A Review of Column-Oriented Datastores

By: Zach Pratt

Independent Study
Dr. Maskarinec
Spring 2011

Table of Contents

1 Introduction

Internet-scale applications are a relatively recent development in the realm of computing and technology, requiring massive scale to support hundreds of millions of concurrent users. These applications are distributed across multiple clusters, which consist of hundreds of server nodes located in multiple, geographically dispersed data centers. Applications which are hosted in such heavily distributed environments present a number of major engineering challenges. One such challenge is the reliable storage of application data and state, which is a critical requirement for applications of any size (DeCandia and Hastorun pg.205).

While the classical theories of relational database design and normalization are well understood, in practice the rigorous constraints placed on these systems do not always effectively meet the storage requirements of Internet-scale applications. Companies such as Google, Amazon, and Facebook have been actively researching the best methods to satisfy their applications' data storage needs. This research has resulted in the development of a new breed of non-relational, column-oriented distributed datastores. These datastores provide a simpler data model and focus on availability and incremental scalability, sometimes at the cost of consistency.

Classic relational database systems support robust data modes, a query language in SQL, referential integrity, and strong consistency. They are designed to handle a few thousand requests over the course of a day and utilize a single database instance on a server. Carefully planned partitioning schemes, replication, and clustering can enable systems to scale out to handle even more load. However, maintaining consistency and availability becomes increasingly difficult as the distributed system continues to grow. Strong consistency across server nodes requires multi-phase commit strategies and synchronous replication, which increase the latency of requests and contributes to the potential for failures.

With the aforementioned design challenges in mind, much research has been focused on alternative architectures whose goals are to provide incremental scalability, high availability, and high write throughput. The outcome of this research has produced solutions which present much simpler data models than relational databases and put a great majority of the control over the schema and consistency requirements into the hands of client applications. From the perspective of the datastore, data is often stored as pairs of uninterpreted byte strings. This strategy opens the door for integrity issues, since type consistency must be maintained by client applications. This problem is analogous to the difference between statically typed languages and dynamically typed languages. Additionally, distributed datastores typically follow one of two common logical data models: flat, unstructured sets of key-value pairs (hash tables) and semi-structured collections of key-value pairs (multi-dimensional maps) organized as column families.

At the heart of any datastore is the ability to durably store database objects. Highly distributed datastores provide methods for dynamically partitioning data and consistent distribution of the chunks of data across server nodes. The goal is to achieve balanced load and data distribution while avoiding the need for additional administrative overhead in managing underlying storage of the system. This improves scalability and flexibility, since nodes can be added and removed from the cluster as requirements evolve without direct intervention by a system administrator. It also reduced the potential for human error since these decisions are coordinated amongst the nodes in the cluster.

When contrasting relational database systems with column-oriented distributed datastores, one must consider the problems with which Internet-scale applications deal with. These problems are often quite different from applications that require lesser scale. Maintaining availability, relatively low latency, and an acceptable level of consistency in a heavily distributed

system is quite challenging. Many of these challenges were not under consideration when relational databases were first being implemented in the late 1970's and early 1980's. As a consequence, much of the scale-out functionality of relational database systems was added later on and was not part of the initial design considerations.

2 Background

The relational data model has existed for many years, dating back to the 1970's when it was first defined by E.F. Codd (Relational Model). Modern relational database management systems are based on the concepts of Codd's relational data model. These systems allow data to be organized into tables consisting of many columns and rows. Tables can reference other tables through foreign key definitions and many constraints can be defined to help maintain the integrity of the data. In short, relational database management systems provide many facilities for ensuring consistency and the ability for end-users to build up and maintain schema definitions centralized within the database system itself.

This follows the principles of the "separation of concerns model" where the database management system encapsulates the complexity around the physical data model. A data manipulation language and structured query language are also provided as facilities to manipulate and query the data contained with the database system. Client applications use these facilities to access the data according to the needs of the business logic contained within the application. Without these facilities, software developers would have to include logic to maintain the schema definition for the life of the application and to preserve the integrity of the data.

2.1 Basic Properties of an RDBMS

Aside from the general facilities provided to client applications, relational database management systems use a number of techniques to manage the data as many client applications

access and modify data. At the core of these techniques is the notion of a database transaction. E.F. Codd defines a database transaction as "a collection of activities involving changes to the database" (Codd pg.14). Transactions within relational database systems typically follow the ACID model. The ACID model and the significance of its various commandments are integral to the design of both relational database systems and column-oriented distributed datastores. The four characteristics of the ACID model are atomicity, consistency, isolation, and durability (ACID).

