Designing Highly Scalable Web Applications using Data Stores

Marten Klencke <mklencke@gmail.com>

Master Thesis

Vrije Universiteit Amsterdam

Computer Science - Internet & Web Technology Specialization

Student Number 1271229

Supervisor: Guillaume Pierre

Second Reader: Guido Urdaneta

November 29, 2010

Abstract

This paper describes the issues that arise when, and also the benefits that come from, rebuilding an existing, relational database-based web application, to run on top of a distributed, column-oriented data store. The research was done by porting a MySQL-based Wiki software called Wikka to HBase while carefully analyzing the necessary steps and caveats associated with this work. We tested the scalability properties of the resulting software on a dedicated compute cluster using custom benchmarking software as well as a previously developed framework called WikiBench. The end result is proof of the (read) scalability of such a system and a set of rules and guidelines for any similar porting effort.

Contents

1 Introduction

In recent years we have witnessed a significant paradigm shift in software and personal computing in general. With the advent of the World Wide Web and ubiquitous broadband access to it, desktop software is increasingly undergoing a migration from the Personal Computer back to centralized servers. While there are tremendous benefits to this "cloud computing" approach, including world-wide data availability, cooperation and social networking aspects, it also brings up an important new issue: how exactly do we write software that scales to hundreds of thousands of concurrent users while ensuring correct operation and data integrity?

One main aspect of this scaling problem is the storage, management and querying of huge amounts of structured data. Traditional ACID-like relational databases are typically optimized to run on a single system, possibly with replicas to increase the sustainable number of concurrent read operations. Using techniques like sharding it is also possible to distribute writes across multiple machines, but this approach already cuts back on relational capabilities and may force us to rethink the design of our software in the first place.

Several key players in the web application market with extreme scalability demands, notably Google and Amazon, have taken a different approach and researched novel non-relational storage solutions optimized for horizontal scaling and now popularly referred to as "NoSQL" data stores.

Today, there are several options for highly scalable data stores in existence, some of them very mature, manageable and surely fit for deployment in production environments. However, the programming models and assumptions of these systems differ significantly from traditional relational databases, requiring software developers to rethink their designs and making the process of porting existing RDBMS-based applications to them nontrivial. Unfortunately, there is currently a lack of general guidelines for developers that are interested in utilizing these data stores.

In this paper, we attempt to fill this void, focusing on Apache HBase[1], which is inspired by Google's BigTable paper[4]. We present information about the workings and assumptions of this data store, as well as a generic approach for developers to follow when building new applications

[1] http://hadoop.apache.org/hbase/

on top of, or porting existing applications to HBase.

Existing Wiki software, Wikka[2], will serve as an example project for this paper. We ported it from MySQL to HBase and assessed the read performance of the resulting application. The porting process was generalized into reusable guidelines and we observed a near-linear speedup for read requests to the system, paving the way for further research in this area such as porting other software and assessing realistic read/write performance.

This paper is structured as follows. Section 2 explains key concepts and the design of HBase and other non-relational data stores. Related work is outlined in section 3. The difficulties and issues that come up when using HBase for (web) application development are discussed in section 4, after which the Wikka port, including design choices and performance results, is described in sections 5 and 6. Section 7 contains the general porting considerations and guidelines that are the goal of this paper and section 8 concludes.

2 Background

Most commonly, existing web applications use technology stacks along the lines of LAMP (Linux, Apache, MySQL, PHP). Dynamic web pages are generated using a scripting language (although compiled languages such as Java and C# are also very common) and communicate with database servers in order to retrieve dynamic content, add content and update existing content. Typically, such a database server runs a Relational Database Management System, or RDBMS, which provides high-level access to the data and takes care of many of the details while trying to optimize the querying and storage of data, as well as ensuring data integrity. However, these features are difficult to combine with modern scalability demands, which can often not be met by a single machine.[2, 9] The resulting distribution of data complicates the implementation of features like joins and transactions while still remaining consistent and responsive. A transaction, for example, could block other operations on all affected machines, diminishing performance overall.

Non-relational databases, on the other hand, are highly scalable but lack the higher-level rela-

[2] http://wikkawiki.org/

People

id	name	spouse	town
0	John	12	1
1	Jack	NULL	1
3	Bill	NULL	5
12	Jane	0	1
13	Anne	14	6
14	Ted	13	7

Towns

id	name	country
0	Amsterdam	1
1	NYC	4
2	Den Haag	1
3	Berlin	3
4	Paris	8
5	London	17

Friends

id	first	second
0	0	3
1	0	13
2	0	14
3	1	12
4	1	13
...	...	...

Figure 1: Relational Model Example

tional features and consistency guarantees of RDMBSs. The following subsections give an overview of RDBMS features and then explain why some of them can not be scaled without compromise. Finally, the data storage model of BigTable and HBase is introduced, after which the section concludes with a quick look at some other alternative data storage models.

2.1 Relational Database Management Systems

In an RDBMS, data is stored using the relational model.[5] Tables with predefined columns store the individual entries as tuples of values. These values may be references to other rows, which can be in other tables, making it possible to form complex relationships between data entries . Figure 1 illustrates some properties of the relational model:

- Rows usually have a main id by which they can be looked up and referenced. This is called the primary key. Such a key is indexed and is usually, but not necessarily, a numerical value. For example, indexed string values are also possible, but these result in lower performance.
- There are several types of relationships, notably one-to-one, one-to-many, many-to-one and

many-to-many. One-to-one relationships can be modeled using a separate column in a table (like the "spouse" and "town" columns in the "People" table). The other types of relationships are commonly modeled using separate dedicated tables, as illustrated by the "Friends" table.

