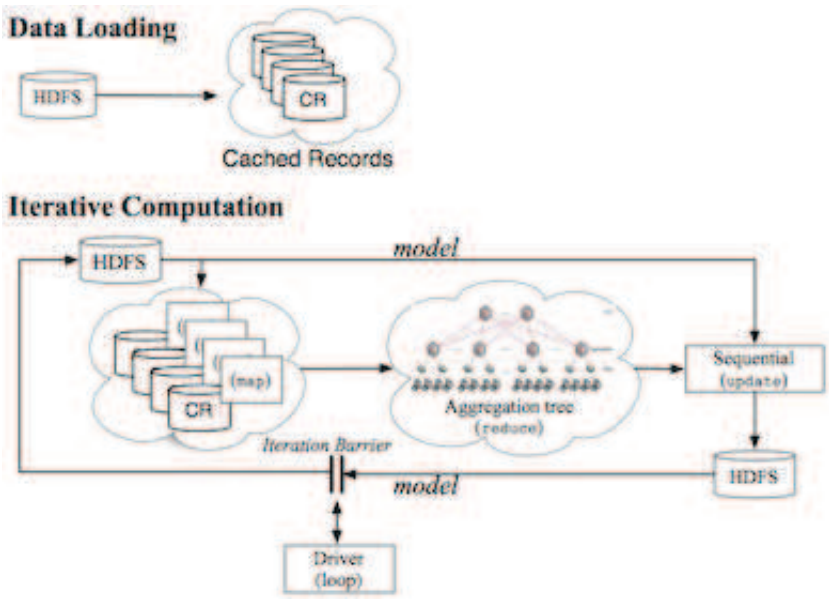

Figure 2: Hyracks physical plan for Iterative Map-Reduce-Update.

tree) determined by the optimizer. The final aggregate statistic is passed to the **Sequential** operator, which updates the global model and stores it in HDFS, along with some indicator that determines if another iteration is required based on evaluating the "loop" condition against the global model. The **Driver** detects this update and schedules another iteration on a positive indicator, terminating the computation otherwise.

This description of this physical plan highlights two choices to be determined by the optimizer—the number of nodes allocated for the map step, and the fan-in of the aggregation tree used in the reduce step.

4.2 The Plan Space

There are several considerations that must be taken into account when mapping the physical plan in Figure 2 to an actual cluster of machines. Many of these considerations are well-established techniques for executing data-parallel operators on a cluster of machines, and are largely independent of the resources available and data statistics. We begin by discussing these "universal" optimizations for arriving at an execution plan. Next, we highlight those choices that are dependent on the cluster configuration (i.e., amount of resources) and computation parameters (i.e., input data and aggregate value sizes).

4.2.1 Universal Optimizations

Data-local scheduling is generally considered an optimal choice for executing a dataflow of operators in a cluster environment: a map task is therefore scheduled on the machine that hosts its input data. **Loop-aware scheduling** ensures that the task state (i.e., the hash-table state associated with a join condition) is preserved across iterations. **Caching** of immutable data can offer significant speed-ups between iterations. However, careful consideration is required when the available resources do not allow for such caching. For example, Spark [15] assumes that sufficient main memory is always available to cache the data across iterations, but offers no assistance to the programmer for tuning this parameter, which may not even be feasible with the given resource allocation. **Efficient data representation** can offer significant performance improvements. For instance, we use a binary formated file, which has benefits in terms of space savings over simple Java objects.

4.2.2 Per-Program Optimizer Decisions

The optimizations discussed in Section 4.2.1 apply equally to all jobs and can be considered best practices inspired by the best-performing systems in the literature. This leaves us with two optimization decisions that depend on the cluster and computation parameters, which we highlight below. We have developed a theoretical foundation for an optimizer that can make these choices effectively, but elide the details due to space constraints.

Data partitioning determines the number of `map` tasks in an Iterative Map-Reduce-Update physical plan. For a given job and a maximum number N_{max} of machines available to it, the optimizer needs to decide which number $N \leq N_{max}$ of machines to request for the job. This decision is not trivial, even when ignoring the multi-job nature of today's clusters: more machines reduce the time in the map phase but increase the cost of the reduce phase, since more objects must be aggregated. The goal of data partitioning is to find the right trade-off between `map` and `reduce` costs: the `map` time *decreases* linearly with the number of machines and the `reduce` time *increases* logarithmically (see below). Using measurements of the actual times taken to process a single record in `map` and `reduce`, an optimal partitioning can be found in closed form.

Aggregation tree structure involves finding the optimal fan-in of a single `reduce` node in a balanced aggregation tree. Aggregation trees are commonly used to parallelize the `reduce` function. For example, Hadoop uses a `combiner` function to perform a single level aggregation tree, and Vowpal Wabbit [1] uses a binary aggregation tree. For the programming model we described above —with a blocking `reduce` operation— we have shown that the optimal fan-in for the aggregation tree is independent of the problem, both theoretically and empirically. In fact, its actual value is a per-cluster constant. The intuition behind this derivation lies in the difference between the optimal aggregation tree for a small versus a large number of leaf (`map`) nodes being sheer scaling, a process for which the fan-in of the tree does not change. Furthermore, there is an independence between the transfer time and aggregation time, in that time spent per aggregation tree level and the number of levels balance each other out.

5 Conclusion

The growing demand for machine learning is pushing both industry and academia to design new types of highly scalable iterative computing systems. Examples include Mahout, Pregel, Spark, Twister, HaLoop, and PrItr. However, today's specialized machine learning platforms all tend to mix logical representations and physical implementations. As a result, today's platforms 1) require their developers to rebuild critical components and to hardcode optimization strategies and 2) limit themselves to specific runtime implementations that usually only (naturally) fit a limited subset of the potential machine learning workloads. This leads to the current state of practice: implementing new scalable machine learning algorithms is very labor-intensive and the overall data processing pipeline involves multiple disparate tools hooked together with file- and workflow-based glue.

In contrast, we have advocated a declarative foundation on which specialized machine learning workflows can be easily constructed and readily tuned. We have verified our approach with Datalog implementations of two popular programming models from the machine learning domain: Iterative Map-Reduce-Update, for deriving linear models, and Pregel, for graphical algorithms (see [5]). The resulting Datalog programs are compact, tunable to a specific task (e.g., Batch Gradient Descent and PageRank), and translated to optimized physical plans. Our experimental results show that on a large real-world dataset and machine cluster, our optimized plans are very competitive with other systems that target the given class of ML tasks (see [5]). Furthermore, we demonstrated that our approach can offer a plan tailored to a given target task and data for a given machine resource allocation. In contrast, in our large experiments, Spark failed due to main-memory limitations and Hadoop succeeded but ran an order-of-magnitude less efficiently.

The work reported here is just a first step. We are currently developing the ScalOps query processing components required to automate the remaining translation steps; these include the Planner/Optimizer as well as a more general algebraic foundation based on extending the Algebricks query algebra and rewrite rule framework

of ASTERIX [4]. We also plan to investigate support for a wider range of machine learning tasks and for a more asynchronous, GraphLab-inspired programming model for encoding graphical algorithms.

References

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *CoRR*, abs/1110.4198, 2011.

[2] Ashima Atul. Compact implementation of distributed inference algorithms for network[s]. Master's thesis, EECS Department, University of California, Berkeley, Mar 2009.

[3] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[4] Vinayak R. Borkar, Michael J. Carey, and Chen Li. Inside "Big Data Management": Ogres, Onions, or Parfaits? In *EDBT*, 2012.

[5] Yingyi Bu, Vinayak Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling datalog for machine learning on big data. Technical report, CoRR. URL: http://arxiv.org/submit/427482 or http://isg.ics.uci.edu/techreport/TR2012-03.pdf, 2012.

[6] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[7] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1):1153–1165, 2008.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[10] Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Tim Furche, Georg Gottlob, Giovanni Grasso, Oege de Moor, and Andrew Sellers, editors, *Datalog 2.0*, Lecture Notes in Computer Science. Springer, 2011. 40 pages.

[11] Michael Kearns. Efficient noise-tolerant learning from statistical queries. In *Journal of the ACM*, pages 392–401. ACM Press, 1993.

[12] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[13] Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Machine learning in scalops, a higher order cloud computing language. In *BigLearn Workshop (NIPS 2011)*, Sierra Nevada, Spain, December 2011.

[14] Markus Weimer, Sriram Rao, and Martin Zinkevich. A convenient framework for efficient parallel multipass algorithms. In *LCCC : NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds*, December 2010.

[15] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. HotCloud'10, page 10, Berkeley, CA, USA, 2010.

[16] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *DOOD*, pages 204–221, 1993.

Building LinkedIn's Real-time Activity Data Pipeline

Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede,
Richard Park, Jun Rao, Victor Yang Ye
LinkedIn

Abstract

One trend in the implementation of modern web systems is the use of activity data in the form of log or event messages that capture user and server activity. This data is at the heart of many internet systems in the domains of advertising, relevance, search, recommendation systems, and security, as well as continuing to fulfill its traditional role in analytics and reporting. Many of these uses place real-time demands on data feeds. Activity data is extremely high volume and real-time pipelines present new design challenges. This paper discusses the design and engineering problems we encountered in moving LinkedIn's data pipeline from a batch-oriented file aggregation mechanism to a real-time publish-subscribe system called Kafka. This pipeline currently runs in production at LinkedIn and handles more than 10 billion message writes each day with a sustained peak of over 172,000 messages per second. Kafka supports dozens of subscribing systems and delivers more than 55 billion messages to these consumer processing each day. We discuss the origins of this systems, missteps on the path to real-time, and the design and engineering problems we encountered along the way.

