1

5 NoSQL databases and Hadoop

Say you are planning or maintaining an application, and it needs to scale or to be speeded up. That is of course a very common problem. In this book we will concentrate on the NoSQL approaches to the problem. Of course, by now the boundaries are blurred; for example, Cassandra has implemented CQL, or Cassandra Query Language, and code-named it YesSQL. Still, before you plunge and spend your effort on a specific implementation, you need to be sure that you are going in the right direction and that you have selected the right datastore to use. Besides, the principles of many NoSQL datastores are the same, even if implementation differ, so it pays to study them.

In this chapter therefore, we will look at various NoSQL approaches, give an example of the use of each, and thus feel more comfortable with our subsequent choice. In doing this, I stand somewhat at odds with some of my colleagues . You see, many of them have already gone through the pains of the selection, prototyping, and are running their sites and companies using specific architectures. Therefore, they can tell you something like "just concentrate on HBase, since this is a tried-out and true solution." That means it worked for them and they are happy with it. On the other hand, you may not have done so, and are at the beginning of this road. By the way, even they internally agonize and worry if they had made the right choice. Therefore, we will spend a little time and review the alternatives. Eventually though, we will spend most of our time on HBase, because it has a good record of successful implementations. Still, we will also analyze the opinions of those who chose Cassandra, CouchDB, MongoDB, etc., because wisdom comes from the multiplicity of advice.

5.1 Choosing your NoSQL database

Say you have a very common problem, that is, your website is slow. Not only that, but it is getting slower - as more users are coming on. You may have a SQL solution already in place, and it may be prohibitive to re-write it completely with NoSQL. In that case, a memory caching solution may be in order, and we will look at it first in the coming sections. It may also be that you are free to choose your NoSQL technology, and then your choices maybe between document-oriented (like MongoDB and CouchDB) and more general key-value databases, like Cassandra and HBase. Then your task will be to select the right one and to build your data model to fit the structure of your solutions. We will show the situations where this is applicable and give examples of using these databases.

We will also use the considerable experience accumulated and shared by the industry leaders: Google, Yahoo, Facebook, Amazon, etc. There is a good reason for that: with NoSQL and with Big Data in general we are on the bleeding edge of technology. There are few uncontested standard approaches, and new competing offerings come up every day. Thus, it becomes ever more important to use the advice of that who have "been there, done that," and we will not ignore it.

5.1.1 Distributed hash tables for in-memory databases

So, your site is built on MySQL and you have no desire to re-write it, but it getting slower and slower. I had such example, http://www.quizrevolution.com/, so let's look at it. When I became an architect for the site, it was slow. Besides, we had problems ramping up and down: meaning, when we deployed a new database machine, we needed hours to dump the SQL and then more hours to restore it on another machine. With hundreds of millions of rows, the dump/restore process can be quite lengthy. In seeing what we did with the site, we will also have an example of solving similar problems for other situations.

The first thing we did was to transfer the MySQL database to Amazon Relational Database Service (Amazon RDS). This is essentially the same MySQL, but Amazon RDS "automates common administrative tasks, offers feature rich functionality that enhances database availability and scalability, significantly reducing the complexity of managing and the cost of owning database assets." What does it mean in plain terms? Amazon store your database and runs it for you. You are still running on one machine, so that is vertical scalability, but it is the ultimate vertical scalability: in a matter of minutes you can take your database and switch it from running on the smaller Amazon machine to the largest one, with 7

Gigs of RAM and 20 EC Compute Units. What took hours before takes minutes on this monster machine. Then you can switch back to the smaller instance and save money.

True, this is not NoSQL and not open-source; it is also not free, but hey, sometimes the programmer has to do what the programmer has to do, and compromise between build-it-yourself and buy it. With this out of the way, let's turn to speeding things up with an in-memory database, namely, memchached.

TECHNIQUE 1: Offloading your database with a distribute memory cache

PROBLEM

Your SQL-based website is running slow. You have exhausted your options, and your database is still overloaded. You want to somehow offload it without a big re-design.

SOLUTION

You can use memcached as an intermediate buffer between your database and your web pages. Memcached is a general-purpose distributed memory caching system. It was originally developed by Brad Fitzpatrick for his website LiveJournal in 2003. It is currently in use by such sites as LiveJournal, Wikipedia, Flickr, Bebo, Twitter, Typepad, Youtube, WordPress, and Craigslist, and is thus proven in practice and worth looking at. How does it work?

You can think of memcached as one giant hash table, which is in addition distributed between different servers. You add servers to your memcached at will, and the server then caches your objects. Here is how you start it (doing this in PHP, since it is a common language for web development, and since our production system did runin PHP

We will differentiate between development, testing (qa) and production environing. This is a good programming practice to think of such thing beforehand. We will first prepare an object that we can re-use anytime we need an access to memcached servers.

