Automatic Management of Partitioned, Replicated Search Services

Florian Leibert, Jake Mannix, Jimmy Lin, and Babak Hamadani
Twitter
795 Folsom St.
San Francisco, California
@flo @pbrane @lintool @babak_hamadani

ABSTRACT

Low-latency, high-throughput web services are typically achieved through partitioning, replication, and caching. Although these strategies and the general design of large-scale distributed search systems are well known, the academic literature provides surprisingly few details on deployment and operational considerations in production environments. In this paper, we address this gap by sharing the distributed search architecture that underlies Twitter user search, a service for discovering relevant accounts on the popular microblogging service. Our design makes use of the principle that eliminates the distinction between failure and other anticipated service disruptions: as a result, most operational scenarios share exactly the same code path. This simplicity leads to greater robustness and fault-tolerance. Another salient feature of our architecture is its exclusive reliance on open-source software components, which makes it easier for the community to learn from our experiences and replicate our findings.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Design, Performance, Reliability

Keywords

information retrieval, distributed retrieval architectures, configuration management, failover, robustness

1. INTRODUCTION

It is widely known that caching, partitioning, and replication are three primary strategies for scaling up large, distributed web services (e.g., search) with high-throughput and low-latency requirements. Although these approaches are well-known both in the systems literature as "design patterns" for building large-scale distributed systems [9, 4] and application fields such as information retrieval (IR) [3, 1] and database systems [7], we find that the academic literature is missing discussions of issues that are critical to their operation in real-world production environments. Focusing in particular on partitioning and replication, some of these important questions include:

- How do we manage partition and replica configurations in a manner that minimizes human involvement and increases robustness?
- How do we coordinate automatic, seamless failover of partition and replica servers in the presence of unreliable commodity hardware?
- How do we dynamically adjust configurations in response to increasing (or decreasing) load in an elastic manner?
- How do we deploy new code and refresh backend indexes in a robust manner with minimal service disruption?

Our goal is to share experiences in answering the above questions, in the context of user search, a production service that has been running since March 2011 at Twitter. We hope that this paper fills gaps in the academic literature and provides a foundation on which others can build.

The solution that we describe makes extensive use of ZooKeeper [10], a wait-free coordination system originally developed by Yahoo! and is now an open-source project. We make use of the design principle that eliminates the distinction between failures and other anticipated service disruptions (e.g., code deployments, configuration changes, etc.). This philosophy is espoused by Armando Fox, as conveyed by James Hamilton in his classic article "On Designing and Deploying Internet-Scale Services" [9]. Erasing the distinction between anticipated and unanticipated service disruptions simplifies code paths and ensures that fault-recovery mechanisms are exercised regularly, thus increasing the robustness of the system as a whole.

Another salient feature of our architecture is that it is entirely built on open-source software components. Search capabilities are provided by the Lucene search engine. Transport and RPC between individual components are handled by Thrift. Lucene indexes are built from relevance signals mined using Hadoop, the open-source implementation of MapReduce [6], via Pig, a high-level dataflow language for expressing large-data operations [14]. Since our entire software stack is built on freely available components, it becomes easier for others to apply the lessons learned to similar services and scenarios.

SOCC'11, October 27–28, 2011, Cascais, Portugal.

The primary contribution of this paper is the explication of many production and operational issues associated with partitioned, replicated search services. Many IR textbooks (e.g., [12, 5]) provide an overview of how search engines are scaled up by partitioning the indexes and replicating the partitions, and IR research papers take this for granted (e.g. [13]). However, there is scant discussion of how these general principles are put into practice, for example, the questions raised above. In a way, the architecture described in this paper may be viewed as a reference implementation that fleshes out what many IR researchers and practitioners think they know, but the details of which may be a bit fuzzy and have never been thoroughly documented.

The remainder of the paper is organized as follows: Section 2 describes design requirements of the service and provides an overview of the state of knowledge with respect to building distributed search services. Section 3 details the design of our architecture and its operational lifecycle. Experiences in production deployment are presented in Section 4. We discuss future work in Section 5 before concluding.

2. BACKGROUND

The context of our work is user search in Twitter, the popular microblogging service whereby users can send short, 140-character messages, called "tweets", to their "followers" (other users who subscribe to the messages). Conversely, users can receive tweets from people they follow via a number of mechanisms, including web clients, mobile clients, and SMS. As of June 2011, Twitter has over 200 million users who collectively post over 200 million tweets per day.

We begin this section with a discussion of the requirements of our application and then proceed to overview well-known design principles for scaling up distributed search services. We close with a discussion of gaps in the current academic literature that we've identified—these are questions to which our work contributes.