1 Introduction

Activity data is one of the newer ingredients in internet systems. At LinkedIn this kind of data feeds into virtually all parts of our product. We show our users information about who has viewed their profile and searching for them; we train machine learning models against activity data to predict connections, match jobs, and optimize ad display; we show similar profiles, jobs, and companies based on click co-occurance to help aid content discovery; we also populate an activity-driven newsfeed of relevant occurrences in a user's social network. Likewise, this data helps keep the web site running smoothly: streams of activity data form the basis of many of our security measures to prevent fraud and abuse as well as being the data source for various real-time site monitoring tools. These uses are in addition to the more traditional role of this kind of data in our data warehouse and Hadoop environments for batch processing jobs, reporting, and ad hoc analysis. There are two critical differences in the use of this data at web companies in comparison to traditional enterprises. The first is that activity data is often needed in real-time for monitoring, security, and user-facing product features. The second is that these data feeds are not simply used for reporting, they are a dependency for many parts of the website itself and as such require more robust infrastructure than might be needed to support simple log analysis. We refer to this real-time feed of messages as our activity data pipeline. We describe the motivations and design of a data pipeline built around Apache Kafka [11], a piece of infrastructure we designed for this usage and subsequently released as an open source project with the Apache Software Foundation.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 Previous Systems

We will describe some details of our previous generation of infrastructure for these domains to help motivate our design. We believe they are fairly representative of industry practices. We began, several years ago, with two separate systems: a batch-oriented system that handled *user* activity data, designed purely to serve the needs of loading data into a data warehouse, and a second system that handled *server* metrics and logging but provided this feed only to our monitoring system. All of these systems were effectively point-to-point data pipelines that delivered data to a single destination with no integration between.

Our user activity tracking system consisted of a simple HTTP logging service to which applications directly published small batches of xml messages; these messages were written to aggregate files and then copied to ETL servers where they were parsed, and loaded into our relational data warehouse and Hadoop [6] clusters. The problems with this service included the lack of real-time data access, and the limitation that only a single destination for data was supported (though many other uses for the data had since arisen). In addition, because there was a large diversity of activity data types captured, and the xml schemas were determined by the application emitting the data, schemas would often change unexpectedly, breaking downstream processing. Reasoning about or testing for compatibility was difficult because of the very asynchronous nature of this data consumption, making it hard to assess what code might break due to any given change. The use of XML was itself problematic and dated from the early days of the company. Custom parsing work had to be done for each xml type to make the data available and this parsing proved very computationally expensive and difficult to maintain. The fragility and complexity of this pipeline lead to inevitable delays in adding new types of activity data, which resulted in shoe-horning new activities into inappropriate existing types to avoid manual work, or worse, not capturing activities at all.

Monitoring data was not handled by this system due to its hourly batch-oriented nature, and was instead scraped using JMX hooks in applications and other system tools and made available in Zenoss, an open source monitoring tool. At LinkedIn, monitoring data consists of application counters that track various system statistics as well as structured logs. This includes fairly sophisticated request tracing capabilities in the service layer modeled after Google's Dapper system [14] to allow correlating requests across a graph of service calls. Unfortunately adding and maintaining metrics in this system was a cumbersome manual process and the data was not available elsewhere.

We had several goals in replacing these systems. The first was to be able to share data easily between these different types of feeds. For a large-scale consumer website, operational system metrics about server and service performance *are* business metrics. They are of concern to everyone from the individual engineers and operations staff to our executive team. Artificially separating these from more traditional business metrics lead to difficulties correlating problems that appeared at the system level with impacts in business metrics. Many types of system problems and bugs go undetected in system metrics but have negative business impact (e.g. a decrease in page views, sign-up rate, or other activity). Since our business metrics were not available in real-time we were dependent on a data warehouse ETL pipeline that ran with several hours of delay to detect many types of problems. This lag lead to long detection times for certain problems as well as ever-increasing pressure on the latency of the data warehouse ETL processes. Likewise, because our operational metrics were only captured in a real-time monitoring system they were not easily accessible for deeper longitudinal queries for capacity analysis or system debugging. Both systems suffered due to the sheer number of feeds that were maintained by hand, and both had similar backlogs of missing data that needed to be added by the central teams responsible for the systems. One goal was to ensure that any system—Hadoop, monitoring tools, etc—could integrate with the data pipeline and produce or consume any of these data sources with minimal additional integration work.

We had a broader motivation based in our conception of the role of the data warehouse for an internet company such as LinkedIn. The traditional role of the data warehouse is to curate and maintain a cleansed, structured copy of all data. Access to clean, complete data is of the utmost importance to a data-centric company, however having the data warehouse be the sole location of the cleansed and structured data is problematic. Our

data warehouse infrastructure consists of both a traditional relational system as well as a set of large Hadoop clusters. These are inherently batch-oriented systems that process data on an hourly or daily cycle. Making this the only location where clean data is uniformly available is problematic because it limits access to clean data to batch-oriented infrastructure. Modern websites are in large part real-time or near-real-time systems and have significant need to populate data in systems with low latency. We wanted to move as much of the structure and cleanliness "up-stream" into the real-time feed and allow batch systems to inherit this structure as one among many consumers. This helps to scale the organizational problem of integrating all data by moving many problems off the central team that owns Hadoop or the data warehouse infrastructure and bringing it to the owner of the relevant data sources directly.

1.2 Struggles with real-time infrastructure

We made several exploratory attempts to support real-time consumption with the existing pipeline infrastructure. We began by experimenting with off-the-shelf messaging products. Our tests with these systems showed poor performance when any significant backlog of persistent data accumulated, making them unsuitable for slower batch consumption. However, we envisioned them running as a real time feed in front of our central logging service as a method of providing real-time access to that data stream, with the existing batch file consumption remaining untouched. We selected ActiveMQ as a potential system for the message queue. It did well in our real-time performance tests, being capable of processing several thousand messages per second, and was already being used successfully in other LinkedIn systems for more traditional queuing workloads. We released a quick prototype built on ActiveMQ to test the system under full production load without any products or systems depending on it other than a test consumer and the tracking system that produced log file output. In our production tests we ran into several significant problems, some due to our approach to distributing load, and some due to problems in ActiveMQ itself. We understood that if the queue backed up beyond what could be kept in memory, performance would severely degrade due to heavy amounts of random I/O. We hoped we could keep the pipeline caught up to avoid this, but we found that this was not easy. Consumer processes balanced themselves at random over ActiveMQ instances so there were short periods where a given queue had no consumer processes due to this randomization. We did not want to maintain static assignment of consumer to brokers as this would be unmanageable, and downtime in a consumer would then cause backup. We also encountered a number of bugs that came out under production load that caused brokers to hang, leak connections, or run out of memory. Worse, the back-pressure mechanism would cause a hung broker to effectively block the client, which in some cases could block the web server from serving the pages for which the data was sent. The "virtual topic" feature we used to support clustered consumers required duplicating the data for each consumer in a separate queue. This leads to some inefficiencies since all topics would have at least two copies of the data, one for the real-time consumer and one for the batch log aggregation service. Further we had difficulties with ActiveMQ's built in persistence mechanism that lead to very long restart times. We believe some of these issues have since been addressed in subsequent ActiveMQ releases, but the fundamental limitations remain. We have included more detailed comparative benchmarks in a prior publication [11].

These problems motivated us to change direction. Messaging systems seem to target low-latency settings rather than the high-volume scale-out deployment that we required. It would be possible to provide enough buffer to keep the ActiveMQ brokers above water but according to our analysis this would have required several hundred servers to process a subset of activity data even if we omitted all operational data. In addition this did not help improve the batch data pipeline, adding to the overall complexity of the systems we would need to run.

As a result of this we decided to replace both systems with a piece of custom infrastructure meant to provide efficient persistence, handle long consumer backlogs and batch consumers, support multiple consumers with low overhead, and explicitly support distributed consumption while retaining the clean real-time messaging abstraction of ActiveMQ and other messaging systems. We put in place a simple in-memory relay as a stop-gap solution to support real-time product use-cases and began the development of the first version of Kafka.

2 Pipeline Infrastructure: An Overview of Apache Kafka

In this section we give a brief overview of Apache Kafka. Kafka acts as a kind of write-ahead log that records messages to a persistent store and allows subscribers to read and apply these changes to their own stores in a system appropriate time-frame. Common subscribers include live services that do message aggregation or other processing of these streams, as well as Hadoop and data warehousing pipelines which load virtually all feeds for batch-oriented processing.

2.1 Conceptual Overview of Kafka

Kafka models messages as opaque arrays of bytes. Producers send messages to a Kafka *topic* that holds a feed of all messages of that type. For example, an event that captures the occurrence of a search on the website might record details about the query, the results and ranking, and the time of the search. This message would be sent to the "searches" topic. Each topic is spread over a cluster of Kafka brokers, with each broker hosting zero or more partitions for each topic. In our usage we always have a uniform number of partitions per broker and all topics on each machine. Each partition is an ordered write-ahead log of messages. There are only two operations the system performs: one to append to the end of the log, and one to fetch messages from a given partition beginning from a particular message id. We provide access to these operations with APIs in several different programming languages. Our system is designed to handle high throughput (billions of messages) on a moderate number of topics (less than a thousand). All messages are persistent, and several tricks are used to make sure this can be done with low overhead. All topics are available for reading by any number of subscribers, and additional subscribers have very low overhead.

Kafka is a publish/subscribe system, but our notion of a subscriber is generalized to be a group of co-operating processes running as a cluster. We deliver each message in the topic to one machine in each of these subscriber groups. Consumer clusters vary in size: real-time services that consume data feeds are often made up of a few dozen servers, whereas our Hadoop clusters typically range between 100 and 500 nodes.

Both broker and consumer group membership is fully dynamic, which allows us to dynamically add nodes to the groups and have load automatically rebalanced without any static cluster configuration. We use Zookeeper [9] to manage this group membership and assign partition consumers. Each consumer process announces itself as part of a named group (e.g. "hadoop-etl") and the appearance or disappearance of such a process triggers a re-assignment of partitions to rebalance load across the new set of consumer processes. From the time of this assignment to the next reassignment, each process is the only member of the group allowed to consume from its set of partitions. As a result the partition is the unit of parallelism of the topic and controls the maximum parallelism of the subscriber. This arrangement allows the metadata concerning what has been consumed from a particular partition by a given consumer group to be a simple eight-byte monotonically increasing message id recording the last message in that partition that has been acknowledged.