Listing 5.1 Initializing your memcached servers

```
class MysiteMemcached extends Memcache {
  public function MysiteMemcached( $environment = null) {
  if( empty($environment)) {
    $environment = ENVIRONMENT_CODE;
  }

  switch( $environment ) {
```

1 Prepare a global object for code re-use

2 Differentiate between

```
    case ENV_DEV:                                                    environments
      $this->addServer('localhost', 11211);
      break;
    case ENV_QA:
      $this->addServer('qa1.mysite.com', 11211);
      $this->addServer('qa2.mysite.com', 11211);
      break;
    case ENV_PROD:
                                                                     3 On EC2, you can
      $this->addServer('prod1.mysite.com', 11211);                     use internal IP here
      $this->addServer('prod2.mysite.com', 11211);
      $this->addServer('prod3.mysite.com', 11211);
      break;
    default:
      throw new Exception("Unknown environment $environment");
    }
  }
}
```

Now that we have this memcached class, any time we need access to objects whose size lends itself to being stored in the cache, we can ask the memcached class if what we are looking for is already cached. If it is, it will return it to us, and if not, it will get it from the database. This looks as following

Listing 5.2 Using memchached

```
class DisplayManager {
  private  $sql = null;
  private $memcache = null;
  public function PlayManager() {
                                                                     1 Prepare access
    $this->sql = new App_SQL;                                          objects to be used
    $this->memcache = new MysiteMemcache();                            by the class
  }
  public function getAllItemsByUserID($id) {
                                                                     2 If the object we
    $cached = $this->memcache->get(                                    look for is already
      $id . '-' . 'getAllItemsByUserID');                              in memcached -
    if( !empty($cached )) {                                            return it
      return $cached;
    }
                                                                     3 If not, get it from
    $uncached =                                                        the database
      $this->getItemsBySQL(
      $this->sql->query("select * from item
        where userID = %i order by orderID", array($id)));
                                                                     4 and put it in
      $this->memcache->set($id . '-' . 'getAllItemsByUserID',          memcached, then
        $uncached, 0, ITEMDATA_EXPIRY_SECS);                           return it
      return $uncached;
    }
```

```
  }
}
```

DISCUSSION

Memcached is not a panacea for every situation. It works best when you are caching objects of medium size. The largest size you can save in memcaches is 1 megabyte. If you are trying to save something larger than that, you are probably doing something wrong and need to re-think your design. If your caches take too long to fill, then again you might be better off going back to a database, but perhaps to a NoSQL database after all. Once you have your memcached working, you need to evaluate the effect of one of the servers going off-line, and you need to monitor your solution for performance.

Memcached is a simple, very easy to install, and scalable key-value datastore, which runs in memory. That is a good introduction to key-values stores in general, and now we are ready to tackle more advanced uses of the same idea. First, however, we will review the general pros and cons of NoSQL.

5.1.2 The pros and cons of NoSQL movement, hybrid approaches

We already touched on the pros and cons of NoSQL datastores. Let us now briefly review them, before we make the final plunge into a NoSQL solution. Use SQL if

- Your data volume is low, in the millions, not billions of rows
- Your data model is very complex, with many interdependent relationships
- Your data columns are relatively few, up to a hundred, they are defined and don't change often
- Your database gives you a decent response time, and scales with too much expense

On the other hand, here when you would use a NoSQL datastore:

- Your data volume is high, from tens of millions to billions of rows
- Your data model is not very complex
- Your data columns are numerous from hundreds to millions, and they need to be defined by the application dynamically
- You need fast response and failover from a datastore that spans tens to hundreds to thousands of servers, and you don't want to spend a fortune on it

5.1.3 Document-based NoSQL databases

Let's recall the eDiscovery or compliance area of applications that we've dealt with in the previous chapter. In eDiscovery, you need to look at every document found on the person's (the people in these applications are called custodians), and extract it all: text, metadata (like who the custodian actually is, when was the file created, printed, etc.) and potentially even forensics information, from simple clues, such as attempts to rename the file extention to avoid easy analysis, to hiding information is sector slack space or using steganography. Oof, it took a lot of breath even to enumarate them! But never mind, we will extract all the metadata and store it, and it may be a few hundred fields, since different data sources and different file formats have different metadata. Now how do we store it? It so happens that the document-based NoSQL databases make the best natural fit for these kinds of applications. What are the their advantages? Take, for example, CouchDB, it

- Document-oriented, which provides a natural fit
- Allows any amount of metadata, which can differ from row to row
- Has a very large limit (4 Gigs) on document size. You can hardly find user documents (think of MS Office documents) larger than that, because they need to fit in memory