Atomicity requires that if any operations during the transaction fail, then the whole transaction fails. This provision avoids the possibility for partial modifications to the data, which could introduce inconsistencies. Data is said to be consistent if it is valid. For example, in a two node replicated relational database, data is only valid if the same copy of the data is on both nodes. This consistency model is sometimes referred to as strong consistency. Isolation requirements work to ensure that copies of the data remain consistent by avoiding changes to existing data by more than one transaction.

In order to achieve isolation, relational databases typically make use of locking strategies and data versioning. Durability requires that any changes made during a transaction will survive any fault in the system after the transaction has been committed. A common technique for handling durability is to write the changes made by a transaction to a transaction log. If there is a fault in the system, changes that were not persisted to disk can be recovered by reading the transaction log. It is important to understand that durability does not necessarily imply persistence. Data may be sufficiently replicated across several nodes in the system so as to achieve a high level of durability, with no single node having actually written the data persistently to disk.

Robust transaction handling provides a strong basis for recovery handling as well. For example, if we detect halfway through the transaction that a mistake has occurred, we can simply issue a rollback instead of a commit. Since data is not typically written to the live copy of the data until a commit is issued, this method allows us to discard all invalid changes quickly and efficiently. If our database system experiences a failure which terminates the process before all data has been persisted to disk after a commit, we can recover our data from the transaction log. All the previously mentioned features provide convenient means to maintain strong consistency and durability across a database environment.

Strong consistency in a distributed database essentially requires the system to sacrifice availability and latency in order to maintain consistency. If the requirements of a client application dictate that the system be available for writes no matter the situation, then the model of strong consistency may not be the best choice. Consistency in a distributed database also increases the latency for a given data modification, due to the need to synchronously replicate the data to other nodes. Additional requirements for consistency with a distributed database include either a centralized lock manager or a distributed lock manager. While relational database software packages provide many tunable parameters, they do not typically allow the user to tune the level of consistency for the needs of the application.

2.2 Example RDBMS Implementations

Two examples of relational database software packages are IBM's DB2 and MySQL. IBM's DB2 has been used for many years as the back-end to many mainframe-hosted business applications that are accessed by thousands of users. DB2 implements many of the features from the standard SQL specification and provides a number of tools to help administer and maintain

the database system. Maintenance of a DB2 database system is a fairly complex assignment and becoming a skilled DB2 database administrator requires years of experience.

A discussion on the scalability of a DB2 database system is a complex topic, compounded by the fact that there are so many different flavors of DB2. At the time of writing, the bleeding edge of scalable DB2 technology is DB2 pureScale (Eaton). pureScale is a proprietary clustered database technology that will only run on certain IBM hardware. It employs a central lock server and a seemingly unlimited number of storage nodes in the cluster which all make use of a clustered file system for storage. It is designed to provide the reliability of relational database technology with the ability to incrementally scale out as demand increases. There is not a great deal of information available on this technology, so it is difficult to say exactly how it compares to other distributed datastores.

While DB2 is a closed source and proprietary software package typically hosted on big iron, MySQL is open source and typically hosted on commodity hardware servers. MySQL is used by Google and other popular social networking sites such as Digg. Facebook has also used MySQL extensively in the past, but they are actively migrating to their own database technology, Cassandra. MySQL is highly configurable and as a result can be tuned to meet the needs of many types of workloads. MySQL is designed to be relatively easy to use and simpler to maintain than DB2 and Oracle.

MySQL supports two high availability, scale out options. One option is to use master/slave replication, where replication from the single master node to slave nodes is asynchronous (MySQL :: MySQL 5.0 Reference Manual). This option is good for applications that are light on writes and heavy on reads. A primary disadvantage with this option is that all writes must occur on the master node. The other option is to use the clustered version of

MySQL. The clustered version of MySQL uses synchronous replication and its own storage engine (MySQL :: MySQL 5.0 Reference Manual).

The high availability and scalability features of both DB2 and MySQL highlight two popular storage architectures with distributed and clustered systems: shared-disk and shared nothing. DB2 pureScale requires the use of IBM's GPFS distributed file system and therefore is a shared-disk architecture (Eaton). Both the MySQL replication and cluster setups are shared-nothing architectures where each node uses autonomous storage (MySQL :: MySQL 5.0 Reference Manual). Each architecture has its advantages and disadvantages, but both are commonly used in column-oriented datastores.