Because it is the sole consumer of the partition within that group, the consuming process can lazily record its own position, rather than marking each message immediately. If the process crashes before the position is recorded it will just reprocess a small number of messages, giving "at-least-once" delivery semantics. This ability to lazily record which messages have been consumed is essential for performance. Systems which cannot elect owners for partitions in this fashion must either statically assign consumers with only a single consumer per broker or else randomly balance consumers. Random assignment means multiple consumers consume the same partition at the same time, and this requires a mechanism to provide mutual exclusion to ensure each messages goes to only one consumer. Most messaging systems implement this by maintaining per-message state to record message acknowledgement. This incurs a great deal of random I/O in the systems that we benchmarked to update the per-message state. Furthermore this model generally eliminates any ordering as messages are distributed at random over the consumers which process in a non-deterministic order.

Message producers balance load over brokers and sub-partitions either at random or using some application-

supplied key to hash messages over broker partitions. This key-based partitioning has two uses. First the delivery of data within a Kafka partition is ordered but no guarantee of order is given between partitions. Consequently, to avoid requiring a global order over messages, feeds that have a requirement for ordering need to be partitioned by some key within which the ordering will be maintained. The second use is to allow in-process aggregation for distributed consumers. For example, consumers can partition a data stream by user id, and perform simple in-memory session analysis distributed across multiple processes relying on the assumption that all activity for a particular user will be sticky to one of those consumer process. Without this guarantee distributed message processors would be forced to materialize all aggregate state into a shared storage system, likely incurring an expensive look-up round-trip per message.

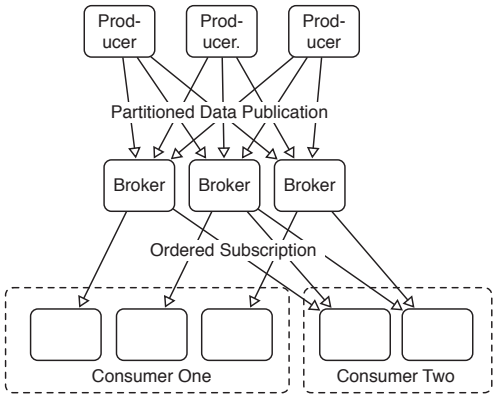

Figure 1: The physical layout of a Kafka cluster. This figure depicts several producer processes sending data to a single multi-broker Kafka cluster with two clustered subscribers.

In our current deployment Kafka doesn't attempt to provide guarantees to prevent all possibility of data loss. A small number of messages can be lost in the event of a hard server failure before messages are transfered to the Kafka brokers or flushed to disk. Instead of attempting to provide hard guarantees, we measure data at each logical tier (producer, broker, and consumers) to measure loss or delay. We discuss this approach and the results we achieve for our production system in a subsequent section.

2.2 Kafka Usage at LinkedIn

Kafka handles over 10 billion message writes per day with a sustained load over the highest traffic hour that averages 172,000 messages per second. We see very heavy use of our multi-subscriber support, both for our own inter-data-center replication and various external applications that make use of the data. There are a total of 40 real-time data consumers that consume one or more topics, eight of which are our own replication and pipeline monitoring tools, and 32 of which are applications written by other users for other product features or tools. In total, we see a ratio of roughly 5.5 messages consumed for each message produced in our live data centers. This results in a daily total in excess of 55 billion messages delivered to real-time consumers as well as a complete copy of all data delivered to Hadoop clusters with a short delay fashion.

We support 367 topics that cover both user activity topics (such as page views, searches, and other activities) and operational data (such as log, trace, and metrics for most of LinkedIn's services). The largest of these topics adds an average of 92GB per day of batch-compressed messages, and the smallest adds only a few hundred kilobytes. In addition to these outbound topics that produce feeds from live applications, we have "derived" data feeds that come out of asynchronous processing services or out of our Hadoop clusters to deliver data back to the live serving systems.

Message data is maintained for 7 days. Log segments are garbage collected after this time period whether or not they have been consumed. Consumers can override this policy on a per-topic basis but few have seen a need to do so. In total we maintain 9.5 TB of compressed messages across all topics.

Due to careful attention to efficiency, the footprint for the Kafka pipeline is only 8 servers per data center, making it significantly more efficient than the activity logging system it replaced. Each Kafka server maintains 6TB of storage on a RAID 10 array of commodity SATA drives. Each of these servers has a fairly high connection fan-in and fan-out, with over 10,000 connections into the cluster for data production or consumption. In addition to these live consumers we have numerous large Hadoop clusters which consume infrequent, high-throughput, parallel bursts as part of the offline data load.

We handle multiple data centers as follows: producers always send data to a cluster in their local data center, and these clusters replicate to a central cluster that provides a unified view of all messages. This replication works through the normal consumer API, with the central destination cluster acting as a consumer on the source clusters in the online data centers. We see an average end-to-end latency through this pipeline of 10 seconds and a worst case of 32 seconds from producer event generation to a live consumer in the offline data center. This delay comes largely from throughput-enhancing batching optimizations that will be described in the next section. Disabling these in performance experiments gives end-to-end round trips of only a few milliseconds but incurs lots of small random I/O operations.

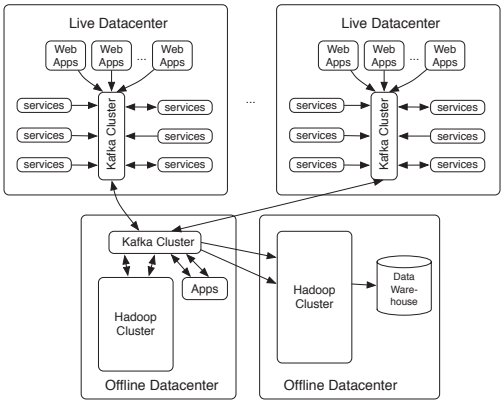

Figure 2: A simplified view of how the data pipeline fits into LinkedIn's data center topology. Our configuration always sends live traffic to a local Kafka cluster and these live clusters are replicated to an aggregate cluster which contains a full view of all data for processing or loading into Hadoop. Data is also sent back from these applications and from Hadoop processing as new Kafka streams. The "live data centers" are where the real-time application serving happens and the "offline data centers" house Hadoop and other offline analytical infrastructure.

3 Engineering for High Throughput

The primary design criteria for a pipeline such as this is maximizing the end-to-end throughput while providing low-latency and reasonable semantics to the system users. It is also important that throughput and latency remain constant with respect to the volume of unconsumed data as we often have slow or batch-oriented consumers. We will describe some of the optimizations that are essential for achieving high and consistent throughput.

High-throughput is one advantage of the design of traditional log aggregation systems over most messaging systems. Data is written to a set of text logs with no immediate flush to disk, allowing very efficient I/O patterns. Unfortunately there are a number of drawbacks. Simple log files provide very poor semantics with respect

to message loss and make low-latency incremental consumption challenging. Partial messages can occur in the event of a crash and can not automatically be corrected if the system doesn't understand the internal log format, resulting in corrupt data in the feed. The unit of consumption is usually the file as there is no logical identifier at the message level. This introduces the log file (which should be an implementation detail) as a first-class concept; clients need to remember which files have been processed and restart correctly in the event of a failure. Likewise, the file must be complete before it can be aggregated for processing, which introduces artificial latency. This latency can be mitigated by rolling over files more often (for example, some companies use "minute files") but this comes at a cost of exploding the number of files maintained. One of our design goals was to maintain the simple I/O model of logging systems while providing the cleaner, more granular model of messaging systems.

Kafka uses three primary techniques to improve the effective throughput. The first is partitioning up data so that production, brokering, and consumption of data are all handled by clusters of machines that can be scaled incrementally as load increases. The second is to batch messages together to send larger chunks at once. The third is shrinking the data to make the total data sent smaller. These techniques are well known, but not always used in messaging systems we tested. We will show how these techniques can be implemented end-to-end throughout the system.

3.1 Batching

Kafka bridges the gap between the online request-driven world, which supplies small messages one at a time as events occur, and the near-real-time asynchronous systems or the batch processing systems that feed off this data. Directly processing these small messages as they occur in a request-driven fashion leads to small network and disk operations which in turn severely limit throughput. This trade-off between latency and throughput is well known [12] and exists in some form in many disk and network I/O systems. The fundamental technique we use is to incur a small amount of latency to group small messages together and improve throughput. We implement this in a way that is tunable at the application level. We performed a series of simple experiments on a configuration that matches our production hardware to demonstrate the effect of individual batching optimizations over unbatched operation.

The Kafka producer client can be configured to send messages in either a synchronous or asynchronous fashion. The async mode allows the client to batch small random messages into larger data chunks before sending it over the network. TCP implementations generally support a similar trick for batching using Nagle's algorithm, but we do this at the application level. This allows us to implement a precise timeout and message count threshold that triggers the send, guaranteeing that we never batch more than a given number of messages and that no message is held for more than a configurable number of milliseconds. This precise control is important since messages sent asynchronously are at risk of loss if the application crashes. Originally we performed this batching only on a per-topic basis, however performance analysis showed it to be very effective and there are many low-volume topics, so we reworked our APIs to support a "multisend" operation. This allows us to group messages for low-volume topics into batches with the high-volume topics and further reduce the number of small requests. In our tests allowing the producer to batch messages into groups of 200 improves throughput by a factor of 3.2.

The broker does not do further batching of filesystem writes as we found the cost of a buffered write to be extremely low, especially after the client side batching has taken place. Instead we simply rely on the filesystem pagecache to buffer writes and delay the flush to disk to batch the expensive physical disk writes. This ability to delay physical write activity allows the operating system I/O scheduler to coalesce small random appends to hundreds of topics into a larger linear write pattern. Putting all data immediately into the filesystem has the benefit that no data is lost during application crashes that leave the operating system intact. This flush batching is also governed by a configurable policy that is triggered by an elapsed time or the number of accumulated messages in that partition. A similar delayed flush policy is implemented in Linux's "pdflush" daemon [10]; but

again doing this in the application allows us to explicitly respect message boundaries and have different policy configurations for different topics including an option for synchronous flushing of writes for important topics. Without this technique, our strategy of always persisting messages would be extremely inefficient: we see an improvement of 51.7x over immediately flushing each write due to the large number of topics and partitions and resulting random I/O incurred.

Consumer data fetch requests are also effectively batched. Data consumption occurs with the client requesting messages from the broker by specifying a given starting message id and some user-specified maximum buffer size (say one megabyte) to fill with message data. This batching improves performance in our tests by a factor of 20.1. As with the send API, this fetch API is a "multifetch" allowing the user to pull data from many topics and partitions in a single request to avoid small requests on low-volume topics as messages trickle in.