- Relational databases promote the notion of "normalizing" data. This means that it is preferable to store each piece of information only once and reference it instead of duplicating the information itself. Sometimes this imposes rules and guidelines on top of the relational data model. For example, if the "spouse" column in the "People" table has a value in each partner's row, the information content is the same as when one of them is NULL. In the latter case, however, this may decrease performance because an additional index or table scan is needed to find the spouse of the person that has the NULL value. Additionally, there must be a rule that determines which of the rows will store the NULL value (e.g. the one in the row with the highest primary key value).

Data from RDBMS is usually queried using the Structured Query Language (SQL), making it very easy for programmers to query and join data from multiple tables using simple statements. SQL supports many more operations, such as atomic transactions, inserts, value-generating functions (e.g. date and time), simple arithmetic, etc. A typical RDBMS will employ many different strategies to optimize the supported operations using techniques like indices and several layers of caches.

The downside to being able to utilize all these features in a generic way is that the database system can mostly not reason about the data itself. Therefore, the (normalized) data becomes a highly integrated, interdependent network of data for which ACID (Atomicity, Consistency, Isolation, Durability) properties must be maintained. As explained above, this makes an RDBMS difficult to scale horizontally across multiple machines that are running in parallel.

However, there are several ways to circumvent these limitations and scale an application using a relational database for its storage:

Replication creates multiple read-only "mirrors" of a database server. Adding a replica essentially increases the number of concurrent read requests that the system can handle. Write requests, however, become more expensive as they must consistently be performed on all replicas as

well as the master server.

Sharding splits up the data into parts that reside on different database servers. Sharding can be done by moving tables to different servers or even splitting up the data in a single table. The major drawback of sharding is that it makes certain operations like joins and global aggregations unusable. These must then be performed in the application software that can talk to all database servers. Essentially, many relational capabilities, which are a major selling point for RDBMS, are lost while the overhead of managing the databases is still there.

The question then is: why use an RDBMS at all? The next section will discuss a type of NoSQL database, which does not offer relational capabilities but instead offers many scalability features.

2.2 BigTable and HBase

In 2006 Google published its landmark paper about BigTable[4], a distributed column-oriented data store that focuses on availability and consistency. Its API exposes a sparse table structure with unique, alphanumerically ordered, row keys and arbitrary columns per row. Although it is mostly schema-less, columns are grouped into predefined column families, which are optimized for locality and also serve as a level of configuration. Hence the description "column-oriented data store". As a result, data that is commonly accessed together should be grouped under a single column family.

Cells are addressed using a (row key, column family, column) tuple. Additionally, cells can be versioned, where column families can be configured to allow for either a maximum number of versions or employ a temporal expiration scheme.

BigTable guarantees atomicity of single-row operations: multiple subsequent operations on a single row can be grouped together and will be executed in a single sweep. Parallel processes trying to access the same row will always see the row in a consistent state. It is also possible to atomically increase a cell integer value by a given amount, which is useful for building things like reliable counters.

Because row keys are alphanumerically ordered, ranges of rows can be retrieved by specifying a start and an end row key. It is not necessary for these range delimiters to actually exist in the

data store. The ordering also enables BigTable to optimize for locality of successive rows.

Shortly after the publication of the BigTable paper, an effort was initiated to build HBase[3], aiming to implement the features provided by BigTable. It is now an open-source project under the Apache Foundation. This Java-based system can be accessed directly from Java code using an intelligent client class that connects to an HBase cluster. It is also possible to communicate with HBase via a high performance Thrift[1][1][4] gateway, enabling a variety of programming languages (including C++, Java, Python, PHP and more) to utilize it. However, this Thrift API offers only a subset of the full functionality of the Java-based API (see section 4).

HBase can run on a cluster of commodity hardware. It's underlying file system, the Hadoop Distributed File System[5], provides automatic (and configurable) replication and distribution of data. HBase makes sure that column families are stored together on disk.[6] The system automatically splits the data into regions of adjacent rows. These regions are then distributed over the region servers, which are like data nodes in a distributed file system. The (configurable) maximum region size affects how many regions exist in the data store and how easily regions can be distributed over all the region servers in the system. For example, if this parameter is larger than the combined size of data in a table, there will only ever be one region, which can only be served by one region server.

Scaling up an HBase installation is done by adding more region servers and making sure each of them is responsible for a portion of the regions.

2.3 Consistency

The lack of transactions over multiple rows in data stores like HBase means that, if data to be written spans multiple rows and/or tables, the system may become inconsistent in case of failures (e.g. in the data store, internal network, etc.) between separate (but logically transactional) write operations or because of race conditions between simultaneous batches of operations.

However, it is important to note that many such inconsistencies are not necessarily a problem

[3] http://hbase.apache.org/
[4] http://incubator.apache.org/thrift/
[5] http://hadoop.apache.org/hdfs/
[6] http://wiki.apache.org/hadoop/Hbase/HbaseArchitecture

for web applications. For example, website users will not generally notice a visitor counter that is slightly off or a page that is missing from the most recently edited pages list on a Wiki. Furthermore, such inconsistencies can be fixed periodically (e.g. by a consistency checker that is run every night).