In short, this seems to be a match made in heaven. This is not very surprising, if one recalls that CouchDB, Damien Katz, was initially the designer for Lotus Notes, so that sometimes CouchDB is labeled as "Notes done right." So how exactly do we take the documents that we process in eDiscovery and put them into CouchDB? Here is how,

Listing 5.3 Storing documents in CouchDB

```
// create a database object pointing to the database "mycouchdb" on the local host
Database db = new Database("localhost", "mycouchdb");
// create a hash map document with two fields
Map<String,String> doc = new HashMap<String, String>();
doc.put("author", authorValue);
doc.put("custodian", cutodianValue);
// create the document in couchdb
db.createDocument(doc);
```

A little explanation is in order. CouchDB can work with many programming languages, but given the choice, we used the Java client, Jcouchdb. As you can see, CouchDB allows you complete flexibility in regards to schema, in fact, here we

have not predefined schema at all, but instead set up the column names as we go, according to the needs of the application. This is common with NoSQL databases. The specifics of CouchDB require us to provide the definition of the document that we will be using, similar to the table name and properties in the SQL world, and this is done with the following piece of JSON code

```
package org.jcouchdb.document;
import org.svenson.JSONProperty;
public interface Document
{
    @JSONProperty( value = "_id", ignoreIfNull = true)
    String getId();
    void setId(String id);

    @JSONProperty( value = "_rev", ignoreIfNull = true)
    String getRevision();

    void setRevision(String revision);
}
```

As you can see, we still need the document ID. CouchDB does not generate one for us, but requires that we provide a unique ID instead. Again, this is common with NoSQL databases, and there is a good reason for that. Had we asked it to provide us with a unique ID, we would need to make the operation of getting it and then using it atomic, which would right away make a bottleneck. But that's no problem, because we already thought of that. We used one reducer, and it provided us with a global counter, resulting in a unique and moreover sequential ID. This approach may be an anti-pattern in some situations, and we already discussed the possible alternatives to it.

5.1.4 Generic key-value databases

5.1.5 Cassandra and Hadoop

5.2 Thinking about your data model in new terms

5.2.1 Data model expressed in Cassandra

5.2.2 NoSQL design patterns

TODO - this section needs work

There are a few ways that people commonly use Cassandra that might be described as design patterns. I’ve given names to these common patterns: Materialized View, Valueless Column, and Aggregate Key.

Materialized View

It is common to create a secondary index that represents additional queries. Because you don't have a SQL WHERE clause, you can recreate this effect by writing your data to a second column family that is created specifically to represent that query.

For example, if you have a User column family and you want to find users in a particular city, you might create a second column family called UserCity that stores user data with the city as keys (instead of the username) and that has columns named for the users who live in that city. This is a denormalization technique that will speed queries and is an example of specifically designing your data around your queries (and not the other way around). This usage is common in the Cassandra world. When you want to query for users in a city, you just query the UserCity column family, instead of querying the User column family and doing a bunch of pruning work on the client across a potentially large data set.

Note that in this context, "materialized" means storing a full copy of the original data so that everything you need to answer a query is right there, without forcing you to look up the original data. If you are performing a second query because you're only storing column names that you use, like foreign keys in the second column family, that's a secondary index.

Valueless Column

Let's build on our User/UserCity example. Because we're storing the reference data in the User column family, two things arise: one, you need to have unique and thoughtful keys that can enforce referential integrity; and two, the columns in the UserCity column family don't necessarily need values. If you have a row key of Boise, then the column names can be the names of the users in that city. Because your reference data is in the User column family, the columns don't really have any meaningful value; you're just using it as a prefabricated list, but you'll likely want to use values in that list to get additional data from the reference column family.

Aggregate Key

When you use the Valueless Column pattern, you may also need to employ the Aggregate Key pattern. This pattern fuses together two scalar values with a separator to create an aggregate. To extend our example further, city names typically aren't unique; many states in the US have a city called Springfield, and there's a Paris, Texas, and a Paris, Tennessee. So what will work better here is to fuse together the state name and the city name to create an Aggregate Key to use in

our Materialized View. This key would look something like: TX:Paris or TN:Paris. By convention, many Cassandra users employ the colon as the separator, but it could be a pipe character or any other character that is not otherwise meaningful in your keys.