In combination these batching techniques allow us to give end-to-end throughput that matches the I/O capacity of the underlying disk sub-system, 50-100 MB/sec for an array of commodity disks. There is little or no negative impact from small messages or large backlogs of unconsumed data. We give a more detailed comparison to other messaging systems in previous work [11].

Batching Type	Improvement	Default Threshold
Producer	3.2x	200 messages or 30 seconds
Broker	51.7x	50000 messages or 30 seconds
Consumer	20.1x	1 megabyte

Table 5: Improvements due to batching

3.2 Shrinking Data

Shrinking data is the most important performance technique as one of our most fundamental bottlenecks is the bandwidth between data center facilities. This bandwidth is more expensive per-byte to scale than disk I/O, CPU, or network bandwidth capacity within a facility. There are two basic approaches to shrinking data: exploiting explicit structure in message serialization and exploiting implicit structure using compression.

We extract repeated known structure from the messages by associating schemas with the topics that capture the repeated field names and type information used for efficient storage. Schemas have other benefits in ensuring compatibility and allowing integration which we will discuss in a later section. Though Kafka itself is serialization-agnostic, LinkedIn has standardized on the use of Apache Avro [4] as the schema and serialization language for all of its activity data messages as well as data in Hadoop and most other data systems. In uncompressed form our Avro data was roughly 7 times smaller than XML messages in the activity logging system we replaced.

One advantage of traditional file-logging is that it naturally allows compressing an entire log file as a single stream, whereas no messaging system we are aware of provides this kind of batch compression of messages. Most messages are small and once they are encoded in a compact serialization format have little redundancy on their own. Kafka allows the batch compression of multiple messages into a single composite message set to allow effective compression. We find that this batch compression of a groups of a few hundred messages is comparable to full file compression in compression ratio and allows us to use compression end-to-end from the message producer to the destination consumer. Compression is done as part of the batching in the producer API and compressed message sets containing a few hundred messages are sent to Kafka where they are retained and served in compressed form. Kafka supports a variety of compression algorithms, but at LinkedIn we use standard GZIP as it provides tight compression. We see a compression ratio of roughly 3x on our Avro data sets. Despite the CPU intensive nature of GZIP compression we actually see a 30% improvement in throughput in our tests, presumably due to the reduced network and filesystem I/O.

3.3 Reliance on Pagecache

Kafka does not attempt to make any use of in-process caching and instead relies on persistent data structures and OS page cache. We have found this to be an effective strategy. In Linux (which we use for our production environment) page cache effectively uses all free memory, which means the Kafka servers have roughly 20GB of cache available without any danger of swapping or garbage collection problems. Using this cache directly instead of an in-process cache avoids double-buffering data in both the OS and in process cache. The operating system's read-ahead strategy is very effective for optimizing the linear read pattern of our consumers which sequentially consume chunks of log files. The buffering of writes naturally populates this cache when a message is added to the log, and this in combination with the fact that most consumers are no more than 10 minutes behind, means that we see a 99% pagecache hit ratio on our production servers for read activity, making reads nearly free in terms of disk I/O.

The log file itself is written in a persistent, checksummed, and versioned format that can be directly included when responding to client fetch requests. As a result we are able to take advantage of Linux's sendfile system call [10]. This allows the transmission of file chunks directly from pagecache with no buffering or copying back and forth between user and kernel space to perform the network transfer.

These three features—the high cache hit ratio of reads, the sendfile optimization, and the low bookkeeping overhead of the consumers—make adding consumers exceptionally low impact.

4 Handling Diverse Data

Some of the most difficult problems we have encountered have to do with the variety of data, use cases, users, and environments that a centralized data pipeline must support. These problems tend to be the least glamorous and hardest to design for ahead of time because they can't be predicted by extrapolating forward current usage. They are also among the most important as efficiency problems can often be resolved by simply adding machines, while "integration" problems may fundamentally limit the usefulness of the system. Our pipeline is used in a direct way by several hundred engineers in all LinkedIn product areas and we handle topics that cover both business metrics, application monitoring and "logging", as well as derived feeds of data that come out of the Hadoop environment or from other asynchronous processing applications.

One of the biggest problems we faced in the previous system was managing hundreds of evolving XML formats for data feeds without breaking compatibility with consumers depending on the data feeds. The root cause of the problem was that the format of the data generated by applications was controlled by the application generating the data. However testing or even being aware of all processing by downstream applications for a given data feed was a difficult undertaking. Since XML did not map well to the data model of either our Hadoop processing or live consumers, this meant custom parsing and mapping logic for each data consumer. As a result loading new data in systems was time consuming and error prone. These load jobs often failed and were difficult to test with upcoming changes prior to production release.

This problem was fixed by moving to Avro and maintaining a uniform schema tied to each topic that can be carried with the data throughout the pipeline and into the offline world. Those responsible for the application, those responsible for various data processing tasks, and those responsible for data analysis, could all discuss the exact form of the data prior to release. This review process was mandatory and built into the automatic code review process schema addition and changes. This also helped to ensure that the meaning of the thousands of fields in these events were precisely documented in documentation fields included with the schema. We have a service which maintains a history of all schema versions ever associated with a given topic and each message carries an id that refers to exact schema version with which it was written. This means it is effectively impossible for schema changes to break the deserialization of messages, as messages are always read with precisely the schema they were written.

This system was an improvement over XML, however during the process of rolling out the system we found that even these improvements were not sufficient in fully transitioning ownership of data format off a centralized team or in preventing unexpected incompatibilities with data consumers. Although the messages were now individually understandable, it was possible to make backwards incompatible changes that broke consumer code that used the data (say, by removing needed fields or changing its type in an incompatible way). Many of these incompatible changes can easily be made in the application code that generates the data but have implications for the existing retained data. Historical data sets are often quite large (hundreds of TBs), and updating previous data to match a new schema across a number of Hadoop clusters can be a major undertaking. Hadoop tools expect a single schema that describes a data set—for example Hive [7], Pig [8], and other data query tools allow flexible data format but they assume a static set of typed columns so they do not cope well with an evolving data model. To address these issues we developed an exact compatibility model to allow us to programmatically check schema changes for backwards compatibility against existing production schemas. This model only allows changes that maintain compatibility with historical data. For example, new fields can be added, but must have a default value to be used for older data that does not contain the field. This check is done when the schema is registered with the schema registry service and results in a fatal error if it fails. We also do a proactive check at compile time and in various integration environments using the current production schema to detect incompatibilities as early as possible.

5 Hadoop Data Loads

Making data available in Hadoop is of particular importance to us. LinkedIn uses Hadoop for both batch data processing for user-facing features as well as for more traditional data analysis and reporting. Previous experience had lead us to believe that any custom data load work that had to be done on a per-topic basis would become a bottleneck in making data available, even if it were only simple configuration changes. Our strategy to address this was to push the schema management and definition out of Hadoop and into the upstream pipeline system and make the data load into Hadoop a fully automated projection of the contents of this pipeline with no customization on a per-event basis. This means that there is a single process that loads all data feeds into Hadoop, handles data partitioning, and creates and maintains Hive tables to match the most recent schema. The data pipeline and the Hadoop tooling were designed together to ensure this was possible. This ability is heavily dependent on Hadoop's flexible data model to allow us to carry through a data model from the pipeline to Hadoop and integrate with a rich collection of query tools without incurring non-trivial mapping work for each usage.

To support this we have built custom Avro plugins for Pig and Hive to allow them to work directly with Avro data and complement the built-in Avro support for Java MapReduce. We have contributed these back to the open source community. This allows all these common tools to share a single authoritative source of data and allow the output of a processing step built with one technology to seamlessly integrate with other tools.

The load job requires no customization or configuration on a per-topic basis whatsoever. When a new topic is created we automatically detect this, load the data, and use the Avro schema for that data to create an appropriate Hive table. Schema changes are detected and handled in a similar fully-automatic manner. This kind of automated processing is not uncommon in Hadoop, but it is usually accomplished by maintaining data in an unstructured text form that pushes all parsing and structure to query time. This simplifies the load process, but maintaining parsing that is spread across countless MapReduce jobs, pig scripts, and queries as data models evolve is a severe maintenance problem (and likely a performance problem as well). Our approach is different in that our data is fully structured and typed, but we inherit this structure from the upstream system automatically. This means that users never need to declare any data schemas or do any parsing to use our common tools on data in Hadoop: the mapping between Avro's data model and Pig and Hive's data model is handled transparently.

We run Hadoop data loads as a MapReduce job with no reduce phase. We have created a custom Kafka

InputFormat to allow a Kafka cluster to act as a stand in for HDFS, providing the input data for the job. This job is run on a ten minute interval and takes on average two minutes to complete the data load for that interval. At startup time the job reads its current offset for each partition from a file in HDFS and queries Kafka to discover any new topics and read the current log offset for each partition. It then loads all data from the last load offset to the current Kafka offset and writes it out to Hadoop, project all messages forward to the current latest schema.

Hadoop has various limitations that must be taken into consideration in the design of the job. Early versions of our load process used a single task per Kafka topic or partition. However we found that in practice we have a large number of low-volume topics and this lead to bottlenecks in the number of jobs or tasks many of which have only small amounts of processing to do. Instead, we found that we needed to intelligently balance Kafka topic partitions over a fixed pool of tasks to ensure an even distribution of work with limited start-up and tear-down costs.

This approach to data load has proven very effective. A single engineer was able to implement and maintain the process that does data loads for all topics; no incremental work or co-ordination is needed as teams add new topics or change schemas.

6 Operational Considerations

One of the most important considerations in bringing a system to production is the need to fully monitor its correctness. This kind of monitoring is at least as important as traditional QA procedures and goes well beyond tracking simple system-level stats. In practice, we have found that there are virtually no systems which cannot tolerate some data loss, transient error, or other misfortune, provided it is sufficiently rare, has bounded impact, and is fully measured.

One deficiency of previous generations of systems at LinkedIn was that there was no such measurement in place, and indeed it is inherently a hard thing to do. This meant that although certain errors were seen in the old system, there was no way to assert that the final number of messages received was close to the right number, and no way to know if the errors seen were the only errors or just the ones that were visible.