Another famous example is the Amazon shopping cart. It can happen that two customers order the same product within a short time-span. Obviously, if there is only one such item in stock, only one customer can receive it. However, there may a discrepancy between the actual item count and the item count in the web site database, causing both orders to succeed. In such a case, Amazon has opted to convert the eventually failing second order into a back-order and notifying the customer. The important thing is to gracefully handle and sort out inconsistencies when they do start to become visible.[10]

Consistency is not an all or nothing property. Several degrees of consistency exist, including so-called "eventual consistency". A system that has this property is guaranteed to always end up in a consistent state if given enough time. In the meantime, there may be inconsistencies between parts of the system. For example, writes may be lazily propagated throughout a network of nodes, in which case they end up at every node eventually but until then not every node has the same information.

In 2000, Eric Brewer of the University of California at Berkeley first introduced what would eventually become the CAP Theorem of Brewer's Theorem in his PODC keynote "Toward Robust Distributed Systems"[3]. It states that a distributed system can not have all three of the following properties, and so must always part with at least one of them:

1. Consistency. All nodes see exactly the same data at all times. Once a write operation has been completed, all subsequent read operations in the system from then on will see that write. As mentioned above, several lesser degrees of consistency exist.

2. Availability. The system is operational and responsive at all times, including when nodes fail.

3. Partition Tolerance. If the system is at some point split into different partitions that are no longer connected to each other, it will still be able to perform all read and write operations. This may, for example, be the case if there is a high degree of replication and the system can

successfully merge data when partitions are reconnected at a later point in time.

According to the CAP Theorem, we can not have all three properties at the same time and for the same data. For example, a system that has no replication at all can easily be highly consistent (each request is routed to the correct node), but must give up availability when partitioned. This may be implemented as a "please try again later" response to clients. On the other hand, if it implements replication and remains available at all times the different partitions will end up with different data when clients start writing.

Therefore, for a combination of data and point in time, a choice must always be made, and systems usually have a preferred default consistency model. BigTable and HBase are CA systems[4], meaning they are strongly consistent and highly available in the face of failures, but at the cost of not handling network partitions well. Should a network partition inadvertently occur, they may become unavailable.

2.4 Other Data Stores

Several other non-relational column-oriented data stores exist, such as Amazon SimpleDB, Hypertable and Cassandra, with varying CAP properties. For example, Cassandra is very similar to HBase in terms of data format, but eventually consistent. Although it is a highly available data store that will continue to work when faced with network partitions, some writes may take time to be available throughout the whole system. Recent releases have made these properties more configurable, enabling developers to choose the properties that are most suitable for their data and even varying them over time.[7]

In addition to BigTable-inspired data stores, many other distributed, non-relational data stores exist that implement different designs, again sporting various consistency models[9]. Most notably:

- Simple key-value stores such as Amazon Dynamo[6], Voldemort[8], MemcacheDB[9] and Redis[10]. As the name implies, these typically do not have more functionality than writing and reading

[7] http://cassandra.apache.org/
[8] http://project-voldemort.com/
[9] http://memcachedb.org/
[10] http://redis.io/

a value by key.

- Document databases such as CouchDB[11] and MongoDB[12]. Typically, they are schema-free, but store their documents in a known format such as JSON or XML. Document databases can be queries via "views", describing parts of the documents to be queried, which are executed in parallel across the nodes of the distributed system.

- Graph Databases. These types of database are based on graph theory and optimized for storing vertices and edges with accompanying properties. For example, a database of people, their interconnections (friendships, work relations) and their individual properties such as age, interests, etc.

In this paper we will limit ourselves to HBase and, by extension, other systems based on BigTable for the porting effort. Building a highly scalable web application on any of the other types of non-relational data stores remains a subject for further research.

3 Related Work

As touched upon in the introduction, the problem of efficiently distributing web applications over multiple machines is a very topical issue and area of research. Many efforts exist to implement solid replication of traditional RDBM systems, scale web server software and optimize caching behavior. However, the main related work that is of significance to this paper concerns data denormalization, decentralization and synchronization methods and similar efforts of building web applications on top of non-relational data stores.

3.1 Denormalization

Typical web applications based on a relational data model have interdependencies between data in queries and transactions that drive them. As a basic example, a Wiki page displaying the name of the latest author will need to fetch the page contents as well as the author information. Typically

[11] http://couchdb.apache.org/
[12] http://www.mongodb.org/

these two data items are stored in different tables, making it necessary to perform a join query or multiple queries on the database. Such interdependencies using foreign keys can be much more elaborate, resulting in a tightly integrated database that can not easily be split up to run on multiple machines in a scalable cluster.

One way to address this problem is to denormalize the data. That is, to introduce redundant data in order to remove interdependencies between tables. In the previous example of a Wiki page showing the page contents and the name of the latest author, the latter could be redundantly stored with the page contents, instead of only in a separate users table.

As demonstrated by Wei et al.[11], if this technique is executed correctly, the data can be clustered into disjoint data services which can then run on separate machines and scaled individually.

Non-relational data stores such as HBase inherently require similar methods as it is not possible to define relationships between data tables.

3.2 Decentralization

While techniques such as denormalization and caching are intended for improving the performance of traditional client-server architectures, novel architectures are also being researched. Urdaneta et al. propose the design of a decentralized system for hosting Wikipedia content in a collaborative fashion.[7] Similar to large-scale collaborative computation efforts such as Seti@Home, the idea is to ask users to share storage and bandwidth, effectively creating a distributed hosting platform consisting of (tens of) thousands of heterogeneous machines.

Several problems need to be solved in order for such a system to be effective. First, data must be distributed and replicated in such a way that adequate performance is maintained for all users. This involves correct routing and page placement protocols and algorithms. Additionally, new fault tolerance and security issues arise in such a large network of untrusted servers.

The proposed page placement techniques could be used in our implementation of a Wiki on top of HBase (influencing region composition and placement), but this does not fall within the scope of this paper. It remains an option for further research.