5.2.3 Secondary indexes in NoSQL databases

TODO - this section needs re-working

One of the common issues of dealing with the Apache Cassandra database is how to do secondary indexes of columns within a row. This post will discuss one technique, far from the only one, for how to manage this. One thing that experienced Cassandra users will hopefully find interesting is that SuperColumns will not be used at all to accomplish this in order to avoid the complexity and limitations they introduce. Also, it should also be pointed out that Cassandra will have native secondary index support in the upcoming 0.7 release (see CASSANDRA-749), which will make this all much simpler, but the idea is still valid for how to think about about this sort of thing, and will still be applicable in some situations. Once that version gets closer to release, I'll do a follow up post looking at it. So, to start, let's assume a scenario where we have a container (ex. a group) of items (ex. users in the group), each of which has an arbitrary set of properties, which are searchable by value in the context of the container. Items might also be members of other containers, but we won't explicitly deal with that in this scenario. One way to model this in Cassandra is to have two Column Families (Cassandra-speak for table-like entities) which we'll just call CF's for short. The first CF would be for the item's properties, which we'll call Item_Properties. This is the simplest form of using Cassandra's data model. Rows in Item_Properties will be looked up by a key, which in this example will be UUID's. The columns inside the CF are the property names and the values stored in the columns are the property values. CF: Item_Properties Key: item_id Compare with: BytesType Name Value property_name property_value ... ... The next CF is for the containers that will hold sets of items, and is appropriately named Container_Items. Each column in a Container_Items row is the key of an row in Item_Properties. This is one of the first things that trips people up in using Cassandra. While you can use Column Families as simple tables and use each row the same way you'd use a row as a record in a normal database, each row can be used as a simple table in and of itself, particularly for modeling join tables. In the Container_Items CF, each column uses the key of an row in Item_Properties as it's name and the timestamp of when it was added to the collection as the column value. This row can grow quite

large as more and more items are added to it, but since each column is about 42 bytes for the UUID and the timestamp, this is going to hit a limit of 40M+ items in a group under pre-0.7 limitations. For a container of users in a group, this might be a reasonable limitation, but you'd also use this same mechanism for collecting things like status updates (i.e. "tweets") which could much more conceivably exceed that. The 0.7 release of Cassandra will remove this limitation. UPDATE: In version 0.7, you can now have 2B columns in a row CF: Container_Items Key: container_id Compare with: TimeUUIDType Name Value item_id insertion_timestamp ... ... So far, this is fairly basic Cassandra data modeling. Where things get complicated is when one wants to start to retrieve items from the group by querying against specific property values. For this, you'll need to manage your own indexes as this is outside of the purposefully minimalist design of Cassandra. To do this, we'll create two more column families. This first CF holds the actual index, and we use the container ID with the property name we want to index as the key to look it up. It would look something like this: CF: Container_Items_Property_Index Key: container_id + property_name Compare with: compositecomparer.CompositeType Name Value composite(property_value, item_id, entry_timestamp) item_id ... ... Where this indexing technique departs a bit from techniques described elsewhere is how each column in the index is constructed. Cassandra provides a set of column types that it can use to sort the columns in a row. You can only specify one and it's defined at column family creation time. Cassandra also lets you define your own column types and in what we've done here is create a CompositeType (see http://github.com/edanuff/CassandraCompositeType). This will let us define columns that Cassandra will sort for us that combine several component values into a single composite type. This lets us create unique column entries for potentially (and probably) non-unique property values by adding additional values to the column as discriminators. So, now, in order to query a container for all the items where a property equals a certain value, we'd take the key of the container and append the property name to it, and do a slice query against the value. All columns where the value was the first component in the composite type would be returned, and the value of each column would be the key of the appropriate row in Item_Properties. Not that it doesn't actually have to be precise equality, range queries will work with this technique too. The final issue to deal with is what happens when the value of a property is changed and the index has to be updated. The simple answer is that you'd introduce a column for the new value in the

Container_Items_Property_Index column family and remove the column for the old value. However, for a number of reasons related to Cassandra's model of eventual consistency and lack of transactions, simply reading the previous value from Item_Properties before updating it and then removing the index entry for that value from Container_Items_Property_Index will not reliably work. To do this we maintain a list of previous values for the specific property of a given item and use that to remove these values from the index before adding the new value to the index. These are stored in the following CF: CF: Container_Item_Property_Index_Entries Key: container_id + item_id + property_name Compare with: LongType Name Value entry_timestamp property_value ... ... We remove these columns from the CF after retrieving them, so this row never gets too large, in most cases probably never grows much larger than one or two columns, more if frequent changes are being made. By the way, it's a really good idea to make sure you understand why this CF is necessary because you can use variations of it to solve a lot of problems with "eventual consistency" datastores.