2.3 RDBMS Weaknesses

Database architects and data modelers will undoubtedly encounter times when the data at hand does not lend itself well to relational database design. Examples of such data include hierarchical relationships, recursive relationships, and key-value pair data. These examples can certainly be handled by a relational database, but not as efficiently as other databases that directly support these sorts of data models. Internet-scale applications must take the efficiency of their data model implementation into careful consideration. They cannot afford the penalties of non-optimal handling of the data by relational databases.

The storage subsystem is a key component to most non-trivial relational database deployments. This is primarily because relational database operations are I/O bound (joins especially) and they greatly benefit from disks that have low latencies and high throughput. Careful partitioning strategies can help alleviate these problems, but they usually must be defined as part of the table definition. If tables grow in an unforeseen way and a partitioning strategy was not a part of the initial design, most relational database systems do not offer the ability to

dynamically partition the table after it reaches a certain size. This means increased administrative work, potential down time, and additional regression tests to ensure that no new bugs were introduced into the design in order to partition each table.

Implementations that rely upon shared-disk further increase the reliance on high end hardware due to the way shared-disk storage is implemented. Shared-disk storage often takes the form of a storage area network (often referred to as SAN) which requires special network infrastructure to support it. To achieve the appropriate throughput, SANs often use fiber optic communication channels. Each of these components increase the overall cost of the system, especially when one accounts for the redundancy needed in order to achieve a high level of performance and availability.

Maintaining a high level of availability for database systems is another critical objective. A replication strategy is routinely used to help achieve the objective of high availability. That replication strategy is often synchronous due to the need for relational databases to maintain strong consistency. As the implementation grows in size and complexity, so does the network that is needed to support it. Synchronous replication increases latency and also is highly sensitive to network failures, which become more and more likely as the size of the network grows.

Eventually this becomes an impractical method for maintaining operations. At that point, the developer is left with a few choices, some of which are: scrap the system or build the replication logic into the client application to overcome the limitations of the database and network. The latter option greatly increases the complexity of the client application and moves a portion the data model out of the database and into the application. Some relational database systems, such as MySQL, offer the ability to use asynchronous replication. Under this replication strategy, the potential for inconsistencies in the data increases. Additionally, writes are not

transparently proxied and therefore they must occur against a single master node or master cluster. This requirement must be taken into consideration when designing the client application (MySQL :: MySQL 5.0 Reference Manual).

2.4 Motivation for Column-oriented datastores

As mentioned earlier, incremental scalability is a desirable characteristic of distributed systems. It makes the overall system more predictable, which in turn makes effective data center and capacity planning much easier. It can also provide sites with the flexibility of coping with big swings in load without having to redesign the entire system topology. Keeping the per-node administration overhead down is also critical. All of these benefits directly affect the costs of maintaining operations, which at Internet-scale, can be quite significant.

In its simplest form, incremental scalability means that when a system administrator adds a node to the cluster the overall capacity of the cluster will increase by a constant amount. This is why incremental scalability is also sometimes referred to as linear scalability. In order for datastores to achieve this, they must be able to dynamically partition the data over a set of nodes in the cluster (Lakshman and Prashant pg.2). The datastore must also be capable of self-balancing the data across its nodes, which in turn will theoretically lead to an even distribution of load.

No fewer than three major components are needed in order to achieve dynamic partitioning of data. The first component is a scheme to determine how to distribute the data. Two popular implementations of this scheme are a consistent hashing mechanism and row key range partitioning. The second component is storage for persisting data objects. The major determining factor for the storage solution is whether or not the datastore is based on a shared nothing or shared disk architecture. The third component is a method to determine the state and

capabilities of each node in the cluster. To achieve this, some type of membership protocol or central node directory is maintained. These topics will be covered in more detail later in the paper.

Relational database operations, such as joins, are I/O bound operations and directly benefit from robust storage subsystems. Column-oriented datastores do not support joins and they also buffer large amounts of data in memory. In general, reads within a column-oriented datastore are more CPU bound rather than I/O bound. Writes within a column-oriented datastore first occur against an in-memory data structure that is periodically flushed to disk. This results in sequential I/O for chunks of data, which is less I/O intensive than the many random writes of smaller objects that occur in relational databases.