2.1 Application Requirements

User search is a recently-introduced functionality in Twitter that allows users to find other users either by name (i.e., directory search) or by expertise on a particular topic.[1] Example of topical queries range from broad ones such as "sports", "gardening", and "wine", to narrow interests such as "hadoop" or "cupcake recipes".

The design goals of Twitter user search include seamless scaling up to loads on the order of tens of thousands of queries per second (although the initial launch configuration handled only a fraction of that). Each individual request has a target 99th percentile latency of 200ms and a target mean latency of roughly a tenth of that. We aimed to design a "zero-configuration" system that minimizes human involvement in all aspects of its operation: setup, error handling, code deployments, index updates, etc. Finally, we assume a single datacenter environment.

Early in the design process, we explicitly decided against real-time or near real-time index updates as a requirement, which means there may be a noticeable delay between certain actions and their impact on the service: for example, the retrieval algorithm considers a user's profile description, but edits to it will not be immediately reflected. This decision was made for two main reasons. First, user search, like general web search, is inherently adversarial, in that there will always be attempts to "game" the ranking algorithm so that certain users will be ranked higher than otherwise warranted. Real-time index updates provides a rapid feedback cycle that benefits attackers. Second, a less-stringent update requirement allows us to compile relevance signals offline, thus broadening the range of algorithms that can be brought to bear and the amount of data we can consider.

[1] http://blog.twitter.com/2011/04/discover-new-accounts-and-search-like.html

2.2 General Architecture

Partitioning, replication, and caching are three strategies for scaling large, distributed search services with stringent throughput/latency requirements. In more detail, they are:

Partitioning (also called sharding) involves splitting the document collection into disjoint segments and building separate indexes over each of these partitions. Each index is assigned to a separate index server at runtime for serving queries. A broker is responsible for forwarding requests to *all* of the partition index servers and integrating partial results from each before passing the final output back to the requester. Partitioning is employed primarily for two reasons: to exploit parallelism in the index construction process, and to reduce query latency at retrieval time.

In our case, indexes are served from main memory, and RAM costs on a single machine are nonlinear beyond a certain point. If the index size doubles, the same performance (throughput and mean latency) can be maintained by doubling the number of partitions. Without partitioning, the original hardware requires (quickly nonlinear) upgrade costs to maintain performance. To a first order approximation, query execution time is proportional to index size, so a partitioned architecture exploits parallelism to reduce latency. From a fault tolerance point of view, partitioning is also helpful since service quality degrades gradually as partitions are knocked offline or timeout under heavy load.

Replication involves multiple instances of the same service running independently to share the incoming request load. In a standard distributed search architecture, there will be multiple replicas serving the same index partition, so that the broker needs to query only a covering set of all partitions (i.e., select one from the set of available replicas). Typically, the broker itself is replicated to eliminate single points of failure. Replication has little impact on latency and is primarily exploited to increase throughput and fault tolerance.

Caching involves storing request results in a more readily-accessible manner (e.g., for on-disk data, cache in memory) to decrease latency and increase throughput. Information retrieval researchers have explored both caching of postings lists and caching of results [2]. In the case where indexes are completely held in memory (e.g., as is the case with our service), caching postings becomes mostly unnecessary. Typically, there will be a number of cache servers, independent from the index servers: these are consulted by the broker in lieu of executing a query from scratch.

Partitioning, replication, and caching are typically integrated into a distributed search architecture in the manner shown in Figure 1. The partitions are shown in columns and replicas in rows in the main box, with cache servers off to the side. A number of replicated brokers is responsible for servicing requests (e.g., from the frontend) by either consulting the cache or dispatching queries to the partitions servers and

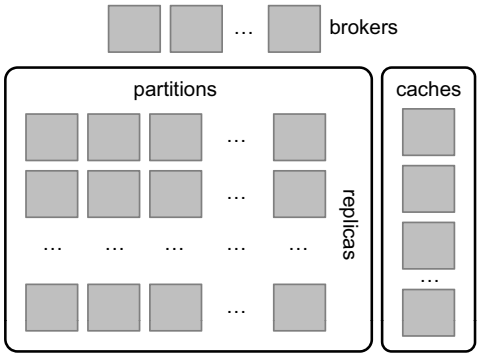

Figure 1: Illustration of a standard distributed search architecture that takes advantage of partitioning, replication, and caching.

integrating the results. Note that this organization is logical as opposed to physical: in the simplest case, there is a one-to-one mapping between index servers and physical machines, but this need not be the case. For example, multiple logical index servers may run on the same physical machine, either as different processes or mediated through a virtualization layer. Another common pattern is to have a subset of partition servers (or possibly all of them) perform "double duty" as brokers, or embed the broker logic directly in client code, since brokers consume few resources in practice.