5.2.4 Client or Thrift access?

Factory to access either Hector or Thrift

```
object CassandraClientFactory {
  def getClient: CassandraClient = {
    var whichClient = Constants.cassandraClient
    // temp hack for reassignment
    whichClient = "hector"
    //whichClient = "thrift"
    if (whichClient == "hector") {
      new CassandraClientHector
    }
    else {
      new CassandraClientThrift
    }
  }
}
```

How do we then use our refactored interface? Here is getting columns from family in Thrift

```
  def fetchColumnsFromFamily(columnFamily: String, key: String, rangeCount: Int = -1
    // setup the query
    val slicePredicate = new SlicePredicate
    val sliceRange = new SliceRange
```

```
    sliceRange.setStart(new Array[Byte](0))
    sliceRange.setFinish(new Array[Byte](0))
    slicePredicate.setSlice_range(sliceRange)

    val parent = new ColumnParent(columnFamily)
    //val parent = new ColumnParent(Constants.eventSequenceColumnFamily)

    val keyRange = new KeyRange
    keyRange.setStart_key(key)
    keyRange.setEnd_key(key)
    if (-1 < rangeCount) {
      keyRange.count = rangeCount
    }
    val connector: Connect = new Connect
    val client: Cassandra.Client = connector.connect
    // execute the query
    val rawEvents = client.get_range_slices(parent, slicePredicate, keyRange, Consta
    val sliceList = buildListFromKeySlice(rawEvents.get(0))
    connector.close
    sliceList
  }
```

And here is the same in Hector

```
  def fetchColumnsFromFamily(columnFamily: String, key: String, rangeCount: Int = -1
    debug("fetchColumnsFromFamily")
    assert(columnFamily != Constants.eventSequenceColumnFamily)
    val cluster: Cluster = HFactory.getOrCreateCluster("Test Cluster",
      Constants.host + ":" + Constants.port)
    val keyspaceOperator: Keyspace = HFactory.createKeyspace("GhxTnt", cluster)
    keyspaceOperator.setConsistencyLevelPolicy(GhxConsistencyLevelPolicy)
    val stringSerializer: StringSerializer = StringSerializer.get
    val reversed = false
    val count = if (rangeCount > -1) rangeCount else -1
    val query: RangeSlicesQuery[String, String, String] =
      HFactory.createRangeSlicesQuery(keyspaceOperator, stringSerializer, stringSeri
    query.setKeys(key, key)
    query.setRange("", "", reversed, rangeCount)
    query.setColumnFamily(columnFamily)
    val result = query.execute
    val orderedRows: OrderedRows[String, String, String] = result.get
    val sliceList = ListBuffer[Map[String, String]]()
    for (row <- orderedRows) {
      val columnSlice = row.getColumnSlice
      val columns = columnSlice.getColumns
      val mapBuilder = new MapBuilder[String, String, HashMap[String, String]](new H
      for (column <- columns) {
        // each map store just one column
        mapBuilder += column.getName -> column.getValue
      }
      sliceList += mapBuilder.result.toMap
    }
    sliceList.result
  }
```

5.2.5 Inbox indexing with Cassandra

TODO - this section needs reworking

Cassandra is a distributed storage system for managing structured data that is designed to scale to a very large size across many commodity servers, with no single point of failure. Reliability at massive scale is a very big challenge. Outages in the service can have significant negative impact. Hence Cassandra aims to run on top of an infrastructure of hundreds of nodes (possibly spread across different datacenters). At this scale, small and large components fail continuously; the way Cassandra manages the persistent state in the face of these failures drives the reliability and scalability of the software systems relying on this service. Cassandra has achieved several goals – scalability, high performance, high availability and applicability. In many ways Cassandra resembles a database and shares many design and implementation strategies with databases. Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. The rest of the material talks about the data model and the distributed properties, provided by the system. Data Model Every row is identified by a unique key. The key is a string and there is no limit on its size. An instance of Cassandra has one table which is made up of one or more column families as defined by the user. The number of column families and the name of each of the above must be fixed at the time the cluster is started. There is no limitation the number of column families but it is expected that there would be a few of these. Each column family can contain one of two structures: supercolumns or columns. Both of these are dynamically created and there is no limit on the number of these that can be stored in a column family. Columns are constructs that have a name, a value and a user-defined timestamp associated with them. The number of columns that can be contained in a column family is very large. Columns could be of variable number per key. For instance key K1 could have 1024 columns/super columns while key K2 could have 64 columns/super columns. “Supercolumns” are a construct that have a name, and an infinite number of columns assosciated with them. The number of “Supercolumns” associated with any column family could be infinite and of a variable number per key. They exhibit the same characteristics as columns. Distribution, Replication and Fault Tolerance Data is distributed across the nodes in the cluster using Consistent Hashing based and on an Order Preserving Hash function. We use an Order Preserving Hash so that we could perform range scans over the data for analysis at some later point.

Cluster membership is maintained via Gossip style membership algorithm. Failures of nodes within the cluster are monitored using an Accrual Style Failure Detector. High availability is achieved using replication and we actively replicate data across data centers. Since eventual consistency is the mantra of the system reads execute on the closest replica and data is repaired in the background for increased read throughput. System exhibits incremental scalability properties which can be achieved as easily as dropping nodes and having them automatically bootstrapped with data. First deployment of Cassandra system within Facebook was for the Inbox search system. The system currently stores TB’s of indexes across a cluster of 600+ cores and 120+ TB of disk space. Performance of the system has been well within our SLA requirements and more applications are in the pipeline to use the Cassandra system as their storage engine.