It is because of these characteristics that column-oriented datastores can make use of storage subsystems consisting of commodity hardware and still remain performant. A node in such a system may only need 2 or 3 consumer grade (commodity) hard drives to sufficiently handle its storage needs. Contrast that with the needs of a node within a relational database cluster, where 15k RPM hard drives with fiber channel interfaces and expensive RAID controllers are the norm. Multiplying the cost difference times the number of nodes needed to effectively support operations yields a significant savings in overall cost.

Column-oriented datastores excel at workloads that favor availability and low latency over strong consistency. Examples of such workloads include persisting a user's session, managing a shopping cart, or providing storage for server log files. Since the per-node cost within the cluster is low, a high level of redundancy can be achieved much more affordably. These datastores are designed to take advantage of redundant hardware and replication is a focal

point of their design. In order to provide a high level of fault tolerance, many of these datastores (including Cassandra and Dynamo) make use of shared-nothing decentralized storage.

Many of these datastores allow developers to specify their application's requirements for consistency, latency, and redundancy. This design decision is inspired by the CAP Theorem, which was first devised by Eric Brewer and then later formalized by Nancy Lynch (Featherston pg.2). It states that when an exception occurs in the system, such as a network failure, the datastore must make a trade-off between availability and consistency (Gilbert and Lynch pg.2). Others have pointed out that even under normal operations, trade-offs must be made between consistency and latency to remain performant (Featherston pg.2). These principles run contrary to the ACID model of relational databases, but are a fundamental part of how many column-oriented datastores operate.

Amazon's Shopping Cart application utilizes Dynamo's decentralized storage model to ensure that writes to a user's shopping cart will always succeed. The ability to achieve this directly affects Amazon's business, in that each lost write could potentially mean a loss in revenue. In this case, Amazon would rather have multiple durable versions of a user's shopping cart than to lose any write operations. In this scenario, either the client application or the datastore must be capable of reconciling the differences in the versions of the shopping cart, thus increasing complexity. This is a common trade-off in these systems.

Building on the principle that the datastore must choose between latency and consistency, some client applications make use of lesser levels of consistency in order to achieve a high write throughput. Facebook's Inbox Search is such an application (Lakshman and Malik pg.1). According the Lakshman and Malik, the Inbox Search application must handle billions write per

day and support hundreds of millions of users. In this case, achieving a high write throughput is a matter of both quality of service and necessary in order to maintain operations.

At this scale, with cluster sizes on the order of over one hundred nodes in multiple data centers, maintaining strong consistency amongst replicas over a wide area network is generally unsustainable. One popular technique to overcome this problem is to utilize a method of eventual consistency. This technique allows for asynchronous replication, but it introduces the possibility of multiple versions of a data item being available at a given time. The focus of the problem is sustaining write throughput and not consistency.

In addition to sustaining write throughput, there are times when the need arises to provide storage for data that does not lend itself well to a relational data model. Sometimes this data is called semi-structured or unstructured data. Examples of this type of data include web pages, application sessions, and user extensible data. Column-oriented datastores excel at handling this type of data. Two popular ways of organizing this data include a flat set of key-value pairs or collections of columns with a row key. Amazon's Dynamo uses the flat key-value pair model while Google's BigTable and Facebook's Cassandra use the columns with a row key model.

The key-value pair implementations, such as Dynamo, make use of text-based keys and an associated binary value. The binary value is open to interpretation by the client application. The columns and row key implementations, like Cassandra, present a more robust data model where row keys can be associated with multiple column keys and their values. It is also important to point out that there is a single global namespace in datastores like Dynamo. The data model is entirely under the control of client applications. In datastores like Cassandra, there is support for hierarchical namespaces and logical groupings of columns into column families, which does centralize some of the data model in the datastore.

2.5 Basic Properties of a Column-oriented Datastore

In highly replicated distributed systems, there is often a great emphasis on the durability of data and not just persistence. Durability requires that writes from a particular operation are stored in such a way that if an exception were to occur, that data can be recovered. The copies might be stored in memory or on disk, but they are typically written to a commit log. Durability can be achieved through replication as well, while persistence is focused on storing the data on disk for long term storage. Some column-oriented datastores, such as Dynamo, support pluggable storage engine architecture for persistence (DeCandia and Hastorun pg.213). Other datastores like BigTable and Cassandra utilize their own storage engine; both use SSTables (Featherston pg.5).