To address this we added deep monitoring capabilities, in addition to the normal application metrics we would capture for any system. We built a complete end-to-end audit trail that attempts to detect any data loss. This is implemented as a special Kafka topic that holds audit messages recording the number of messages for a given topic in a given time period. Each logical tier—a producer or consumer in some data center—send a periodic report of the number of messages it has processed for each topic in a given ten minute time window. The sum of the counts for each topic should be the same for all tiers for a ten-minute window if no data is lost. The count is always with respect to the time stamp in the message not the time on the machine at the time of processing to ensure the consumption of delayed messages count towards the completeness of the time bucket in which they originated not the current time bucket. This allows exact reconciliation even in the face of delays. We also use this system to measure the time it takes to achieve completeness in each tier as a percentage of the data the producer sent. We phrase our SLA as the time to reach 99.9% completeness at each tier below the producer. This audit monitoring measures each step the data takes throughout the pipeline—across data centers, to each consumer, and into Hadoop—to ensure data is properly delivered in a complete and timely fashion everywhere. We have a standalone monitoring application that consumes the audit data topic and performs an exact reconciliation between all tiers to compute loss or duplication rates as well as to perform graphing and alerting on top of this data.

This detailed audit helped discover numerous errors including bugs, misconfigurations, and dozens of corner cases that would otherwise have gone undetected. This general methodology of building a self-proving verification system into the system (especially one that relies on the fewest possible assumptions about the correctness of the system code) is something we found invaluable and intend to replicate as part of future systems.

We are able to run this pipeline with end-to-end correctness of 99.9999%. The overwhelming majority of

data loss comes from the producer batching where hard kills or crashes of the application process can lead to dropping unsent messages.

7 Related Work

Logging systems are a major concern for web sites, and similar systems exist at other large web companies. Facebook has released Scribe as open source which provides similar structured logging capabilities [3] as well as described some of the other real-time infrastructure in their pipeline [2]. Other open source logging solutions such as Flume also exist [5]. Yahoo has some description of their data pipeline [1] as well as of a Hadoop specific log-aggregation system [13]. These pipelines do not seem to unify operational and business metrics, and Yahoo seems to describe separate infrastructure pieces for real-time and batch consumption. Flume and Scribe are more specifically focused on loading logs into Hadoop, though they are flexible enough to produce output to other systems. Both are based on a push rather than pull model which complicates certain real-time consumption scenarios.

8 Acknowledgements

We would like to acknowledge a few people who were helpful in bringing this system to production: David DeMaggd was responsible for the operation and rollout of LinkedIn's Kafka infrastructure, John Fung was responsible for performance and quality testing, Rajappa Iyer and others on the data warehouse team helped with data model definition and conversion of the warehouse to use this pipeline as well as helping define SLAs and monitoring requirements. In addition we are grateful for the code and suggestions from contributors in the open source community.

References

[1] Mona Ahuja, Cheng Che Chen, Ravi Gottapu, Jörg Hallmann, Waqar Hasan, Richard Johnson, Maciek Kozyrczak, Ramesh Pabbati, Neeta Pandit, Sreenivasulu Pokuri, and Krishna Uppala. Peta-scale data warehousing at yahoo! In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 855–862, New York, NY, USA, 2009. ACM.

[2] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[3] Facebook Engineering. Scribe. https://github.com/facebook/scribe.

[4] Apache Software Foundation. Avro. http://avro.apache.org.

[5] Apache Software Foundation. Flume. https://cwiki.apache.org/FLUME.

[6] Apache Software Foundation. Hadoop. http://hadoop.apache.org.

[7] Apache Software Foundation. Hive. http://hive.apache.org.

[8] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, August 2009.

[9] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[10] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010.

[11] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. *ACM SIGMOD Workshop on Networking Meets Databases*, page 6, 2011.

[12] David Patterson. Latency lags bandwidth. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 3–6, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Ariel Rabkin and Randy Katz. Chukwa: a system for reliable large-scale log collection. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[14] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.

Big Data Storytelling through Interactive Maps

Jayant Madhavan, Sreeram Balakrishnan, Kathryn Brisbin, Hector Gonzalez, Nitin Gupta,
Alon Halevy, Karen Jacqmin-Adams, Heidi Lam, Anno Langen, Hongrae Lee,
Rod McChesney, Rebecca Shapley, Warren Shen
Google Inc.

Abstract

Google Fusion Tables (GFT) brings big data collaboration and visualization to data-experts who possess neither large data-processing resources nor expertise. In this paper we highlight our support for map visualizations over large complex geospatial datasets. Interactive maps created using GFT have already been used by journalists in numerous high-profile stories.

1 Introduction

Much of the current excitement about Big Data refers to performing analytics over terabytes or petabytes of data. However, this typical focus on compute-intensive tasks ignores the data management needs of a large class of non-technical data experts, who are also relevant to the Big Data generation. Prime examples of such users include journalists, NGOs, social scientists, high school teachers, and other domain data activists. These users have access to large interesting data sets, but do not have the resources nor technical expertise to build their own systems, nor the need to run large-scale analytics. Instead their motivation is centered on advocacy through data and their expertise is comparable to that of advanced spreadsheet users. Among other things, to them Big Data implies the ability to tell their stories through compelling visualizations backed by large amounts of data, often stitched together from varied sources.

Google Fusion Tables (GFT) offers collaborative data management in the cloud for such data experts. We emphasize ease of use for tasks such as sharing, collaboration, exploration, visualization and web publishing. We support interactive visualizations, such as maps, timelines, and network graphs, that can be embedded on any web property. Users can update data and those updates will propagate to all uses of the data. We also enable users to easily find and reuse related data sets thereby enabling the integration of data from a myriad of producers.

Thus far, GFT has received considerable use among journalists who embed customized map visualizations as part of their articles. Maps created using GFT are regularly featured in articles published by prominent news sources such as the UK Guardian, Los Angeles Times, Chicago Tribune, and Texas Tribune. In this paper we focus on the scalable serving infrastructure that we built to support interactive maps over large complex geospatial data sets. Not only can viewers zoom, pan, and click on map features, the map presentation can be customized on the fly, reflect near-real time changes to the underlying data, and show the results for a potentially arbitrary viewer query.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

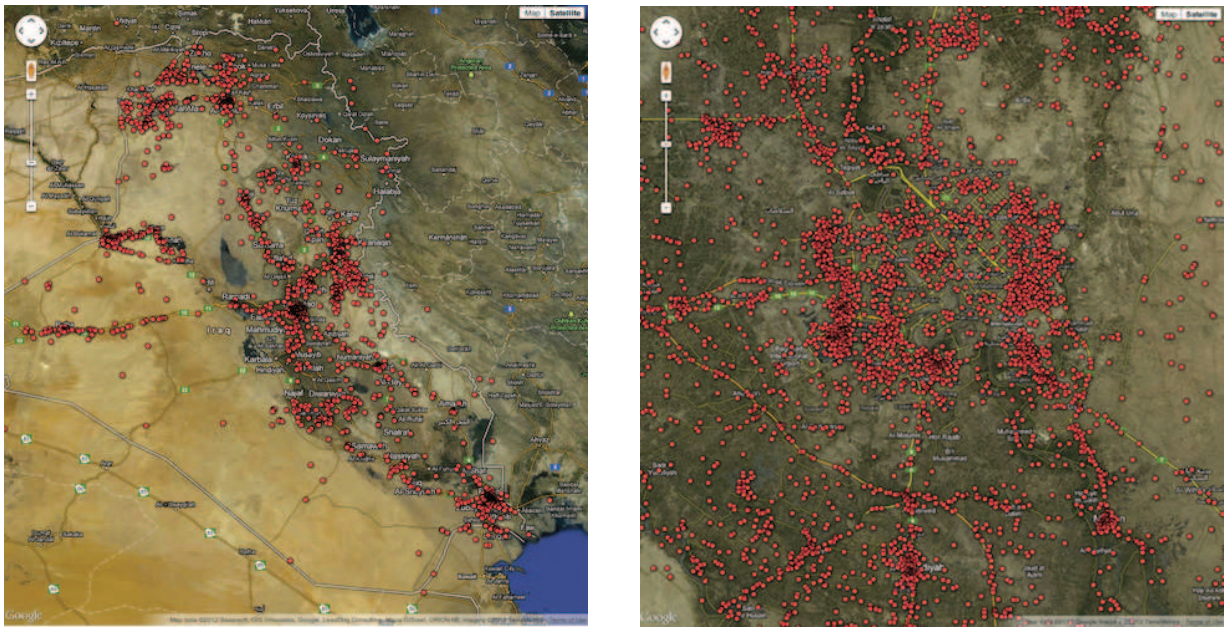

Figure 1: Map from Guardian Datablog article titled *Wikileaks Iraq war logs: every death mapped* [10]. The left shows the entire country, while the right is zoomed into the area around Baghdad. The map plots the more than 50,000 mortality incidents in Iraq recorded in the documents released by Wikileaks.

Through three examples (Figures 1, 3, and 5), we highlight the challenges addressed in order to scale our support to increasingly complex geospatial data. The net result is that we can now support interactive maps of data sets that have millions of features and can be embedded on high-traffic websites. Most importantly, we have relieved our users, the data experts, from the burden of thinking about systems and scalability, thereby letting them focus their efforts on their data quality and storytelling.

Consider our first example. In October 2010, Wikileaks released a collection of documents that have come to be known as the Iraq War Logs. Among other things, the documents list each incident between 2004 and 2009 that resulted in a civilian or military death. The documents were shared with multiple news organizations. As part of their analysis, the UK Guardian Datablog published an interactive map plotting each incident in the Iraq War Logs (Figure 1). The map visualization was backed by a Fusion Table created by the Guardian staff. Readers could zoom in and pan around the map or click on individual points to get specific details about the corresponding incident. The interactive nature of the map enabled their readers to better grasp the severity of the situation on the ground, be it the large numbers of murders in city of Baghdad or the impact of IEDs on the highways.

Embedding a Google Map with custom data was by no means new in 2010. However, the size of the underlying data and the magnitude of the user requests supported made the maps novel. The map in Figure 1 represents more than 50,000 incidents. When the story broke, more than 10 million user requests for this map were served during a 24 hour period, with a peak traffic of more than 1000 QPS.