5.2.6 Faster communications with HBase

TODO - this section needs reworking

Facebook's New Real-Time Messaging System: HBase To Store 135+ Billion Messages A Month TUESDAY, NOVEMBER 16, 2010 AT 7:52AM You may have read somewhere that Facebook has introduced a new Social Inbox integrating email, IM, SMS, text messages, on-site Facebook messages. All-in-all they need to store over 135 billion messages a month. Where do they store all that stuff? Facebook's Kannan Muthukkaruppan gives the surprise answer in The Underlying Technology of Messages: HBase. HBase beat out MySQL, Cassandra, and a few others. Why a surprise? Facebook created Cassandra and it was purpose built for an inbox type application, but they found Cassandra's eventual consistency model wasn't a good match for their new real-time Messages product. Facebook also has an extensive MySQL infrastructure, but they found performance suffered as data set and indexes grew larger. And they could have built their own, but they chose HBase. HBase is a scaleout table store supporting very high rates of row-level updates over massive amounts of data. Exactly what is needed for a Messaging system. HBase is also a column based key-value store built on the BigTable model. It's good at fetching rows by key or scanning ranges of rows and filtering. Also what is needed for a Messaging system. Complex queries are not supported however. Queries are generally given over to an analytics tool like Hive, which Facebook created to make sense of their multi-petabyte data warehouse, and Hive is based on Hadoop's file system, HDFS, which is also used by HBase. Facebook chose HBase because they monitored their usage and figured out what the really needed. What they needed was a system that could handle two types of data

patterns: A short set of temporal data that tends to be volatile An ever-growing set of data that rarely gets accessed Makes sense. You read what's current in your inbox once and then rarely if ever take a look at it again. These are so different one might expect two different systems to be used, but apparently HBase works well enough for both. How they handle generic search functionality isn't clear as that's not a strength of HBase, though it does integrate with various search systems. Some key aspects of their system: HBase: Has a simpler consistency model than Cassandra. Very good scalability and performance for their data patterns. Most feature rich for their requirements: auto load balancing and failover, compression support, multiple shards per server, etc. HDFS, the filesystem used by HBase, supports replication, end-to-end checksums, and automatic rebalancing. Facebook's operational teams have a lot of experience using HDFS because Facebook is a big user of Hadoop and Hadoop uses HDFS as its distributed file system. Haystack is used to store attachments. A custom application server was written from scratch in order to service the massive inflows of messages from many different sources. A user discovery service was written on top of ZooKeeper. Infrastructure services are accessed for: email account verification, friend relationships, privacy decisions, and delivery decisions (should a message be sent over chat or SMS?). Keeping with their small teams doing amazing things approach, 20 new infrastructures services are being released by 15 engineers in one year. Facebook is not going to standardize on a single database platform, they will use separate platforms for separate tasks. I wouldn't sleep on the idea that Facebook already having a lot of experience with HDFS/Hadoop/Hive as being a big adoption driver for HBase. It's the dream of any product to partner with another very popular product in the hope of being pulled in as part of the ecosystem. That's what HBase has achieved. Given how HBase covers a nice spot in the persistence spectrum--real-time, distributed, linearly scalable, robust, BigData, open-source, key-value, column-oriented--we should see it become even more popular, especially with its anointment by Facebook.

5.2.7 Search with HBase

Lily is an implementation of search backed by HBase. Let us analyze the use of HBase in their code:

Listing 5.4 Lily IndexManager

```
package org.lilyproject.hbaseindex;
```

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableNotFoundException;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.node.ObjectNode;
import org.lilyproject.util.ObjectUtils;
import org.lilyproject.util.hbase.HBaseTableFactory;
import org.lilyproject.util.hbase.HBaseTableFactoryImpl;
import org.lilyproject.util.hbase.LocalHTable;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * Starting point for all the index and query functionality.
 *
 * This class should be instantiated yourself. This class is threadsafe,
 * but on the other hand rather lightweight so it does not harm to have multiple
 * instances.
 */
public class IndexManager {
    private Configuration hbaseConf;
    private HBaseAdmin hbaseAdmin;
    private HBaseTableFactory tableFactory;
    private static final byte[] INDEX_META_KEY = Bytes.toBytes("LILY_INDEX");