In this paper, we primarily focus on partitioning and replication, and leave aside issues related to caching, which have been studied in detail [2, 15] and is not directly relevant to the goals of this work. For simplicity, we opted for a one-to-one index server to physical machine mapping and use other techniques to more fully utilize resources: index servers are multi-threaded to exploit multi-core processors, and partitions are appropriately-sized to fit in memory. However, the techniques discussed in the paper can be easily adapted to alternative logical-to-physical mappings. Finally, we assume a cluster of homogeneous machines—which implies (roughly) equal partition sizes and that each machine is able to handle an equal share of the query load.

2.3 Underexplored Issues

So far, everything we have described is considered common knowledge in building distributed search systems and can be found in standard IR textbooks [12, 5]. However, we found that the academic literature is missing discussion of aspects that are critical for operation in production environments. Several of these key issues are outlined below:

Configuration management. Naturally, there must exist some mechanism for managing the mapping between index servers and the partitions they are serving. The simplest solution is a static configuration file, whereby these mappings are specified explicitly. For convenience, we refer to this as the static assignment approach. At scale in a production environment, this approach becomes difficult to manage. Since a service can span dozens to hundreds of servers (and possibly more), static configuration involves keeping track of many hostnames, ports, index file locations, and other configuration data. Provisioning of additional physical machines would involve manually editing a configuration file and then "refreshing" the state of the service, which can be an error-prone process. In contrast, we describe a dynamic index server assignment strategy that significantly simplifies configuration management: index servers need only one datum, the location of ZooKeeper. All other aspects of partitioning and replication management are handled automatically, as we describe in the next section.

Automatic failover. It is common wisdom that building web-scale services from commodity machines is significantly more cost-effective than from high-end servers [3, 9, 4]. However, commodity machines are less reliable, which places a greater burden on the software to provide fault tolerance. Of course, the general approach is to provide replication, but *how* exactly? Once again, the academic literature has surprisingly little to say in this regard.

Failover mechanisms interact with configuration management in potentially complex ways—for example, a static assignment strategy would require some way to synchronize the state of the service during normal operation and the actual state in the presence of machine failures. In contrast, we describe a coordination system that continuously monitors system state and a robust automatic failover mechanism that requires no manual intervention.

Code deployments and index updates. A service is never static, but rather evolves as bugs are fixed and new features are introduced. Similarly, a distributed search service cannot serve the same index for too long, as data rapidly becomes stale. These two issues require code deployments and index updates on production systems while the service is still handling live load. Obviously, code deployments can be scheduled during off-peak hours, but that comes with its own drawbacks: off-peak hours means that there are few engineers around to troubleshoot if something goes wrong.

Although the general design of partitioned, replicated search services are well-known, we find a dearth of published knowledge on operational issues such as those raised above. This paper shares our experiences in wrestling with these issues, which we hope to be broadly useful to the community.

3. DESIGN

To better understand our design, it is necessary to describe the life cycle of our user search service. At a high level, the service decomposes into processes for gathering relevance signals and building the inverted indexes on the analytics backend and the distributed retrieval architecture for serving queries in real time. The retrieval architecture presents an interface to the frontend (e.g., twitter.com site) that handles the interface, rendering results, etc.

3.1 Index Construction

Before inverted indexes can be built, relevance signals for ranking users must be gathered, filtered, and analyzed. These operations are performed on our Hadoop analytics cluster. Data are written to the Hadoop Distributed File System (HDFS) via a number of real-time and batch processes in a variety of formats (text, delimited records, JSON, Protocol Buffers, Thrift, etc.). These data can be bulk exports from databases (tweets, user profiles, the interest graph), application logs, interaction logs, and more. Note that the analytics cluster serves a variety of purposes beyond processing relevance signals for user search; other tasks include regular batch jobs that feed report generation systems and online dashboards, as well as *ad hoc* jobs such as machine learning experiments by the research group.

The user search algorithm draws from a variety of signals, including tweets, user profiles, the interest graph, as well as other sources. These signals are gathered, filtered, analyzed, and refined by a collection of scripts written in Pig, a high-level dataflow language that compiles into physical plans that are executed on Hadoop [14]. Pig provides concise primitives for expressing common operations such as projection, selection, group, join, etc. This conciseness comes at low cost: Pig scripts approach the performance of programs directly written in Hadoop Java. Yet, the full expressiveness of Java is retained through a library of custom UDFs (user-defined functions) that expose core Twitter libraries and other custom code. The refined relevance signals are written back to HDFS to be indexed; information associated with each user is treated as a "document".