Querying within a column-oriented datastore is often limited to key only lookups, which are provided by each datastore's own API. There is no query language, so data access is totally programmatic. Dynamo provides primary key only access, where every key in the Dynamo instance is unique since Dynamo does not provide a namespace (DeCandia and Hastorun pg.209). Other datastores provide namespaces, such as column families and keyspaces. These must be specified when querying data. To filter the data, some datastores such as BigTable allow for regular expressions to be passed via the query API call to reduce the number of rows that will be returned (Chang and Dean pg.3).

Versioning techniques are critical to the concurrency model of column-oriented datastores. Updates within a row are commonly implemented as atomic operations with a timestamp used to denote the version. In some cases, the latest timestamp is the true version. Since there is no notion of isolation within column-oriented datastores, it is entirely possible that the latest timestamp is not the true version. In these situations, it is up to the client to resolve

version conflicts. In Dynamo terminology, this problem is referred to as semantic reconciliation (DeCandia and Hastorun pg.210). These reconciliation techniques are needed because there is generally no notion of a transaction in column-oriented datastores.

Security and access control is not a strong focus with column-oriented databases. This is an area where relational databases are much more robust. Dynamo, for example, expects to operate in a trusted environment (DeCandia and Hastorun pg.209) and provides no security. BigTable, on the other hand, does support access control lists for column-families, which can be used to limit user capabilities (Chang and Dean pg.3). These access controls pale in comparison to row-level security, label based access control, and role based access control mechanisms supported by many mainstream relational databases.

In general, most popular column-oriented datastores place a lesser value on consistency and integrity compared to fault tolerance and low latency response. The alternative consistency model that these datastores focus on is called eventual consistency. To achieve high availability, replication amongst nodes is utilized extensively. Another wrinkle with respect to the integrity of column-oriented datastores is the fact that there is little or no support for types. Values are stored as uninterpreted byte strings, so it is up client applications in order to maintain consistent typing of values.

Support for recovery is effectively handled by using a commit log. Write operations are written to the commit log after finishing successfully. Without transactions, there is little support for rolling back operations and it is possible that a failure can occur during units of work. The primary goal of the commit log is to aid in providing durable storage for operations. Cassandra and BigTable log all writes to the commit log prior to updating their in-memory data structures or physical files.

3 Theory

In order to realize asynchronous replication on a large scale, column-oriented datastores make use of a technique known as eventual consistency. Eventual consistency, simply put, is a system where updates will reach all nodes eventually (DeCandia and Hastorun pg.207). Some column-oriented datastores provide a configurable consistency model which allows developers to pick the level of consistency that is needed. This choice can be based on the latency and availability needs of an application (Featherston pg.6).

One downside to the eventual consistency model is the possibility of multiple versions of a data item existing in the system at the same time. As a result, version reconciliation logic must either be present in the datastore, the client application, or both. Along these same lines, allowing concurrent updates of data without locking also has the potential to introduce multiple versions of a data item. On the flip side of this problem, using versioning without locking can improve availability and performance, since it eliminates the need for a lock manager. Distributed locking mechanisms add additional overhead and potential for failure, which are counter to the goals of column-oriented datastores.

Eventual consistency stems from the concepts behind the CAP Theorem, which states that it is impossible for a distributed system to provide the following three properties at the same time: consistency, availability, and network partition-tolerance (Gilbert and Lynch pg.1). Therefore, trade-offs must be made in order to maintain operations, especially when faults occur in the system. The defining factor for the overall consistency model for column-oriented datastores depends upon the intended level of fault tolerance. This is because of the relationship between consistency and the number of nodes involved replication.

The more replica nodes involved in the replication scheme, the more difficult it is to maintain consistency. Dynamo and Cassandra are examples of datastores that were designed to be very fault tolerant because of their replication capabilities, while BigTable was not. BigTable focuses on the partitioning of data, but not replication, while Dynamo and Cassandra focus on both. To achieve network partition-tolerance, a single point of failure must be avoided. The decentralized storage and replication of data to other nodes in the cluster is one way Dynamo and Cassandra attempt to achieve network partition-tolerance. This can result in multiple versions of the same key existing in the datastore at the same time. Therefore, these datastores must take into consideration the amount of reconciliation logic that it will need to implement in order to handle the side effects of eventual consistency.

Column-oriented datastores also require methods for dividing up the responsibility of data storage by dynamically partitioning chunks of data for load balancing purposes. Two prevailing methods for accomplishing this are to partition by row key or to use a consistent hashing function. With consistent hashing, the output range of the hash function has a fixed length and is conceptualized as a ring (DeCandia and Hastorun pg.209). Each node in the cluster is assigned a certain range within the hash space. When a client inserts a new key into the system, the key is hashed using the predefined hashing function and a node is chosen to handle the storage of the key. The node chosen by the hash function is often referred to as the "coordinating node" (Lakshman and Malik pg.2).