    /**
     * Constructor.
     */
    public IndexManager(Configuration hbaseConf) throws IOException {
        this(hbaseConf, null);
    }

    public IndexManager(Configuration hbaseConf, HBaseTableFactory tableFactory) thr
        this.hbaseConf = hbaseConf;
        hbaseAdmin = new HBaseAdmin(hbaseConf);
        this.tableFactory = tableFactory != null ? tableFactory : new HBaseTableFact
    }

    /**
     * Creates a new index.
     *
     * @throws IndexExistsException if an index with the same name already exists
     */
    public synchronized Index getIndex(IndexDefinition indexDef) throws IOException,
        if (indexDef.getFields().size() == 0) {
            throw new IllegalArgumentException("An IndexDefinition should contain at
        }

        byte[] jsonData = serialize(indexDef);

        HTableDescriptor tableDescr = new HTableDescriptor(indexDef.getName());
        HColumnDescriptor family = new HColumnDescriptor(Index.DATA_FAMILY, 1, HColu
```

```
                HColumnDescriptor.DEFAULT_IN_MEMORY, HColumnDescriptor.DEFAULT_BLOCK
                HColumnDescriptor.DEFAULT_BLOCKSIZE, HColumnDescriptor.DEFAULT_TTL,
                HColumnDescriptor.DEFAULT_BLOOMFILTER, HColumnDescriptor.DEFAULT_REP
        tableDescr.addFamily(family);

        // Store definition of index in a custom attribute on the table
        tableDescr.setValue(INDEX_META_KEY, jsonData);

        HTableInterface table = tableFactory.getTable(tableDescr);

        byte[] actualMeta = table.getTableDescriptor().getValue(INDEX_META_KEY);
        if (!ObjectUtils.safeEquals(jsonData, actualMeta)) {
            throw new RuntimeException("Index " + indexDef.getName() +
                    " exists but its definition does not match the supplied definiti
        }

        return instantiateIndex(indexDef.getName(), table);
    }

    private byte[] serialize(IndexDefinition indexDef) throws IOException {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(os, indexDef.toJson());
        return os.toByteArray();
    }

    private IndexDefinition deserialize(String name, byte[] jsonData) throws IOExcep
        ObjectMapper mapper = new ObjectMapper();
        return new IndexDefinition(name, mapper.readValue(jsonData, 0, jsonData.leng
    }

    /**
     * Retrieves an Index.
     *
     * @throws IndexNotFoundException if the index does not exist
     */
    public Index getIndex(String name) throws IOException, IndexNotFoundException {
        HTableInterface table;
        try {
            table = new LocalHTable(hbaseConf, name);
        } catch (TableNotFoundException e) {
            throw new IndexNotFoundException(name);
        }

        return instantiateIndex(name, table);
    }

    private Index instantiateIndex(String name, HTableInterface table) throws IOExce
        byte[] jsonData = table.getTableDescriptor().getValue(INDEX_META_KEY);
        if (jsonData == null) {
            throw new IOException("Not a valid index table: " + name);
        }

        IndexDefinition indexDef = deserialize(name, jsonData);

        Index index = new Index(table, indexDef);
```

```
        return index;
    }

    /**
     * Deletes an index.
     *
     * This disables the index table and deletes it.
     *
     * @throws IndexNotFoundException if the index does not exist.
     */
    public synchronized void deleteIndex(String name) throws IOException, IndexNotFo
        HTableDescriptor tableDescr;

        try {
            tableDescr = hbaseAdmin.getTableDescriptor(Bytes.toBytes(name));
        } catch (TableNotFoundException e) {
            throw new IndexNotFoundException(name);
        }

        if (tableDescr.getValue(INDEX_META_KEY) == null) {
            throw new IOException("Table exists but is not an index table: " + name)
        }