Maps are only one of the novel features GFT offers. We support other interactive visualizations such as timelines and network graphs. We offer extensive support for the creation of virtual tables that can be created as views over other tables or by merging multiple tables. We also enable users to search for other public tables. A detailed discussion of these features is beyond the scope of this paper.

The rest of the paper is organized as follows. In Section 2, we begin with an overview of our basic map serving architecture. In Sections 3 and 4 we describe how we responded to increasing demands on our system, as users started storing larger data sets and items with more complex geometries. In Section 5, we highlight the

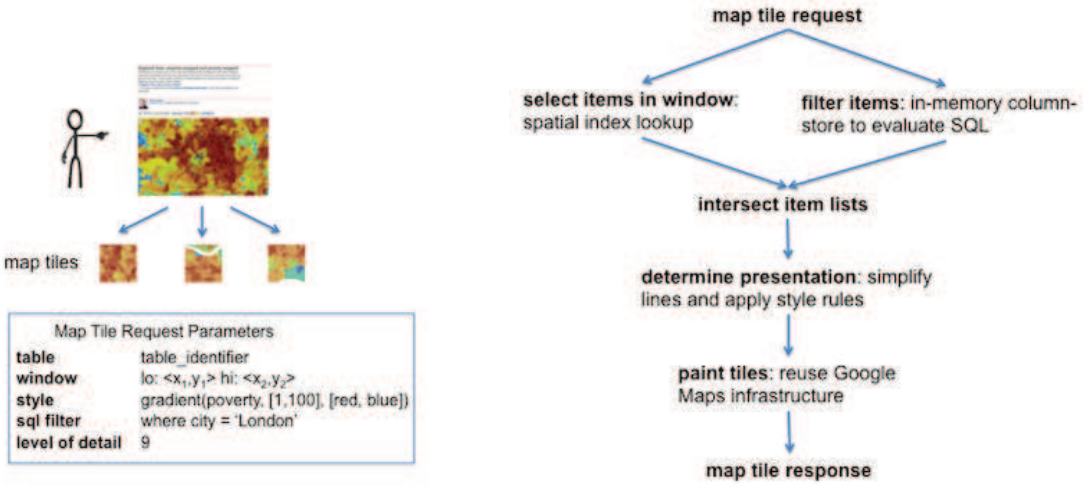

Figure 2: Overview of map processing in GFT. Viewing a map in a browser initiates multiple tile requests (a). Each tile request contains parameters used to compose the response, through the steps outlined in (b).

challenges in supporting merged tables, part of our longer-term goal to bring the worlds structured data together in a useful way. Section 6 concludes.

2 Architecture Overview

GFT lets users store tables of up to 100MB. The main goal of the GFT backend is to support SQL queries (selection, projection, grouping, aggregation, and equijoin) with very low latency and often very high QPS. In addition to the usual data types, we have geospatial data support that lets users store points, lines and polygons to render on a map. The polygons often have very high fidelity (e.g., imagine the polygon describing the boundaries of Canada). Users can set up customized presentations where the shape and color of the features on the map are determined dynamically based on the data.

When a user opens a map, the browser initiates multiple *tile requests* to Google backend servers. The user's viewport is composed of multiple *tiles*, each of which is a 256x256 PNG image. Users interact with the map by zooming in/out or panning around. Each of these actions can lead to more tile requests.

The parameters of a tile request are outlined in Figure 2a. In addition to *table* that identifies the dataset being mapped, the *window* identifies the boundary of the tile. The presentation of each item can be customized by a *style* specification. The style can either be a constant value, a value from a column, or a function (like bucket or gradient) applied to a value from a column. The request can also include an *sql filter*, a conjunction of SQL conditions over various columns of the table. The last parameter, *level of detail* (lod), indicates the zoom level of the tile. The lod specifies the granularity of the tile and ranges from 1 (entire surface of the Earth) to 20 (most zoomed in).

In Figure 2b, we outline the steps involved in computing a tile response. We first determine the items visible in the requested tile with a lookup in a spatial index. In parallel, we evaluate the filter conditions to identify matching items. For items in the intersection of the two lists, we determine the presentation of their geometries. In addition to applying the requested style function, for complex geometries, we try to *simplify* their representation by reducing the number of vertices without losing any fidelity. Finally, we delegate tile rendering to the common Google Maps rendering infrastructure. This rendering component expects a payload that lists the geometry and styling necessary to paint each tile. We discuss the optimizations implemented at each step in the rest of the paper.

3 Scaling to large datasets

Given our close integration with Google Maps, users soon realized the potential in creating interactive maps with large numbers of data-driven features. This led us to focus on some of the scale-up issues involved in serving these visualizations. To ensure the maps are interactive, our servers typically have to answer tile requests within a few hundred milliseconds. In practice, our goal is to answer requests under 10 milliseconds, except in the case of large datasets with very complex geometries. In this section we describe two aspects of our system implemented to achieve fast responses: (1) a highly optimized in-memory column store and (2) a representative and consistent sample of the underlying data per tile.

In-memory Column Store: To meet our millisecond response time goal, we implemented an in-memory query processing system. However, given our desire to support visualizations over a large number of tables (with limited resources), our processing is implemented over an LRU cache that loads tables only on demand. Tile requests typically only access a small number of columns in the underlying table; the only required column is the one defining the geometry to be mapped. Filters and styles are optional. If present in a request, filters and styles typically only require one additional column in the table. We implemented a cache that is managed in terms of column indices. Only columns that have been accessed by active map requests are maintained in memory. Using column-based indices also enables custom compaction algorithms to ensure that even million-row tables have a very compact memory footprint.

The most crucial column index is the spatial index that enables us to perform fast lookups of geometries that intersect any requested tile. This is implemented as a *one-dimensional* index based on the idea of Hilbert curve encoding [7]. An overview of our spatial query processing is presented in [5].

Consistent Spatial Sampling: To ensure quick rendering of individual tiles by the Google Maps infrastructure, the payload for each tile has to be limited to a few hundred items with at most a few thousand vertices. Polygons and lines can have many vertices. Large datasets, especially when zoomed out, can have numerous items per tile. In such cases, we respond with a representative sample of the underlying dataset.

Responding with too many items can sometimes be counter-productive and present a undesirable cognitive load on viewers. When looking at the entire map in Figure 1, all that a viewer can possibly perceive is a dense distribution of points. Hence, it is unnecessary to plot every single point. Instead, it is necessary that the chosen sample is sufficiently representative of the underlying distribution that it looks right to the viewer familiar with the data.

To ensure that tiles are globally consistent, sampling cannot be performed in isolation for each tile request. For example, as a viewer zooms in previously visible items must be retained as new items are displayed. Likewise, items can span multiple adjacent tiles, and hence when panning any such item must either be included in all the adjacent tiles, or in none. To achieve such consistent sampling, we associate a minimum level of detail l_i^m with each data item d_i. For a tile request with level of detail l, we only include items that have at least a minimum level of detail of l, i.e., only if $l \geq l_i^m$. To facilitate efficient query processing, the value l_i^m is stored in the spatial index and thus only items known to be visible at l are retrieved during the spatial lookup.

When sampling point-only datasets, in general, the points can be considered equally important. However, the same is not the case for line and polygons datasets. For example, when zoomed out on a map of islands of the world, excluding Greenland or Madagascar leads to a very misleading and unrepresentative map. We use a weighted sampling scheme where polygons are biased by their area and lines by their length.

In [13], we describe our solution for efficient spatial sampling. We show that spatial sampling in our context can be modeled as an integer logic program (ILP). We present optimizations that make the problem more tractable and also describe practical randomized solutions.

The in-memory column indices and spatial sampling ensure that for datasets with as many as a million features, we can support interactive maps with SQL filters and custom styling. For example, the tile requests for the map in Figure 1 can typically be answered in less than one millisecond. In Section 4, we present the results

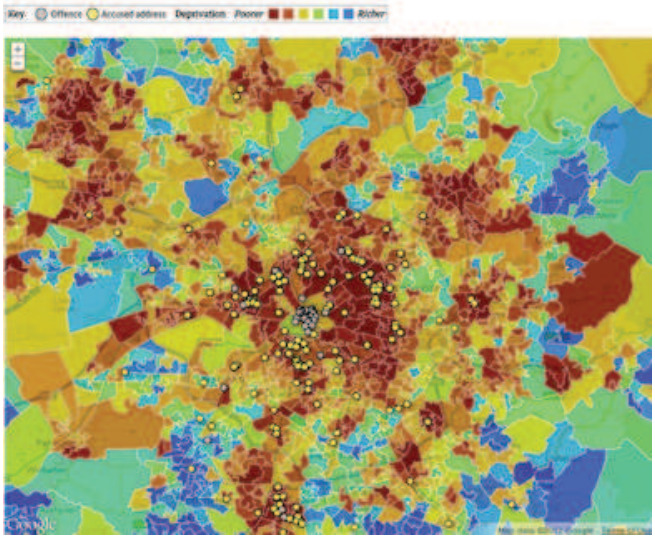

Figure 3: Map from the Guardian Datablog article titled *England riots: suspects mapped and poverty mapped* [11]. The map overlays two datasets. The colored jigsaw-like pattern is a poverty map that paints England wards based on poverty rate. Mapped on top is another dataset with the locations of rioting incidents (grey) and rioting suspects (yellow). The poverty map includes about 32,000 polygons with a total of 1.8 million vertices. The rioting dataset includes about 2100 incidents.

of a performance analysis for a more complex dataset.

4 Scaling to massive and complex polygon datasets

Given the ability to map large point datasets, it was natural that we soon received requests for similar support for maps with complex line and polygon geometries. To illustrate the next level of optimizations that were necessary, consider another post from the Guardian Datablog.

The map in Figure 3 was embedded in the article *England riots: suspects mapped and poverty mapped*, published after the 2011 riots in cities across England. The map shows the wards in England colored by poverty rate. Overlaid are incidents of rioting and the addresses of rioting suspects. The article observes correlations between poverty, disenchantment, and subsequent unrest. The poverty dataset included about 32,000 polygons and a total of about 1.8 million vertices. We now look at some of the optimizations we implemented to support such an intricate map.