Besides acting as a load balancing technique, consistent hashing also aids the system in handling the addition of new nodes to the cluster. When a new node is added to the system, it must be assigned a portion of the hash space for which it will be the coordinating node. One convenient property of consistent hashing is that when a new node is added to the hash space, it

only affects its neighbors. With traditional consistent hashing, upon adding a new coordinating node, the new node will need to work with its neighbors in order for it to take over the storage of a subset of its data. This is because a subset of the data that each neighbor was responsible for should now be the responsibility of the new node.

For nodes to coordinate their responsibilities within the system, a method is needed in order for each node to learn of the state and capabilities of the other nodes in the cluster. Without coordination amongst nodes, administrative intervention would be needed. One way to accomplish this is to utilize a gossip-based protocol, where each node shares its knowledge of the cluster with other nodes; eventually all nodes will have the whole picture. Dynamo and Cassandra utilize a gossip-based protocol for nodes to learn of membership changes and to determine the failure of other nodes. A gossip protocol, among other things, helps other nodes learn of membership changes, the storage responsibilities of other nodes, and routes to other nodes within the cluster (DeCandia and Hastorun pg.209). To accomplish this, a node will periodically communicate with another node to share its membership information.

Another way that nodes coordinate with one another within a column-oriented datastore is through the use of quorums. Some column-oriented datastores allow client applications to designate their required level of consistency. This is typically done by specifying a required number of nodes that are needed to be involved in satisfying a read or write request. For a read request, the client can specify the minimum number of nodes that must successfully respond with data for a read request. For a write request, the client can specify the minimum number of nodes that must durably write the data successfully.

4 Implementations

The last five years have seen an explosion in content and services available on the Internet. Some of these services, such as Google Search, Amazon, and Facebook are used by hundreds of millions of users every day. There has been much research on how to reliably maintain operations at this scale. Part of that research has focused on database systems that are capable of meeting the demanding requirements of such a massive scale. Three successful implementations of such systems are Google's BigTable, Amazon's Dynamo, and Facebook's Cassandra. Each of these systems utilizes a non-relational data model, focuses on concurrency, and provides dynamic partitioning of data.

4.1 Google's BigTable

Google's BigTable was designed to serve the structured data storage needs of several Google services, including their web indexing system, Google Earth, and Google Finance (Chang and Dean pg.1). According to the developers, BigTable was designed to handle "petabytes of data across thousands of commodity servers" (Chang and Dean pg.1). Such is the kind of capacity needed for the largest of Internet-scale applications. Managing this amount of data requires specialized systems that were designed from the ground up to meet specific challenges. In Google's case, BigTable was designed to support distributed storage and dynamic partitioning of data.

BigTable adopts a different data model from traditional relational databases, but still aims to provide a form of structured storage. The data model of BigTable can be conceptualized as a multi-dimensional sorted map. As you may recall, the basic map abstract data structure is essentially a collection of key-value pairs. BigTable allows for a set of key and value pairs to be grouped as column families within a row. Timestamps are used to version the value of each column key.

Each row is a distinct unit of data and they are used to group sets of column families. An individual row is indexed by a unique row key, which serves a couple of important purposes. Firstly, BigTable keeps all of its data sorted by each row's key. This is important for how BigTable optimizes certain operations against its persistent backing store. Secondly, BigTable performs dynamic partitioning based on row key ranges. Row keys can also be used to provide a hint to client applications about the keys and values that it contains. BigTable ensures that each read or write under a single row key is treated as a transaction and only a single row key can be involved in a transaction. It does not support transactions involving multiple row keys.

Column families are used to group sets of column keys and are the basic unit of access control (Chang and Dean pg.2). Column families must be created before any column keys can be associated with them. However, after a column family is created, any number of columns can be associated with it. By convention, all data within a column family is usually of the same type. Column families provide the lowest level namespace within BigTable and are the basis for all disk and memory accounting.

BigTable uses data versioning to support concurrent access and each key can contain multiple versions of its values. Client applications can choose to let BigTable assign the timestamp values, or it can assign them itself. Unless otherwise specified, the latest timestamp is thought of as the latest version of the data. Values are stored in decreasing order according to their timestamps, so the newest version is always at the top. BigTable allows for the client to keep a specified number of versions for the same key, or ones that have been created since a certain point in time.