        hbaseAdmin.disableTable(name);
        hbaseAdmin.deleteTable(name);
    }
}
```

5.3 Performance and scalability

5.3.1 Avoiding single point of failure in HBase

TODO - this section requires reworking the material

In 0.20 the Secondary NameNode performs snapshotting. Its data can be used to recreate the HDFS if the Primary NameNode fails. The procedure is manual and may take hours, and there is also data loss since the last snapshot; In 0.21 there is a Backup Node (HADOOP-4539), which aims to help with HA and act as a cold spare. The data loss is less than with Secondary NN, but it is still manual and potentially error-prone, and it takes hours; There is an AvatarNode patch available for 0.20, and Facebook runs its cluster that way, but the patch submitted to Apache requires testing and the developers adopting it must do some custom configurations and also exercise care in their work. As a conclusion, when building an HA HDFS cluster, one needs to follow the best practices outlined by Tom White, and may still need to resort to specialized NSF filers for running the NameNode.

A common question on the Apache Hadoop mailing lists is what’s going on with availability? This post takes a look at availability in the context of Hadoop, gives an overview of the work in progress and where things are headed.

Background When discussing Hadoop availability people often start with the NameNode since it is a single point of failure (SPOF) in HDFS, and most components in the Hadoop ecosystem (MapReduce, HBase, Pig, Hive etc) rely on HDFS directly, and are therefore limited by its availability. However, Hadoop availability is a larger, more general issue, so it's helpful to establish some context before diving in. Availability is the proportion of time a system is functioning [1], which is commonly referred to as "uptime" (vs downtime, when the system is not functioning). Note that availability is a stricter requirement than fault tolerance – the ability for a system to perform as designed and degrade gracefully in the presence of failures. A system that requires an hour to restart (eg for a configuration change or software upgrade) but has no single point of failure is fault tolerant but not highly available (HA). Adding redundancy in all SPOFs is a common way to improve fault tolerance, which helps [2], but is just a part of, improving Hadoop availability. Note also that fault tolerance is distinct from durability, even though the NameNode is a SPOF no single failure results in data loss as copies of NameNode persistent state (the image and edit log) are replicated both within and across hosts. Availability is also often conflated with reliability. Reliability in distributed systems is a more general issue than availability [3]. A truly reliable distributed system must be highly available, fault tolerant, secure, scalable, and perform predictably, etc. I'll limit this post to Hadoop availability. Reasons for downtime An important part of improving availability and articulating requirements is understanding the causes of downtime. There are many types of failures in distributed systems, ways to classify them, and analyses of how failures result in downtime. Rather than go into depth here, I'll briefly summarize some general categories of issues that may cause downtime: 1. Maintenance – Hardware and software may need to be upgraded, configuration changes may require a system restart, and operational tasks for dependent systems. Hadoop can handle most maintenance to slave hosts without downtime; however maintenance to a master host normally requires a restart of the entire system. 2. Hardware failures – Hosts and their connections may fail. Without redundant devices, or redundant components within devices, a single component failure may cause the entire device to fail. Hadoop can tolerate hardware failures (even silent failures like corruption) to slave hosts without downtime; however some hardware failures on the master host (or a failure in the connection between the master and the majority of the slaves) can cause system downtime [4]. 3. Software failures – Software bugs may cause a component in the system to stop functioning or require a restart. For

example, a bug in upgrade code could result in downtime due to data corruption. A dependent software component may become unavailable (eg the Java garbage collector enters a stop-the-world phase). Hadoop can tolerate some software bugs without downtime; however components are generally designed to fail-fast – to stop and notify other components of failure rather than attempt to continue a possibly-flawed process. Therefore a software bug in a master service will likely cause downtime. 4. Operator errors – People make mistakes. From disconnecting the wrong cable, to mis-configured hosts, to typos in configuration files, operator errors can cause downtime. Hadoop attempts to limit operator error by simplifying administration, validating its configuration, and providing useful messages in logs and UI components; however operator mistakes may still cause downtime. Use cases In order for a system to be highly available, its design needs to anticipate these various failures. Removing single points of failure, enabling rolling upgrades, faster restarts, making the system robust and user friendly, etc are all necessary to improve availability. Given that improving availability requires a multi-prong approach, let's take a look at the relevant use cases for limiting downtime. 1. Host maintenance – If an operator needs to upgrade or replace the primary host hardware or upgrade its operating system, they should be able to manually fail over to a hot standby, perform the upgrade and optionally fail back to the primary. The fail-over should be transparent to clients accessing the system (eg active jobs continue to run). Host maintenance to slave hosts can be handled without downtime today by de-commissioning the host. 2. Configuration changes – Ideally configuration changes to masters should not require a system restart — the configuration can be updated in-place or fail-over to a hot standby with an updated configuration is supported. In cases when they do, the operator should be able to restart the system with minimal impact to running workloads. 3. Software upgrades – An operator should be able to upgrade Hadoop's software in-place (a "rolling upgrade") on slave nodes and via fail-over on the master hosts so there is little or no downtime. If a restart is required it should be accelerated by quickly re-constructing the system's state. 4. Host failures – If a non-redundant hardware component fails, the operating system crashes, a disk partition runs out of space, etc. the system should detect the failure, and, depending on the service and failure, (a) recover, (b) de-commission itself, or (c) fail over to a hot standby. Hadoop currently tolerates slave host failures without downtime, however master host failures often cause downtime. In practice, for a number of reasons, master hardware failures do not cause as much downtime as you might expect: In large

clusters it is statistically improbable that a hardware failure impacts a machine running master services, and operations teams are often good at keeping a small number of well-known hosts healthy. Because there are few master hosts redundant hardware components can be used to limit the probability of a host failure without dramatically increasing the price of the overall system. Highly Available Hadoop

5.3.2 Monitoring HBase

5.3.3 HBase performance limitations

5.3.4 Improving HBase performance