We have built a general purpose inverted indexing framework around the open-source Lucene search engine that better integrates it with Hadoop. It allows expressing index construction as MapReduce jobs and provides hooks for manipulating the document collection, e.g., for controlling document sort order, for specifying the partition function, etc. This framework takes the output of the Pig scripts from the previous processing stage and constructs partitioned Lucene inverted indexes. The final output of the workflow is an operator-specified number of Lucene index partitions residing at a known location in HDFS, organized according to the date the index was created. For example:

```
/processed/usersearch/2011/08/01/indexes
/processed/usersearch/2011/08/02/indexes
...
```

Each of the subdirectories under the `indexes` directories would contain the individual partition indexes. For example:

```
2011/08/01/indexes/partition0
2011/08/01/indexes/partition1
2011/08/01/indexes/partition2
...
```

This entire indexing pipeline is coordinated by our internal workflow manager. The workflow manager tracks dependencies and handles sequencing of Pig scripts and Hadoop jobs of not only user search, but dozens of other services as well. It handles job scheduling, checks to make sure that preconditions have been met before a job is submitted (i.e., output from a previous Pig script is present or data import from an external source has completed), records the state of execution (success or errors, length of job, etc.), and notifies the operator in cases of failure.

Currently, our index construction pipeline is set to run on a daily basis. We are able to accommodate more frequent refreshes, but the upper limit for this design is on the order of an hour. In the case of our application, we feel that this latency is acceptable, since underlying relevance signals do not generally have a high velocity of change.

3.2 Building Blocks

The basic building block of our distributed search architecture is an individual index server, which uses the open-source Lucene search engine to serve a partition index. To achieve low query latency, the entire index is held in memory. Each index server exposes an interface that accepts queries, performs relevance ranking, and returns results. The interface is defined using Thrift, an open-source language-independent serialization and RPC mechanism.

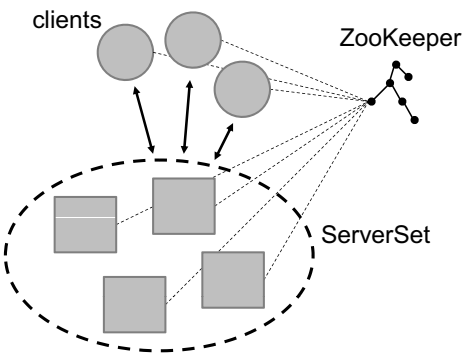

Figure 2: Illustration of the ServerSet abstraction, which provides a load-balanced, fault-tolerant replicated service. Replicas register at a known znode in ZooKeeper. Clients query the same znode to obtain hosts providing the service. Load balancing is provided by the client API.

Higher-level abstractions in our design rely heavily on ZooKeeper, a robust, wait-free coordination system. It provides to clients the abstraction of a set of data nodes (called znodes), organized in a hierarchical namespace, much like a filesystem. Each znode is identified by a unique path and can hold a small amount of metadata. Clients manipulate znodes through the ZooKeeper API, which supports basic operations such as creating, deleting, checking the existence of, listing the children of, and setting a watch on znodes. The watch concept provides a "push notification" mechanism that triggers client callbacks. There are two types of znodes: regular (persistent) znodes and ephemeral znodes, which only exist for the duration of a client session. Zookeeper is implemented as a distributed service that achieves fault tolerance using a variant of Paxos [11].

Our next layer of abstraction is a ZooKeeper-backed software load-balanced service called a ServerSet (illustrated in Figure 2), which provides real-time membership status for a set of servers.[2] The ServerSet abstraction provides clients the view of a fault-tolerant, replicated service: this is how we handle replication of an individual index partition. ServerSet membership is implemented in ZooKeeper by a znode at a known path (essentially, a rendezvous point). An individual host joins the ServerSet by creating a child ephemeral znode under the znode representing the ServerSet. The znode contains metadata about the host, such as what port it is serving on, what its serving status is,[3] etc. For example, the following znodes in ZooKeeper represent a ServerSet `foo` that presently has three members:

```
/service/foo/member_0000
/service/foo/member_0001
/service/foo/member_0002
```

Note that an ephemeral znode is present only when the session is maintained; the znode disappears when the index server is no longer available (deliberate shutdown, unhandled exceptions, hardware failures, or a network partition between ZooKeeper and the server).