3.3 Similar Efforts

LilyCMS[13] is a CMS that has been built from the ground up on top of HBase. Its source code has been opened in July 2010, so it can serve as an example and inspiration for new porting and/or development projects involving web applications using HBase.

However, in general the pool of existing applications is not very great. There are, however, a number of development tutorials[14], presentations[15] and schema design examples[16] available on the web.

4 Issues

The main issues that arise when porting an existing relational database-based web application onto a non-relational data store come from the fundamental differences between these storage techniques. Additionally, in order to avoid balancing and performance problems, special care must be taken while configuring the data store cluster. The following subsections discuss such issues in porting from MySQL to HBase, as in our Wikka example. Subsequently, the process that we went through while porting Wikka to HBase is described in more detail in section 5.

4.1 Differences Between RDBMS and Column-Oriented Data Stores

While both an RDBMS and a Column-Oriented Data Store (CODS) use tables of data, there are several differences between the two that make the task of porting a web application from the former to the latter far from straight-forward. The following is list of major differences that impact any porting effort.

1. Secondary indexes are not possible in a CODS. In order to be able to search efficiently using a field other than the row key, separate index tables have to be constructed and kept up to date.

[13] http://www.lilyproject.org
[14] http://www.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases
[15] http://www.techcast.com/events/apacheconus09/mstack/
[16] http://www.slideshare.net/hmisty/20090713-hbase-schema-design-case-studies

2. There are no transactions in a CODS that transcend the row-level. Using an RDBMS, it is possible to create atomic transactions that affect multiple rows and even multiple tables. If the same data can not be kept in a single row in the corresponding CODS, transactions are not possible and data can become inconsistent in case of failure.

3. Joins between tables can not be performed in a CODS. This means that data either has to be moved to another table that it would normally join with or that data must be denormalized. Again, this introduces the possibility of data inconsistencies.

4. CODS systems do not support SQL, the Structured Query Language, which takes much of the work out of the application developer's hand. When using a CODS, many operations have to be implemented in the application code due to the fact that a CODS typically only has a very simple querying API.

5. It is not possible to define an automatically incrementing index in a CODS.

Of course, CODS also offer some exploitable features that RDBMS do not have:

1. Columns (with column families) can arbitrarily be defined from the application code. They are not part of the schema.

2. Thanks to their non-relational nature, CODS theoretically scale linearly over multiple servers.

3. CODS typically include several mechanisms that make scaling very easy. Data are automatically rebalanced, replicated and restored in case of failures.

The automated linear scaling promise is particularly interesting. It ensures that the web application keeps performing well, even at very high data throughputs.

4.2 Performance and Other Issues

For optimum performance, it is extremely important to configure HBase correctly and make sure the regions are distributed over the cluster nodes sufficiently. Especially with small data sets, the default configuration of HBase has the tendency to become unbalanced. This is because the default

values for maximum region size in HBase and block size in Hadoop are tuned to support very large data sets. For example, the maximum region size is 256MB by default. This means that a region will not be split if it does not grow beyond that size, so if the cluster has 10 nodes and the total data in a table occupies only 1GB, a maximum of about 4 region servers will actually be used, with the other servers remaining idle. Therefore, it is necessary to carefully tune such parameters in the Hadoop and HBase configurations.

Another issue arises when the web application to be ported is not written in Java but another language like PHP (like Wikka). The Thrift interface to HBase does not offer the full functionality of the Java API. For example, there is no function to query for the existence of a row, so this must be done by trying to retrieve a cell from a row and check for exceptions. These problems may gradually disappear as HBase becomes a more common server-side solution.

5 Design

This section describes the methods we used and the design choices we made while porting Wikka from MySQL to HBase. Although this was a non-linear iterative process, the general workflow that emerged was as follows:

1. Analyze the current database tables and their schemas.
2. Find out the queries that are performed on the database.
   (a) Analyze the PHP code generating the queries.
   (b) Turn on logging in MySQL[17] and interact with the web application, noting which queries are being generated.
3. Design HBase tables.
4. Gradually modify PHP code to use the HBase backend instead of the MySQL backend. This results in a hybrid system during development, so we must make sure the data in both backends is coherent.

[17] http://dev.mysql.com/doc/refman/5.1/en/query-log.html

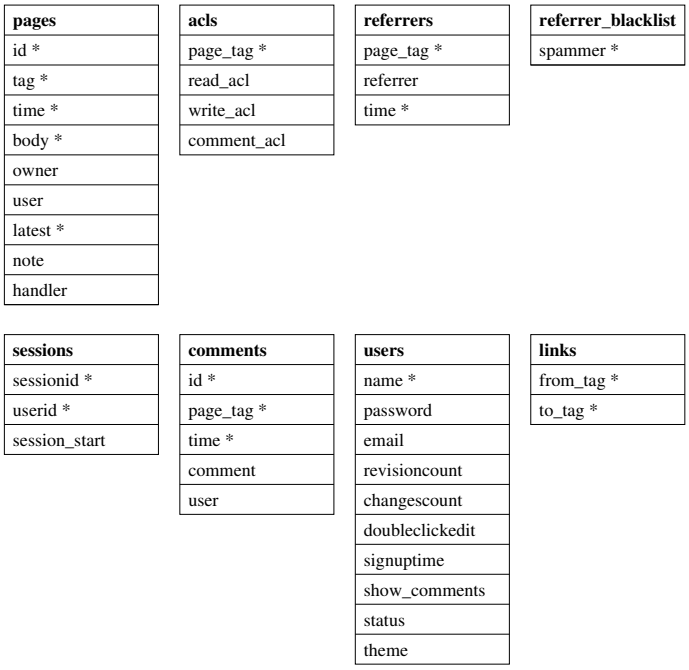

Figure 2: Wikka MySQL Schema

Figure 2 shows the different MySQL tables in Wikka and their fields. An asterisk next to a field name indicates that this field has an associated (primary or secondary) index. The fact that such fields have indices alerts us to the fact that there must be code in Wikka that uses them as lookup keys. As a consequence, we must provide the same functionality in our HBase tables which may require data redundancy.