Line and polygon simplification: As mentioned in the last section, to reduce rendering time, we limit the payload for each tile. However, sampling of items alone does not suffice for high quality line and polygon datasets that can have hundreds of vertices per individual polygon. For example, in Figure 3, there are 18 wards whose boundaries are described by more than 500 vertices and 960 wards whose boundaries are described by more than 200 vertices. Not all vertices are necessary when rendering the polygons, especially when zoomed out. Our first approach was to use well-understood line simplification algorithms (e.g., Douglas-Peucker [4]). While appropriate for lines, such approaches proved insufficient for polygons: simplifying each boundary independently may result in glaring holes between adjacent polygons on a space-filling map like Figure 3.

Our simplification algorithm instead is based on the idea of tile projection. Briefly, given a level of detail, we project lines onto a discrete 256 x 256 grid that represents the eventual tile. If adjacent points project onto the same grid location, they can be collapsed. In addition to being efficient (linear time versus the $O(n \log n)$ running time of the Douglas-Peucker algorithm), this simple approach ensures that (1) lines and polygons can be rendered onto tiles with no perceivable loss in fidelity, (2) there are guaranteed to be no holes between adjacent polygons, and (3) lines are simplified differently and appropriately at different levels of detail.

Effort-based response caching: Though linear in the number of vertices, line simplification can still be computationally expensive. This is especially the case when there are multiple large polygons in tiles at low levels of detail. In practice, we consider a request to be expensive if it takes more than 100ms to compute the payload for

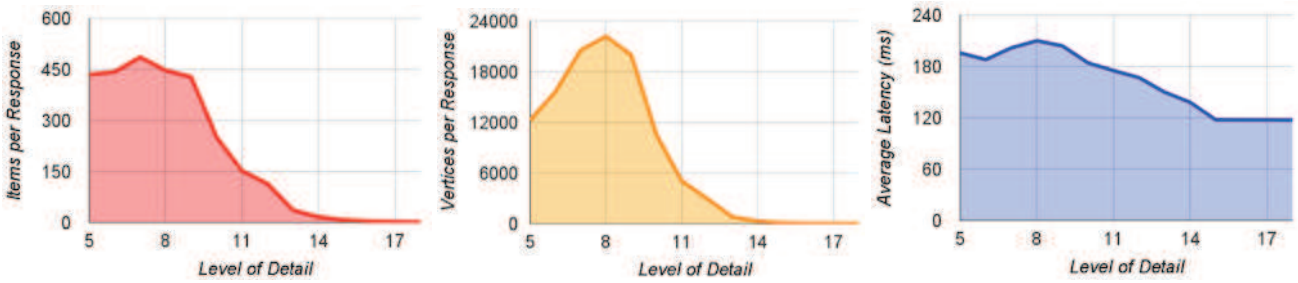

Figure 4: Distribution of average items per response, vertices per response, and GFT server latency with changes in the level of detail for the map in Figure 3. The latency was measured in an experimental setup. To isolate the performance of our own backend from the tile rendering delegated to the Google Maps infrastructure, we do not include the tile rendering time in the measured latencies. As the charts indicate, we never return more than 500 items per tile, never more than 22,000 vertices per tile, and are able to do so within 220 ms (100 ms plus approx 120 ms network latency).

the tile (retrieval + filtering + styling + simplification). At high request rates (more than 1000 per sec), this can significantly increase the CPU load and thereby lead to further degradation in response times. To facilitate scaling to high loads, we cache tile responses. If computing the payload for a tile response takes more than 100ms, we cache the response for a small amount of time. Thus, we only cache potentially expensively responses in heavy load situations.

Effort-based response caching is necessary to support a map like Figure 3 on the main page of a prominent newspaper. We generally avoid aggressive caching of responses on our servers to reduce memory use. We also limit public caching of tile responses due to limitations in the Google Maps API and our need to support near real-time updates on user tables.

Performance Analysis: We now look at the performance implications of using sampling, simplification and in-memory processing for the poverty map in Figure 3. The charts in Figure 4 show the distribution of response sizes and latencies with changing levels of detail. From the first chart, we see that sampling ensures that we never return more than 500 items per tile and that sampling comes into play when the level of detail is below 9. From the middle chart, we see that the average number of vertices per tile tops out at 21,000 at level of detail 8. Simplification plays a critical role below level of detail 8: the sample 450-500 polygons per tiles can be rendered, but with far fewer vertices. The last chart shows the response times (excluding tile rendering time and with caching disabled) as measured at a remote client. We estimate a network delay of about 120 ms. Thus, we have a maximum server processing time of about 100ms, and at higher levels of detail, it barely takes any time. Since the tile rendering payload is limited in size, the tiles are created efficiently. We are able to deliver fairly complex maps with low latencies, thereby ensuring interactivity.

Pre-computing Covers: Lastly, we consider geometries that are much more complex than those in the poverty map. Consider the maps in Figure 5 that were published by The Nature Conservancy (TNC). As an environmental organization, one of the TNCs goals is promoting awareness about critical environmental issues. Their website today includes a large number of maps and charts that display various properties of the different *eco-regions* of the world. The underlying eco-regions dataset (customized in two different ways in Figure 5) includes about 14,500 polygons with a total of 4.3 million vertices. These polygons are significantly more complex than those of English wards in Figure 3. In contrast with the mostly convex boundaries of the English wards, the eco-region boundaries follow natural features that define more convoluted shapes. Specifically, the TNC maps include 716 regions that have more than 1000 vertices describing their boundaries and 59 that have more than 10,000 vertices.

The TNC datasets are classic examples of high quality datasets that only get sporadic traffic. As mentioned

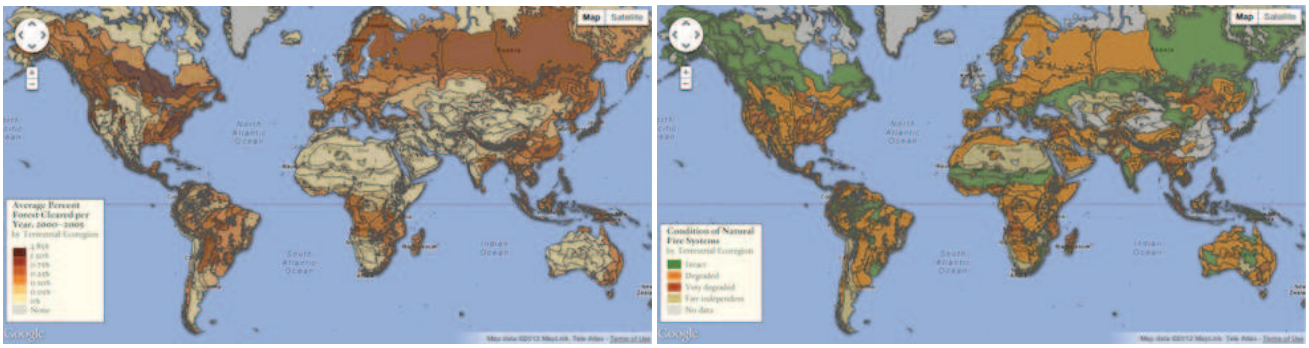

Figure 5: The Nature Conservancy maps showing the percent of forest cleared and conditions for natural fires in various eco-regions of the world [15]. The underlying eco-regions dataset has about 14,500 complex polygons with over 4.3 million vertices.

in Section 3, the column indices are loaded on demand. When the map is accessed for the very first time (or after a long time), the dataset has to be loaded into memory to create the compact in-memory indices. The first step in constructing the crucial spatial index is to compute the Hilbert *cover* for each of the polygons (details in [5]). Computing covers is an optimization problem and can be very expensive when geometries are large and irregular, as these are. As a result, it can take many seconds before the index can be computed and compacted. This leads to bad user experience as the map has to either be left empty or with a *in-progress* indicator. To address this challenge, we pre-compute the covers for large polygon datasets as a background process. When the map is accessed for the first time, the covers are directly loaded to construct the spatial index. This enables us to limit tile response times to less than a few hundred milliseconds in the vast majority of cases. Further reducing the delays when loading complex datasets is a topic on ongoing work.

5 Scaling by Merging

As stated at the outset, an important goal of GFT is to allow users to piece together disparate data sets to obtain new insights. The mechanism we currently offer to achieve that is to create *merged tables*. A merged table is essentially a view with an equi-join over any number of tables on a common key. Merged tables can be manipulated as any other table, including the ability to create embedded maps. Merged tables can lead to interesting modes of collaboration where other users can be granted privileges on partial views of the underlying data. Merged tables in GFT are live, i.e., not materialized. Thus updates made to the underlying tables are propagated to the merged table and queries over the merge are rewritten and executed over their base tables.

Consider the example of the TNC global conservation maps website [15]. There are a total of about 60 maps that paint the different eco-regions of the world according to various metrics (two examples are shown in Figure 5). Each of these maps has several thousand polygons and many million vertices. The resources required to serve these maps is significant, especially given that each map does not individually attract sufficient traffic to merit dedicated resources. However, all of these maps are created from only three distinct base tables that define the polygons for *terrestrial* [9], *marine* [14], and *freshwater* eco-regions [1]. The table backing each map is then created by merging one of the three base tables with another table that has columns specific to that particular map.

Merged tables offer many advantages. First, they offer the opportunity for significant reductions in resource use, while also ensuring better overall performance. Queries over merged tables are re-written as queries over the base tables and reuse the spatial index for the base table column. Given the complexity of the polygons, this leads to a large reduction in the memory footprint for storing indices. There is also a reduction in the index cache miss rate and hence the need to reload the datasets. The reduced footprint also frees up the in-memory

column store to cache indices for other tables.

Second, merged tables are well suited in scenarios where the underlying data originates from different sources. The TNC, for example, works with a number of partner organizations, each of whom provides them with the data for a specific map. The partner organizations can retain ownership and edit privileges over the base tables they contribute, while the TNC is able to merge them with their eco-regions tables to create publishable maps.