A client wishing to use the service provided by the ServerSet queries ZooKeeper for its membership and caches the

[2] Java source available at http://github.com/twitter/commons

[3] For example: ALIVE, STARTING, STOPPING, WARNING, DEAD.

results; subsequent membership changes are processed via callbacks (i.e., watches on the znode). Load balancing is provided on the client-side via a "least-loaded strategy". A priority list of members in the ServerSet is maintained based on number of exceptions—the most lightly-loaded host is selected to handle the next request. If a server is unavailable, the client marks it "dead" for a user-adjustable period of time. In the case of a transient error, the client retries the host later. In the case of a permanent failure, the host's ephemeral znode disappears, triggering the callback for ServerSet membership changes, causing the client to remove the dead host from consideration. Similarly, if a new member joins the ServerSet, the host becomes available to the client (also notified via the ZooKeeper watch). Thus, the ServerSet abstraction provides robust, load-balanced access to a replicated service both in the presence of failure *and* the insertion of additional resources.

The task of a search broker is to distribute queries to all partitions and merge their results. Brokers expose exactly the same Thrift interface as each of the index servers. Each partition is backed by a ServerSet (handling the replication), so the broker simply needs to ensure that each ServerSet is queried. Discovery of the ServerSets corresponding to each partition happens automatically through ZooKeeper, discussed in the next section. Finally, the brokers themselves are organized as a ServerSet, providing a replicated service for the frontend (e.g., twitter.com), which handles the interface, result rendering, etc.

3.3 Configuration Management

The only piece of configuration information that needs to be supplied to every component in our distributed search architecture is the location of ZooKeeper (i.e., hostname and port). All other aspects of configuration management and service discovery are handled automatically.

In ZooKeeper, data associated with user search is maintained at a fixed, known, permanent location, statically specified in the codebase.[4] The relevant znodes in ZooKeeper are shown below (slightly simplified):

```
/twitter/usersearch/bootstrap
/twitter/usersearch/versions/index20110801
```

ZooKeeper is a production service used by many applications across Twitter, so all data related to user search are confined to its portion of the namespace. The "bootstrap" znode acts as a pointer to the current index layout being served, via its metadata. The index layout (in this case, `index20110801`) is associated with its own znode; children of a `partition_list` znode represent the partitions:

```
index20110801/lock
index20110801/partition_list/0
index20110801/partition_list/1
index20110801/partition_list/2
```

Here, the current layout has three partitions. Each of these partition znodes (`0`, `1`, `2`) represents a ServerSet. Metadata attached to each provide the location of the index data (i.e, URI specifying path on HDFS).

At service startup, each index server contacts ZooKeeper for the current layout, and then examines that particular znode. The index server will attempt to "claim" the least-replicated partition and join the corresponding ServerSet (described in Section 3.2). This presents a distributed race condition: servers starting up at the same time see the same partition as "least replicated", and will simultaneously attempt to join that ServerSet. To solve this, we need to establish a total order in which index servers claim partitions.

Fortunately, ZooKeeper provides a mechanism for exactly this. We implemented a distributed lock using the sequential mode for ephemeral znodes. This works as follows: if multiple clients attempt to create ephemeral children of the same parent znode (denoted `lock`), ZooKeeper will assign them names in sequential order. The client which created the lowest numbered znode "wins" the race, and is allowed to join the least replicated ServerSet. At this point, the index server begins initialization: first, it joins the ServerSet it selected, setting the znode metadata to the state STARTING (to denote that it is not ready to serve traffic yet), then it reads the metadata of the partition znode to find the URI of the associated index. A check is performed to see if the index already exists on local disk (which may be the case after recovery from failure). If not, the index is copied from HDFS.[5] After this, the index server loads the index into memory, switches its ServerSet state to ALIVE, releases the distributed lock by deleting its ephemeral sequential locking child, and begins serving requests. Admittedly, this is a slow startup sequence for multiple index servers, but we discuss this point further in Section 5.

One salient feature of our design is that the index server startup sequence makes no distinction between "normal" and unanticipated conditions (i.e., failures). We rely on ZooKeeper to coordinate the assignment of partitions to index servers in all cases. It guarantees that all state update requests are atomic, serializable, and respect precedence, which means that each partition will be evenly replicated—even when multiple index servers make simultaneous attempts to "claim" partitions. Consider the case of an ill-timed failure: multiple servers have started up at the same time, and race to serve the least-replicated partition. The one which wins the race, and holds the lock, then fails during index download, and the server crashes. Since the znodes it created are ephemeral, it leaves the ServerSet it joined and releases the lock, freeing up the remaining operational servers to continue as if no error occurred. Since each of the partition znodes correspond to a ServerSet, the mechanisms described in Section 3.2 ensure robustness both with respect to failures and addition of resources. After recovering from machine failures (which are relatively brief for the most part [8]), an index server will typically rejoin the same ServerSet (since it will be under-replicated) and continue serving requests—all without manual intervention.