MySQL tables often have numerical primary indexes, such as the fields named "id" in the pages and the comments table. In HBase, such primary indices are often superfluous due to the fact that rows are indexed by strings. For the equivalent of the 'pages' table, we can simply use the 'tag' as our key, which makes sense because all pages are identified by a unique tag. It is also how they are identified in links.

One important thing to remember is that HBase rows are alphanumerically sorted and regions of the complete data set are distributed over the data nodes in the cluster. It is a good idea to vary the row key strings as much as possible in order to distribute the data, and thus the load, well

over the available nodes (this is not taking into account the various access patterns and popularity differences between pages, which is proposed as further research in section 8).

The pages table was the first one we ported to use HBase. As its row key, we used the hexadecimal representation of the MD5-sum[18] of the page tag (e.g. `14f92a8ce5229727c33cffee0bc76ef5`), which ensures a good distribution of keys. The actual page tag (e.g. "Home", "FrequentlyAskedQuestions", etc.) does not have to be stored because it can always be extracted from the current URL. Then its MD5-sum is calculated in order to fetch the correct data from HBase.

The id field no longer serves a purpose and can be omitted. We put the owner and handler fields into their own columns under the "meta:" column family. This ensures that all these fields are located very close to each other on disk. Furthermore, it enables the later addition of new metadata fields without changing the database schema (they will all end up under the "meta:" column family).

Wikka stores a backlog of page versions. Each page version has its own time, body, user and note. While HBase has facilities to store multiple versions of data cells, they are not suitable in this case. When using versioned cells, one has to specify either a maximum number of versions to store (when full, a new version will push out the first version) or an expiry time. However, we want to be sure all our versions remain in the system.

The method we chose to implement page versions makes use of the fact that new columns can be added on the fly in HBase. For each of the three affected fields (body, user and note), we create a column family, making the latest version accessible via the <fieldname>:latest columns and older versions via <fieldname>:1, <fieldname>:2, etc. A new meta data field "meta:versioncount" is used to quickly assess the number of versions available. The HBase API now allows us to fetch a specific version of the page or all versions by fetching all columns in the column families.

ACLs is an interesting table because, upon inspection of the schema and code, it becomes clear that it is not necessary to put this data in a separate table at all. The relationship between ACLs and pages is strictly one-to-one. Therefore, we simply added yet another column family "acl:" to the pages table, under which the columns "acl:read", "acl:write" and "acl:comment" are created.

[18] http://en.wikipedia.org/wiki/MD5

The links table is used to, for a page A, quickly assess all pages B, C, ... that have backlinks $B \rightarrow A$, $C \rightarrow A$, ... in their bodies. This is yet another example of a feature that can easily be implemented using a new column family. We added the family "link_from:". Each time a page B is updated it enumerates all links in its body and creates a corresponding "link_from:<md5>" column in the row of each page it links to (with a value of "1"). When a link is removed during a body update, the corresponding backlink is also removed. As a side note, this workflow can cause inconsistencies due to failures or race conditions. For example, if a process updating a page dies prematurely, stale backlinks may remain in the data store. Therefore, it is a good idea to periodically run cleanup processes. We did, however, not implement this.

Comments, again, are strongly linked to pages and may as well be stored in the page table, which also ensures data proximity on disk and thus positively impacts performance. As this, unlike acls, is a many-to-one relationship and the comments table has multiple fields, this is not achievable in an obvious way. We have to resort to one of a number of tricks. In all cases, the time field serves as a nice way to name comment entries:

1. Use multiple column families. For example, "comment:<time>" (stores the comment) and "commentwriter:<time>" (stores the username) together constitute a comment on a page. However, column families may be stored separately on disk so this may necessitate additional seek operations and slow down the system. Also, the <time> values must absolutely remain consistent.

2. Use one column family and add the user field to the column identifier. For example, "comment:<time>-<user>". The comment itself is then stored in the corresponding cell. An added benefit is that two people posting a comment at the same time will both get through because their comments, although having the same time value, will have different column names.

3. Use one column family "comment:<time>" and store both the username and the comment text in the corresponding cell.

The latter two options are more or less on par regarding effectiveness, but we chose to implement option 2 because of its collision avoidance property.

Finally, the users table could almost be ported as-is. The name field in the MySQL database is the primary key in the MySQL database and also serves as the row key in HBase. The password and email fields are stored under the "account:" column family. The other values are stored under the "settings:" column family. Again, using these umbrella column families enables the addition of new settings without changing the schema at a later point in time.

We disabled referrer tracking and referrer blacklisting in our port, so the corresponding database tables did not have to be created. Also, we did not implement full text searching of pages. One way to implement this at a later time is by periodically (e.g. nightly) indexing all page bodies and creating a separate dedicated index using full text indexing code. The fact that this index is sometimes slightly out of date will hardly be bothersome in reality.