Lastly, merged tables offer a better experience when mapping datasets that are likely to be updated. For example, GFT is used to host election maps where electoral districts are colored according to the live vote counts received by various candidates. See [12] and [8] for examples in the context of the recent Turkish national elections and the Iowa caucuses respectively. In such datasets, the geometries of electoral districts remain unchanged, while the vote tallies vary. To set up such a map, the electoral district geometries and vote tallies are stored in two separate tables that are then merged. The indices for the more complex geometry table never change, while only the much more compact vote tallies data has to be periodically reloaded. The details of managing updates is beyond the scope of this paper.

Merged tables have similarly been used in crisis response scenarios to map the availability or status of various resources in disaster zones. For example, GFT was used to track road closures and rolling brownouts in Japan after the 2011 tsunami [3] and road flooding in Vermont during the aftermath of Hurricane Irene [2].

6 Conclusion

Non-technical data users such as journalists, NGOs, social scientists and activists have Big Data challenges that are ignored by the typical focus on analytics and off-line computation. Although their individual tables may be small they can be combined with data from a vast pool of public structured data. Their computation requirements are driven by the need to serve high QPS, low-latency, interactive visualizations of large data sets that may be actively updated. Google Fusion Tables was developed with the needs of these users in mind. It provides them the tools and infrastructure to integrate multiple related data sets, interactively visualize this data and eventually embed the visualizations as part of their storytelling.

Based on real world examples, we discussed the specific challenges we faced to build an infrastructure to support interactive customizable maps over large complex geospatial data sets. In parallel we are also developing infrastructure to enhance the ability of our users to find and combine with other relevant public data sets, either within GFT or as HTML tables on the Web and visualize it in novel ways regardless of data size. More examples of prominent uses of GFT can be found in our gallery [6].

References

[1] R. Abell, M. L. Thieme, C. Revenga, M. Bryer, M. Kottelat, N. Bogutskaya, B. Coad, N. Mandrak, S. C. Balderas, W. Bussing, M. L. J. Stiassny, P. Skelton, G. R. Allen, P. Unmack, A. Naseka, R. Ng, N. Sindorf, J. Robertson, E. Armijo, J. V. Higgins, T. J. Heibel, E. Wikramanayake, D. Olson, H. L. López, R. E. Reis, J. G. Lundberg, M. H. S. Pérez, and P. Petry. Freshwater Ecoregions of the World: A New Map of Biogeographic Units for Freshwater Biodiversity Conservation. *BioScience*, 58(5), May 2008.

[2] Vermont Flooding 2011. http://crisislanding.appspot.com/?crisis=2011_flooding_vermont, August 2011.

[3] http://www.google.com/intl/ja/crisisresponse/japanquake2011_traffic.html, March 2011.

[4] D. Douglas and T. Peucker. Algorithms for the reductions of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2), December 1973.

[5] H. Gonzalez, A. Halevy, C. Jensen, A. Langen, J. Madhavan, R. Shapley, and W. Shen. Google Fusion Tables: Data Management, Integration, and Collaboration in the Cloud. In *SOCC*, 2010.

[6] Google Fusion Tables: Example Gallery. https://sites.google.com/site/fusiontablestalks/stories.

[7] D. Hilbert. Ueber die stetige Abbildung einer Linie auf ein Flchenstck. *Mathematische Annalen*, 38, 1891.

[8] J. Keefe. Making AP election data easy with Fusion Tables. http://johnkeefe.net/making-ap-election-data-easy-with-fusion-tabl, January 2012.

[9] D. M. Olson, E. Dinerstein, E. D. Wikramanayake, N. D. Burgess, G. V. N. Powell, E. C. Underwood, J. A. D'Amico, I. Itoua, H. E. Strand, J. C. Morrison, C. J. Loucks, T. F. Allnutt, T. H. Ricketts, Y. Kura, J. F. Lamoreux, W. W. Wettengel, P. Hedao, and K. R. Kassem. Terrestrial Ecoregions of the World: A New Map of Life on Earth. *BioScience*, 51(11), November 2001.

[10] S. Rogers. Wikileaks Iraq war logs: every death mapped. http://www.guardian.co.uk/world/datablog/interactive/2010/oct/23/wikileaks-iraq-deaths-map, October 2010.

[11] S. Rogers. England riots: suspects mapped and poverty mapped. http://www.guardian.co.uk/news/datablog/interactive/2011/aug/16/riots-poverty-map, August 2011.

[12] C. Sabnis. Visualizing Election Results Directly from the Ballot Box. http://googlepublicsector.blogspot.com/2011/06/visualizing-election-results-directly.html, June 2011.

[13] A. D. Sarma, H. Lee, H. Gonzalez, J. Madhavan, and A. Halevy. Efficient Spatial Sampling of Large Geographic Tables. In *SIGMOD*, 2012.

[14] M. D. Spalding, H. E. Fox, G. R. Allen, N. Davidson, Z. A. Ferdaña, M. Finlayson, B. S. Halpern, M. A. Jorge, A. Lombana, S. A. Lourie, K. D. Martin, E. Mcmanus, J. Molnar, C. A. Recchia, and J. Robertson. Marine Ecoregions of the World: A Bioregionalization of Coastal and Shelf Areas. *BioScience*, 57(7), July 2007.

[15] The Nature Conservancy: Global Conservation Maps. http://maps.tnc.org/globalmaps.html.

General Chairs
John C.S. Lui, Chinese U of Hong Kong
Wei-Tek Tsai, Arizona State University
Paul Watson, Newcastle University

Program Committee Chairs
Roy Campbell, UIUC
Hui Lei, IBM Watson Research
Volker Markl, TU Berlin

Program Committee
Aditya Akella, U of Wisconsin-Madison
Jean Bacon, University of Cambridge
Roger Barga, Microsoft Research
Azer Bestavros, Boston University
Sara Bouchenak, INRIA
Elisa Burtino, Purdue University
Jiannong Cao, Hong Kong Polytechnic U
Michael Carey, UC Irvine
Gregory Chockler, IBM Haifa Research
Yeh-Ching Chung, National Tsing Hua U
Paolo Costa, Imperial College London
Rodrigo Fonseca, Brown University
Geoffrey Fox, Indiana University
Phillip Gibbons, Intel Labs
Garth Gibson, CMU and Panasas Inc
Xiaohui Gu, North Carolina State U
Indranil Gupta, UIUC
Mor Harchol-Balter, CMU
Joseph Hellerstein, Google Research
Hai Jin, Huazhong U of Sci and Tech
Anthony Joseph, UC Berkeley
Masaru Kitsuregawa, University of Tokyo
Erwin Laure, Royal Institute of Technology
Justin Levandoski, Microsoft Research
Wei Li, Beihang University
Baochun Li, University of Toronto
Ling Liu, Georgia Institute of Technology
Stefan Manegold, CWI Amsterdam
Dejan Milojicic, HP Labs
Archan Misra, Singapore Management U
Klara Nahrstedt, UIUC
Priya Narasimhan, CMU
Peng Ning, North Carolina State U
Beng Chin Ooi, National U of Singapore
Manish Parashar, Rutgers University
Guillaume Pierre, VU Amsterdam
Donald Porter, Stony Brook University
Berthold Reinwald, IBM Almaden Research
Stefan Tai, Karlsruhe Institute of Technology
Florian Waas, EMC
Haixun Wang, Microsoft Research - Asia
Jon Weissman, University of Minnesota
Dan Williams, IBM Watson Research
Jeffrey Xu Yu, Chinese U of Hong Kong

Publicity Chairs
Badrish Chandramouli, Microsoft Research
Lin Gu, Hong Kong U of Sci and Tech
Graham Morgan, Newcastle University

Publication Chair
Yinong Chen, Arizona State University

Web Chair
David Webster, University of Leeds

Steering Committee Contact
Jie Xu, University of Leeds

IEEE International Conference on Cloud Engineering (IC2E 2013)

March 25-27, 2013 ◆ San Francisco, USA

www.engineering.leeds.ac.uk/computing/conferences/IC2E2013

Important Dates

Abstracts due: Sept 7, 2012
Full papers due: Sept 14, 2012
Author notification: Dec 7, 2012

IEEE COMPUTER SOCIETY

Call For Papers

The IEEE International Conference on Cloud Engineering (IC2E) is a new conference that seeks to provide a high-quality and comprehensive forum, where researchers and practitioners can exchange information on engineering principles, enabling technologies, and practical experiences as related to cloud computing. By bringing together experts that work on different levels of the cloud stack - systems, storage, networking, platforms, databases, and applications, IC2E will offer an end-to-end view on the challenges and technologies in cloud computing, foster research that addresses the interaction between different layers of the stack, and ultimately help shape the future of cloud-transformed business and society.

Topics of interest include, but are not limited to:

- Infrastructure as a service (IaaS)
- Platform as a service (PaaS)
- Database as a service (DaaS)
- Software as a service (SaaS)
- Network as a service (NaaS)
- Business process as a service (BPaaS)
- Security as a service
- Storage as a service
- Information as a service
- Big data management and analytics
- Virtualization technology
- Performance, dependability and service level agreements
- Cloud security, privacy, and compliance management
- Workload deployment and migration
- Energy management in cloud centers
- Cloud programming models and tools
- Hybrid cloud integration
- Service lifecycle management
- Service management automation
- Metering, pricing, and software licensing

29th IEEE International Conference on

DATA ENGINEERING

BRISBANE, AUSTRALIA | 8 – 11 APRIL 2013

A leading forum for researchers, practitioners, developers and users to explore cutting-edge ideas and to exchange techniques, tools and experiences.

GENERAL CHAIRS

Rao Kotagiri *The University of Melbourne, Australia*

Beng Chin Ooi *National University of Singapore, Singapore*

PROGRAM COMMITTEE CHAIRS

Christian S. Jensen *Aarhus University, Denmark*

Chris Jermaine *Rice University, USA*

Xiaofang Zhou *The University of Queensland, Australia*

IMPORTANT DATES

Research & Industry Papers
Abstracts due
JULY 16 2012

Full paper submissions due
JULY 23 2012

Notification to authors
OCTOBER 14 2012

Final versions due
NOVEMBER 30 2012

www.icde2013.org

IEEE computer society

icde 2013

29th IEEE International Conference on
DATA ENGINEERING
BRISBANE, AUSTRALIA | 8 – 11 APRIL 2013

Images courtesy of Brisbane Marketing and Tourism Australia

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398