Likewise, broker configuration is completely automatic. The brokers are clients to the partition ServerSets. Upon startup, the brokers also locate the partitions via ZooKeeper: by first consulting the bootstrap node, which redirects to the current layout. The brokers themselves form a ServerSet (i.e., registering at a known location) that is presented to the frontend clients.

[4] We argue that hardcoding this path is a reasonable decision since we do not expect it to change throughout the life of the service.

[5] There is nothing special about this being HDFS, as the fetching mechanism uses the schema of the index URI to determine whether the data should be fetched from HDFS, via scp, or even from Amazon's S3 distributed filesystem.

3.4 Operational Lifecycle

As previously mentioned, our architecture makes use of the design principle that eliminates the distinction between failure and other anticipated service disruptions. All processes are monitored by the open-source tool Monit[6] and restarted on failure. Whatever the scenario (service startup, recovery from failure, provision of new resources, etc.), the index server initialization algorithm is exactly the same: first, consult ZooKeeper to join the most appropriate ServerSet, then begin serving the correct partition. Because there is only one code path, it is always exercised, and thus we are more confident that bugs have been eliminated. In this way, simplicity breeds robustness. Other aspects of operations build on the same mechanism, described below.

Code deployments. Code deployment to production is as simple as pushing out new code and killing each of the index servers in turn. When an index server goes down, its corresponding ephemeral znode disappears and clients are notified. Upon restart, the index server—now running new code—automatically rejoins the same ServerSet (since the partition will have the lowest replication) and via the same notification process begins to share the request load. These actions are seamlessly coordinated via ZooKeeper.

Index rolls. Another important part of the user search operational lifecycle is the periodic index roll, the process whereby newly-created indexes by the Hadoop backend are pushed to production. Naturally, this process must occur without service disruption while production traffic continues. Critically, we make the assumption that our machines do not have sufficient memory to simultaneously hold two versions of the partition index. Relaxing this constraint essentially doubles the size of the index that can be served on a machine. However, this means that the most straightforward solution—loading the new index in memory alongside the old version, and then atomically switching the service over to the new index—is not possible.

Our implementation of index rolling occurs in two steps. The first step is to create a new layout by creating a new znode under `versions`, and children under that which have metadata pointing to the new HDFS index URIs.[7] The creation of the new layout is a very lightweight process, and is scripted to run periodically, polling HDFS for new indexes at the known location.

After the new layout has been created, the actual index rolling process is triggered. The currently live layout version (residing in the bootstrap node) is retrieved, set aside, and replaced by the newly created version. The layout roll is quite similar to the process of rolling restarts in code deployments: the process simply kills (using an RPC call via Thrift) an arbitrary replica of the most highly-replicated partition. When an index server restarts, it automatically switches over to serving the new layout (as specified by the bootstrap znode). This approach essentially yields a round-robin restart sequence across the partitions; it ensures that one replica of each partition switches over to the new layout, and then the second replica of each partition, etc. Note that this is a relatively conservative approach, and we discuss alternative designs in Section 5.

[6] http://mmonit.com/monit/

[7] This also means that the partitioning scheme can be changed as frequently as new indexes are created.

Search brokers issue requests only to the single layout they believe is "current", and so when index servers have rolled to the new layout, they are not yet serving traffic since no broker knows how to connect to them. Once enough index servers have rolled to the new layout to handle $\frac{1}{numBrokers}$ worth of the total cluster traffic, the index rolling process restarts a broker, which causes it to start sending all of its traffic to the new layout.[8] After that, more index servers are restarted, until enough of them have rolled to allow for another broker to switch, and so on.

A few safeguards ensure the robustness of the index rolling operation. The rolling process is idempotent: if interrupted for any reason (e.g., network partition, crash of the machine the process is running on, etc.), the process can be run again safely. As it turns out, this is a nice feature that simplifies operational management, as we discuss in Section 4.2. In addition, before a new layout is created, the existing layout is preserved. This supports a recovery mechanism in case any unanticipated errors arise (e.g., a corrupted index). If the index roll does not complete within a specifiable amount of time, an automatic rollback is triggered, reverting the system to its previous layout state.

4. PRODUCTION DEPLOYMENT

The distributed search architecture described here has been running in production and providing user search for Twitter since March 2011. We began development of the service in the summer of 2010. The architecture has worked as designed, and while there have been service disruptions due to RAM exhaustion[9] and some due to internal bugs,[10] we have not experienced a service disruption that can be attributed to distributed system design flaws. In this section, we share our experiences of having deployed and operated the service in production.

At launch time, our production setup was modest, consisting of 3 brokers and 12 index servers. The collection was divided into three partitions with four replicas of each partition. Traffic has steadily increased since launch, and we have been supplied with additional machines. As of August 2011, we are running 3 brokers and 21 index servers.