Until now we have ignored the extra indices on the pages table: time (for sorting page versions by time), tag (for finding pages quickly by tag) and latest (for quick retrieval of the latest version of a page). The latter two are no longer necessary in the HBase system: tag is already the row key and the latest version of a page can easily be fetched from HBase by specifying the corresponding "<fieldname>:latest" columns. Time is only used for sorting, which we can easily do in PHP. In other software, there may be indices that are not so easily omitted or ported. See section 7 for possible solutions.

The final HBase tables that were implemented and that the Wikka code was ported to are depicted in Figure 3 .

6 Evaluation

6.1 Setup

No test suite exists for Wikka, so our port was verified to work correctly by testing it manually on a local machine that runs both the HBase backend and the web server. Once this was completed, we proceeded to test its read performance and scalability when run on multiple machines in parallel. For this, we used the DAS-3[19] cluster at the Vrije Universiteit. It consists of 85 dual-CPU / dual-

[19] http://www.cs.vu.nl/das3/

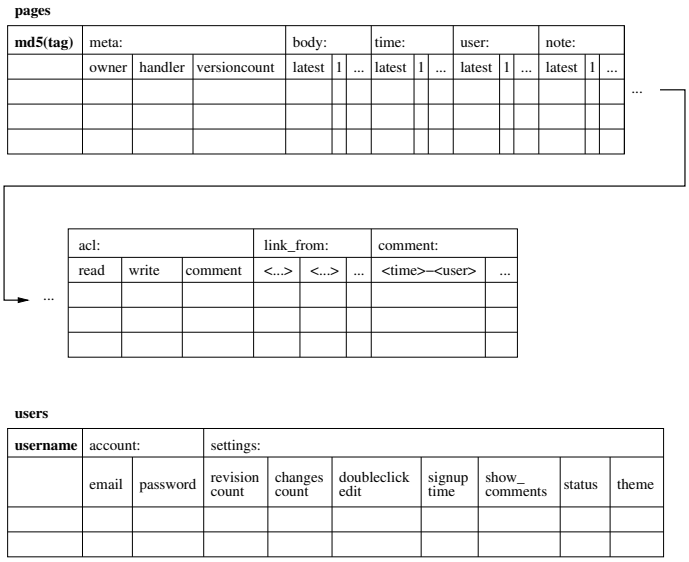

Figure 3: Wikka HBase Schema

core 2.4 GHz AMD Opteron DP 280 compute nodes, each having 4 GB of memory and 250 GB of local HD space. The cluster is equipped with 1 and 10 Gigabit/s Ethernet.

Figure 4 illustrates our deployment setup. The Wikka PHP files were served using lighttpd[20] 1.4.26 running PHP 5.2.13 and XCache 1.3.0 via FastCGI 2.4.0. To ensure that the webserver is not a bottleneck, we ran several webservers that are accessed by the benchmarking clients in a random fashion. For further assurance, we monitored the system load on each machine running a web server and made sure the values were low enough. To connect PHP to HBase, each webserver runs a local Thrift client that connects to the master HBase server.

It is of vital importance to properly configure all server software. For example, the lighttpd FastCGI module must be configured to spawn enough children and the number of allowable requests must be as high as possible. Of course, all these settings have to be set within system limits like available memory and maximum number of file descriptors. In our test setup, we used `PHP_FCGI_CHILDREN = 144`, `PHP_FCGI_MAX_REQUESTS = 10000` and `max-procs = 2` for the lighttpd FastCGI configuration.

[20] http://www.lighttpd.net/

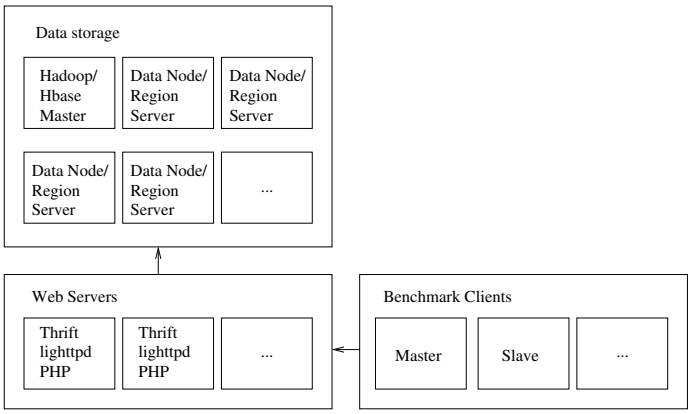

Figure 4: Benchmarking Cluster Setup

6.2 Synthetic Benchmarking

Our first benchmark was conducted to assess the scalability of the system for random read requests. Ideally, we would find that the number of read requests that can successfully be processed within a certain time span increases linearly as HBase region servers are added.

We filled the HBase data store with 100,000 pages, each containing around 8KB of text and 10 links to other pages. The page tags were generated by translating the page number (0-99,999) to a string using a simple conversion scheme: 0 becomes Aa, 1 becomes Bb, etc. This ensures that all the pages have a valid camelcase wiki name. For example, 523 becomes FfCcDd.

Page content was copied from portions of standard Lorem Ipsum generated text. Within each page, 10 links to other pages were created by randomly choosing a valid page number and generating the corresponding camelcase wiki names. These names were then inserted into the page text at random locations. Note that, overall, this results in a more or less even link distribution. This does not mirror a realistic scenario, in which some pages are linked by many more pages than others. For our initial benchmarks, however, this was acceptable, because they served to help us get a general idea of the performance and maximum throughput of the system in a controlled fashion.