An API is provided for programmatic modification and querying of data within BigTable. As is the case with most column-oriented datastores, BigTable does not provide a query or data

modification language. The BigTable API also includes functions for client applications to create and delete tables and column-families and to change access control rights (Chang and Dean pg.3). Because of this open-ended schema design, much of the actual schema is contained within client applications.

The main external components that BigTable relies upon are a distributed file system called Google File System (GFS), a cluster management system, and a distributed lock manager called Chubby. These components are not BigTable specific, and are in fact totally separate items that are used by other non-BigTable related services at Google. BigTable is based on a shared-nothing architecture and relies upon the Google File System for persisting log and data files (Chang and Dean pg.3). Using a distributed file system avoids the cost and complexity of using a shared-disk system, but still provides a reliable storage system for each node to access.

The cluster management system in which BigTable depends upon helps to maintain operations within a cluster of BigTable nodes. Its duties include scheduling jobs, managing resources on shared machines, monitoring machine status, and adding or removing machines from the cluster (Chang and Dean pg.3). BigTable supports adding additional servers to the cluster as needs change and it is the cluster management system that coordinates this effort. The use of a separate cluster management system is different from other datastores which support the ability to manage and coordinate cluster resources on their own. Keeping these responsibilities separate from the BigTable system limits complexity and provides a reusable service for other Google applications.

An entire BigTable cluster will hold a number of tables, which are collections of distinct column families. Portions of the row key range of a table are broken out into an item called a tablet. A tablet is equivalent to a partition within a relational database. The BigTable master

tablet server will handle distributing the storage responsibilities of tablets to various tablet servers in the cluster. It is the master tablet server's responsibility to keep track of which servers are responsible for which tablets. To accomplish this, the master tablet server uses a special metadata table to keep track of the location of all tablets (Chang and Dean pg.4).

Each tablet server is responsible for a set of tablets. Responsibilities include handling read and write operations and splitting tablets once they become too large. The tablet server will communicate with the Chubby lock service to ensure that a client has the appropriate access to perform the requested operation. Additionally, when a write operation occurs against a tablet, the tablet server will record the change in a commit log. This enables the durable storage of data. Tablet servers use GFS in order to store each backing file.

To provide persistent storage of tablets, BigTable uses a file construct called an SSTable. Each tablet is broken up into 100 to 200 megabyte SSTable files, which are organized as a series of blocks and a block index (Chang and Dean pg.4). The block index is used to map blocks to their location on disk which helps reduce the number of disk accesses required to find a block on disk. The SSTable itself is a map of key-value pairs and is kept sorted by keys. To minimize the overhead of keeping the keys sorted, all writes are first buffered in memory, sorted, and later written to disk once it reaches a certain threshold of size.

To speed up lookups into SSTables that are not stored in memory, BigTable uses a technique called a Bloom filter to help minimize I/O (Chang and Dean pg. 7). The Bloom filter is stored externally from the SSTable and is essentially a fixed length bitmap, which can be read to determine whether or not the row and key pair exist in the SSTable. This keeps BigTable from having to scan the entire SSTable when searching for a row and key pair; BigTable can search the Bloom filter structure instead.

Chubby, the distributed locking service used by BigTable, provides a number of important services to a BigTable cluster. Some of Chubby's responsibilities include the storage of access control lists and schema information, discovering the addition and removal of tablet servers within the cluster, and handling client sessions within the cluster. The handling of Chubby sessions plays a very important role within BigTable. All client applications and tablet servers are required to register a session with Chubby prior to accessing anything else within the system.

BigTable is a proven solution, having been in production use since April of 2005 (Chang and Dean pg.13). According to Google, as of 2006 there were 388 non-test BigTable clusters which consist of a combined total of 24,500 tablet servers (Chang and Dean pg.10). It borrows from many different kinds of systems, including parallel database systems and distributed file systems. BigTable is a pioneer amongst distributed non-relational datastores and has served as the model for several other popular distributed datastores, including Facebook's Cassandra.

4.2 Amazon’s Dynamo

Amazon's business is very centered on the user experience. Quick response times, reliability, and high availability are required properties in order to maintain a high quality of service. They are one of the largest e-commerce operations in the world, which services millions of users all over the world (DeCandia and Hastorun pg.205). Every second during an outage costs them significant amounts of money. As a result, their applications require a highly available and responsive datastore for storing application state and data. Based on their needs, Amazon decided to implement Dynamo, a non-relational key-value datastore that is highly available, totally decentralized and is always writable.