4.1 Performance

We note that the single biggest factor affecting the performance of the entire service is the speed of the core Lucene ranking algorithm within each partition index server. Since we currently employ a relatively simple ranking function, a modest hardware setup is sufficient to serve a surprisingly large query load.

Prior to launch, we conducted a series of performance stress tests. We built infrastructure that is able to take production query logs and replay them at various speeds, in real-time and in multiples of real-time. The production

[8] At this point, frontend clients see results from different indexes depending on which broker they connect to, although each request hits indexes only from the same layout. This means that during the index roll process, it is possible to obtain two different ranked lists of results for the same query. We are willing to tolerate these minor inconsistencies since differences between subsequent versions of the indexes are for the most part minor.

[9] Don't forget to raise the memory available to your JVM as the index grows!

[10] Rare broken queries causing elevated error counts, leading front-end safeguards to blacklist the user search service entirely.

setup at launch time could comfortably handle loads up to 15k queries per second (QPS). At around 20k QPS, the service begins to experience 95% latency above 250ms and an unacceptably high number of timeouts. Additional scaling experiments were conducted on Amazon's EC2 service, primarily to test the scale-out capabilities of the architecture: clusters of up to 100 index servers have been launched and verified to serve replayed production query loads without any difficulty.[11]

Despite the growth in traffic since launch, there remains substantial headroom in the service to handle much larger query loads. We are currently developing more sophisticated relevance ranking algorithms which will make fuller use of available computing capacity.

4.2 Operational Experiences

Since the launch of user search, we have had hardware upgrades and the provisioning of additional machines. The architecture adapted to these changes without disruption, exactly as designed.

Some time after launch, operations engineers added more memory to our production machines. This upgrade was performed with minimal coordination: the operations staff had no specific knowledge of the processes running on the machines. They simply brought down each machine in turn, performed the upgrade, and returned the machine to service. When a machine was brought down, clients accessing the ServerSet of which the machine was a member immediately blacklisted the host. Due to sustained query load on the service, this usually happens *before* the watch on the ephemeral znode triggers, notifying the clients that the machine has "officially" left the ServerSet. Load is then redistributed to the remaining replicas, and the service continues operating. After the machine is returned to service, it rejoins the same ServerSet (since it's the least-replicated one); the clients are notified of its availability.

Similarly, when additional machines were provisioned for user search, they joined the existing service without fanfare, as designed. Once the machines were properly configured and code deployed to them, they went through the same startup sequence as in all other operational scenarios previously described. Each machine in turn joined the least-replicated partition and began sharing the query load.

The configuration management and index rolling aspect of our architecture has generally worked as designed. Prior to formal launch, the service underwent a trial period in which it was serving live traffic, but results were simply discarded by the frontends. During this period, we experimented with 3 vs. 4 partitions. Index rolling between different layouts proceeded smoothly. At launch time, we settled on 3 partitions, which was increased to 5 a few months later to reduce index creation time on the Hadoop end; due to natural growth of the Twitter user base, partitions steadily increase in size. The index rolling process switched between the different partitioning schemes seamlessly. In fact, the only change required was a configuration parameter (number of partitions) in the indexer. The index rolling process, upon consulting the index location on HDFS, is able to infer the number of partitions based on the number of subdirectories at that location, each of which corresponds to a partition index (see Section 3.1).

In the lifecycle of the user search service, the index rolling process is perhaps the most intricate operation, since it actively involves configuration changes across all production machines and depends on external components (e.g., HDFS). Although index rolling generally proceeds smoothly, we have encountered a handful of failures since the service launched. While these incidents required operator intervention, none of these failures resulted in a service outage. A few memorable examples: we encountered a transient failure of a rack switch, and as a result, the index server was unable to contact HDFS, causing the index roll to fail partway through.[12] Another time, we accidentally started an indexing job manually, which began by deleting existing indexes. An index roll was proceeding at the same time, which meant that partition indexes disappeared midway through the process, once again causing it to fail. During index rolling, partition under-replication alerts are suppressed by a ZooKeeper lock, but when the rolling process died, the lock was released, which triggered alerts as expected. On-call engineers were then notified to handle the situation.