Imported into HBase, the 100,000 pages amount to around 800 MB of table data. For each benchmark run, we started the Hadoop Distributed File System with the desired number of data nodes and then copied the data into it. Once this was compete, HBase could be started (with as

many region servers as HDFS data nodes). In order to ensure optimal distribution of data, it was imperative to run a major compaction on HBase before starting the benchmark, which rebalances all the regions over the region servers. Also, the maximum region size was configured to be around 50 MB to make sure we have at least one region per server for up to 16 region servers.

Once the HBase and web servers were up and running, we could start benchmarking the maximum number of requests per second in relation to the number of region servers in order to get a measure of the scalability of the system. We defined the maximum number of requests per second to be the maximum number of requests that the system can serve in one second while responding to 95% of the requests within a time limit of 200ms.

Using a specially developed client, we requested pages from the webservers, with a configurable number of concurrent connections (threads) and average delay between requests. We measured the percentage of requests that finish within a limit of 200ms over an interval of 30 minutes. Doing this repetitively while increasing the number of client threads makes it possible to pinpoint the maximum throughput of the server cluster.

We kept a watch on the system load of all components of the test setup: Hadoop/HBase servers, Web/Thrift servers and benchmark clients. In order to ensure that the Web/Thrift servers were not the bottleneck, the amount of servers was increased until their loads were all below 1.0 during benchmarking. Additionally, the load of the Hadoop/HBase master server was verified to be far below 1.0.

The resulting request rates are shown in Figure 5. Note that, because the Thrift API does not provide a method for verifying the existence of a row key, one page request results in 10 requests for linked other pages. Therefore, the actual HBase query rate is close to 10 times the numbers in the graph.

Although the system appears to scale near-linearly, it does not scale optimally: doubling the number of servers does not double the maximum request rate. This may be due to hidden network or communication bottlenecks. We were unable to determine the exact cause.

As can be seen in Figure 5, a single HBase region server can sustain about $75 \times 10 = 750$ page reads per second until requests start taking longer than allowed. It is interesting to compare this to

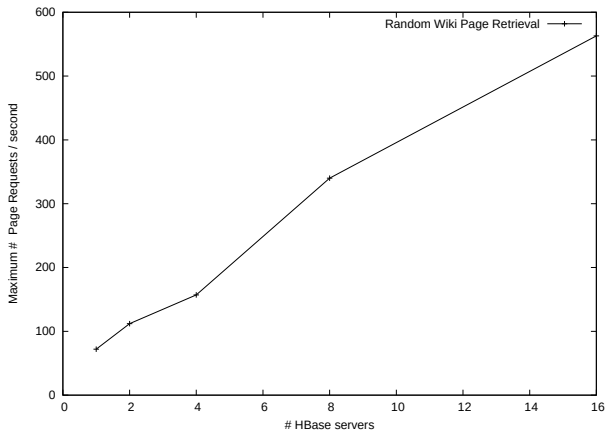

Figure 5: Maximum Request Rates

the original MySQL implementation. We used the same test setup with regards to page contents, lighttpd servers and the benchmarking software, but substituted MySQL for HBase in another test. What we discovered is that there is a major difference in the way in which an overload is handled by the systems. Where HBase requests just start taking longer and longer, MySQL will at some point simply refuse to accept connections. All requests complete far within the time limit (average around 50ms), but when around 200 requests per second (or 2000 page reads per second) is reached, the system starts refusing connections.

Another benchmark result is that the base speed of MySQL requests is significantly higher than HBase. A request to the MySQL system under light load finishes in a few milliseconds, whereas HBase usually require more than 10. This improves with caching over time, but is definitely not on par with MySQL. However, given the age of HBase as a project, we expect such performance issues to be addressed in the future. Furthermore, the scalability features of HBase make it a better candidate for dynamic cluster deployments.

6.3 Realistic Benchmarking

The benchmark described above assesses how the system performs under relatively constant and evenly distributed load. In order to test the system's response to loads (and load variations) that are more realistic (including load spikes, differences in page popularity, etc.), further benchmarking was done using WikiBench[8], a benchmarking tool designed to generate a realistic workload by parsing Wikipedia server logs and replaying them. Its accompanying software, TraceBench, converts raw access logs (kindly provided by the Wikimedia Foundation) into a format that can be fed to the main WikiBench software, which replays the requests. During this conversion process, TraceBench can reduce the number of requests by a specified percentage using one of several sampling methods.[8] It can also insert data for POST requests (to be used by WikiBench during benchmarking) by parsing the request line and fetching the associated data from a local instance of the Wikipedia database. However, because we only tested read performance, we did not use this feature.

The original WikiBench software does not have the ability to contact several web services in a round-robin fashion. We extended the software with some code that does exactly this, allowing the user to specify multiple host names in the `-suthost` parameter, separated by colon characters.

We used 24 hours of Wikipedia access logs sampled down each time to the desired average (because the real number varies greatly over a 24 hour period) number of requests per second over the course of 30 minutes. Before feeding the access logs to TraceBench, they were preprocessed using a script that selects only page GET requests (leaving out POST requests, requests for images, etc.)

Of course, the pages to be requested during benchmarking must also exist in Wikka, so we copied them from a recent Wikipedia database dump. Unfortunately, the wiki markup language differs significantly between MediaWiki and Wikka, which made it necessary to make some alterations to the Wikka rendering engine. It was changed to recognize inter-page links in the MediaWiki format and discard other markup.

Figure 6 shows the result of the WikiBench benchmark, which are strikingly similar to the results obtained by our own custom benchmarking client. We expected the differences in page

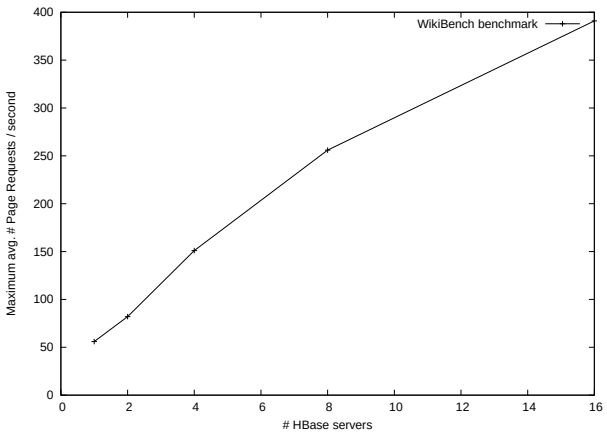

Figure 6: WikiBench Results

popularity[7] to have a greater impact on the speedup, but caching in HBase likely minimizes this effect. We suspect that the generally lower request rate can be attributed to larger page bodies and many more inter-page links (which, in our Thrift-based system, result in whole pages being fetched). In general, however, we see a promising near-linear speedup.

7 Discussion

From the experience of porting Wikka to HBase, a set of guidelines for future ports can be extracted. As described in section 5, the first step is to analyze the existing database schema and database access patterns from the source code of the software to be ported. We then propose the following guidelines for translating the RDBMS tables into HBase tables:

- Start with the main tables around which the others revolve. Choose a field to be used as the row key. Then proceed to work on the other fields. Group similar fields together under column families.

- Make sure row keys have a good distribution of values or employ hashing to generate keys. If

multiple rows are frequently retrieved together, it makes sense to design the row key in such a way that they are adjacent.

- Analyze relationships between tables
  - For one-to-one relationships, the fields of one table can simply be included in the other table.
  - For one-to-many relationships, use a column family and add use new columns under it to establish relationships.
  - Many-to-one relationships can simply be modeled as a simple field containing the row key of the data item in another table. However, for performance reasons, it may be a good idea to denormalize the data and opt for duplication of the data in each record instead of pointing to another table.
  - Many-to-many relationships are best implemented by adding a column family to each of the two tables that both work like the column family in one-to-many relationships. This results in redundancies but is much more efficient than using extra tables for storing the links. Remember that one link table is not enough to go both ways because it can only have one index: the row key.

- Catalog extra indices on the tables, which indicate that the web applications needs to be able to quickly find records based on these values. As HBase tables can only have 1 primary key, separate index tables are needed for secondary indexes. For example, if a table of people, using names as row keys, must also be quickly accessible by city of residence, create a new table for the cities in which the city names are the row keys and people's names are dynamically added as columns under a column family.

- As transactions over multiple rows or tables are not supported, try to make the corresponding modification actions as non-intrusive as possible. For example, mark rows as inactive instead of deleting them and periodically run cleanup jobs. Good inconsistency detection in the application code is also imperative.

- Carefully tune the HBase installation. Make sure the distribution of regions is efficient and regions are small enough that many of them are automatically created. Also, ensure a good replication factor.
- Use the native Java interface for communication with HBase if possible. The Thrift interface has several limitations in comparison.
- When faced with different ways to implement a feature (e.g. a column family with sub-columns or storing structured in the data store cell), carefully think of possible future developments of the application and preferably test each of the options for ease of use and performance before making a final decision.
- As always, general performance tuning guidelines apply to all other server software such as the web server, script engine, caches, etc.

8 Conclusion

We have successfully ported Wikka, a PHP+MySQL web application, to run on top of HBase with a variable number of server nodes. Benchmark results indicate that the read performance scales near-linearly, although the base performance is less than with MySQL. Hopefully this will change as HBase matures in the future.

As an added benefit, the experience of porting an existing web application has exposed some generic methods, guidelines and caveats that can be exploited while porting other software or building new software based on non-relational data stores.

Our results pave the way for a range of future research. Most notably, we have not measured write performance of the HBase-based application. It will be particularly interesting to see how it stacks up against a MySQL based system when writes are incorporated and also distributed over multiple server nodes. A related subject is the automatic balancing of data nodes. As pages in real-life wiki systems vary greatly in request rate, is it possible to use this information about access patterns when balancing/replicating pages over the available data nodes?

Additionally, the results from this paper can be used to port other software such as Wordpress and see if the guidelines hold up and can be extended. As Wikka is a relatively simple system, much more elaborate porting projects can be undertaken.

We would like to propose one final challenging research topic: how viable is the creation of a static database schema and source code analysis tool that can more or less automatically generate HBase schema's from existing RDBMS databases? An intermediate solution would be a tool that does not create HBase tables itself but (perhaps interactively) guides a developer through the design, giving suggestions along the way.

References

[1] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, April 2007.

[2] Martin Arlitt, Diwakar Krishnamurthy, and Jerry Rolia. Characterizing the scalability of a large web-based shopping system, 2001.

[3] E. A. Brewer. Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, 2000.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[5] Paul DuBois. *MySQL*. Addison-Wesley Professional, 2008.

[6] Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *In Proc. SOSP*, pages 205–220, 2007.

[7] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Comput. Netw.*, 53:1830–1845, July 2009.

[8] Erik-Jan van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, May 2009.

[9] Ian Thomas Varley. No relation: The mixed blessings of non-relational databases. Master's thesis, The University of Texas, Austin, Texas, USA, August 2009.

[10] Werner Vogels. Eventually consistent - revisited. December 2008.

[11] Zhou Wei, Jiang Dejun, Guillaume Pierre, Chi-Hung Chi, and Maarten van Steen. Service-oriented data denormalization for scalable web applications. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 267–276, New York, NY, USA, 2008. ACM.