Recovery from these errors was very straightforward: after diagnosing the cause, the operator simply restarted the rolling process, manually specifying the target layout. Recall that, by design, index rolling is an idempotent process (see Section 3.4). Therefore, the error recovery method is no different from normal operations. Index roll failures are of course undesirable, but for the most part, not fatal (i.e., resulting in a service disruption). If the index rolling process fails early on, then only a few replicas have switched over to the new layout, and a broker is unlikely to have rolled over yet. In this case, only a few replicas would be "stranded" in the new layout, unable to serve traffic. This would be no different than a few machines crashing. If the index rolling process fails midway or toward the end, then it means that at least one broker has also rolled over to the new layout. Thus, traffic *is* being served and the newly-rolled index partition servers are sharing the live traffic load. In this case, depending on which broker the frontend contacts, results may be slightly different. However, as we discussed previously, this was determined to be an acceptable service state. In short, index rolling failures are causes for concern, but not panic. This gives the operator the luxury of having adequate time to diagnose the root problem without having to worry about service disruptions.

5. FUTURE WORK

There are a number of improvements for future work. A known weakness of our architecture is its dependence on ZooKeeper. The service itself is distributed and replicated to ensure robustness, so the most common failure scenario is a network partition, i.e., failure of network links. Our system is resilient with respect to transient network errors since clients cache the location of ServerSet members, but a significant network outage will make service discovery impossible. Index servers cannot select a ServerSet to join without contacting ZooKeeper.

[11] Thus, although it may appear that we over-engineered the architecture for the launch hardware configuration (and that a simple static assignment strategy would have worked reasonably), the EC2 experiments verified that our design does work at a scale where simple static assignment would be unmanageable. Our architecture provides much room for growth, which is desirable given Twitter's overall growth trajectory.

[12] Who says network partitions don't happen?

In general, we see several relatively simple modifications that would increase the resiliency of our architecture to network failures. Currently, replica assignment is ignorant of the network topology, so it may be the case that all replicas of the same partition reside on the same physical rack in the datacenter—in which case, the network link becomes the single point of failure. Currently, each index sever joins the least-replicated partition, breaking ties arbitrarily. This can be improved to take into account the network topology: for example, ties could be broken by selecting a partition to place replicas on different racks, thus ensuring the survivability of the service to partial network outages. One can imagine a rich set of alternative selection schemes that balances locality (i.e., taking advantage of higher intra-rack bandwidth and lower latency) and robustness (i.e., distributing replicas across physical racks).

A known weakness of our design is the regulated speed at which our service changes state. Starting up the entire system from scratch is slow because the index servers "claim" replicas sequentially; the next index server "in line" is not allowed to proceed until the previous has successfully initialized. Similarly, during index rolling, only one index server rolls at a time, and the process pauses after each to confirm success. However, this limitation is more the result of engineering conservatism and desired to throttle bandwidth consumption (from copying indexes) than a design limitation. During startup, it would be possible to roll indexes more aggressively, for example, one replica of each partition simultaneously.[13] We have not explored tradeoffs between time, robustness, and bandwidth consumption in alternative designs, but hope to do so in future work. Thus far, we have found existing mechanisms to be adequate—for example, we have never experienced a need to restart the entire service from scratch (thus slow startup has not been an issue).

Presently, we assume a single datacenter environment. However, as the service expands, for both robustness and latency reasons, it will become necessary to serve from multiple geographically-distributed datacenters. Accommodating this will not significantly alter the design presented here: instead, our existing architecture will become another building block and will present a datacenter-level service abstraction. Its client would most likely be a query-routing layer that dispatches queries to the most appropriate datacenter.

6. CONCLUSION

This paper describes the architecture that underlies Twitter user search, which is composed entirely of open-source software components. Building on well-known design principles for distributed search in the academic literature, our goal is to fill in missing gaps about practical, operational aspects. Our architecture represents a single point in the design space of partitioned, replicated search services that encodes a particular set of tradeoffs and assumptions. The design is simple, robust, and provides a reference point for future work exploring alternative architectures.

7. ACKNOWLEDGMENTS

We'd like to thank anonymous reviewers and Jeff Dalton for helpful comments on earlier drafts of this paper. The third author is an Associate Professor at the University of Maryland, and participated in this work while on sabbatical at Twitter. He is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob. The second author thanks Helene for her continuing support and encouragement, without which none of his current career would be possible.

[13]Or more, in the case of a very large cluster. The only constraint is that the number of replicas remaining while some are out of service (while rolling) can handle current traffic on their own.

8. REFERENCES

[1] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. *ICDE*, 2007.

[2] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. *SIGIR*, 2007.

[3] L. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[4] L. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[5] S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[7] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, 1992.

[8] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. *OSDI*, 2010.

[9] J. Hamilton. On designing and deploying Internet-scale services. *LISA*, 2007.

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. *USENIX*, 2010.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[12] C. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.

[13] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. *SIGIR*, 2006.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. *SIGMOD*, 2008.

[15] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. ResIn: A combination of results caching and index pruning for high-performance web search engines. *SIGIR*, 2008.