One interesting aspect of Dynamo that is different from other systems is that it has no security. There is no support for authentication or authorization. Dynamo expects to run in a totally trusted environment (DeCandia and Hastorun pg.208). This highlights the fact that, unlike other systems, Dynamo is a specialized solution that is designed to meet the specific needs of certain classes of applications (ones that favor availability over consistency). Such an environment would have to be designed with great care and would always carry a great deal of risk.

The core components of Amazon's Dynamo are a flat key-value pair data model, eventually consistent replicated storage, and dynamic partitioning using consistent hashing. It achieves its totally decentralized architecture by using a gossip-based cluster membership and failure detection protocol. Dynamo's data model is designed for applications that can utilize a primary-key only access. Examples of such applications at Amazon are shopping carts, session management, and the product catalog (DeCandia and Hastorun pg.205).

Dynamo is a totally unstructured datastore. The schema is entirely procedural and client applications must handle all aspects of schema maintenance. Client applications must use the Dynamo API in order to access and modify data. To lookup data, the client can use the get(key) function (DeCandia and Hastorun pg.209). Keys must be unique within the global Dynamo namespace. The node in which the client application directed the request may or may not be responsible for the storage of the requested key. Therefore, each Dynamo node is capable of transparently routing the client request to the appropriate node. Because of this functionality, a Dynamo cluster resembles a computer network, where routers and switches work together to deliver packets from point A to point B.

To insert or update data, the client will use the put(key, context, object) function (DeCandia and Hastorun pg.209). The context of a put contains some Dynamo specific data and the object is the actual value. Values in Dynamo are handled as uninterpreted byte arrays, so the client application must maintain the integrity of values. In a similar fashion to Dynamo's handling of read requests, each node is capable of routing the write request to the appropriate node or nodes. This functionality improves the overall availability of a Dynamo cluster in that any application can direct any request to any node.

The downside to this is that a request that involves multiple hops will have a higher latency than a request directed at the appropriate node. Each node must also maintain a certain amount of routing information to correctly route requests to other nodes. Dynamo's solution to this problem is to include a client-side library that periodically downloads information about each node so that the client can direct requests to the appropriate node. This is a common theme with Dynamo, where a fair amount of complexity is included in the design of client applications with the benefit of increased performance and availability. Because of this, there is not always a clear distinction between client applications and storage nodes in a Dynamo cluster.

Dynamo is a classic example of a distributed datastore that was designed with the limitations presented by the CAP Theorem in mind. Applications which use Dynamo as their back-end cannot afford to reject updates for the sake of maintaining strong consistency. Dynamo uses an optimistic replication strategy where each replica node will eventually become consistent. Eventual consistency reduces the latency of writes and allows Dynamo to maintain operations even in the face of network and member node failures. As discussed earlier, this capability comes at a price.

Two critical components of Dynamo's eventual consistency implementation are data versioning and syntactic and semantic reconciliation techniques. Dynamo treats each modification as a new version of the data and it is possible for multiple versions of a data item to exist at the same time. To help cope with this, Dynamo creates a vector clock for every version of every object (DeCandia and Hastorun pg.210). Vector clocks can enable Dynamo to perform syntactic reconciliation without having to involve the client application. However, if multiple branches of the data exist, syntactic reconciliation may not be possible. This situation is possible due to the eventually consistent replication model combined with one or more faults in the system. In this case, it is up to the client to include logic to perform a semantic reconciliation of the various versions of the data.

Dynamo uses a consistent hashing mechanism to distribute the storage responsibilities amongst nodes in the cluster. Unlike the vanilla consistent hashing technique, Dynamo assigns multiple regions of the hash space to nodes based on their capacity (DeCandia and Hastorun pg.210). This allows Dynamo to take advantage of the differences in the capabilities amongst nodes and more efficiently balance the load within the cluster. Dynamo can also use information about the amount of load currently being handled by a particular group of nodes to determine how best to allocate additional resources. This requires nodes to be able to share information about themselves with other nodes in the cluster.

Aiding in Dynamo's goal of incremental scalability is a gossip-based membership protocol. Changes in cluster membership will occur. Dynamo requires a system administrator to perform membership maintenance. However, seed nodes, which know about all nodes in the system, share their information with other nodes in the cluster, which in turn share their membership information with peer nodes. Over time, membership information eventually

propagates throughout the entire cluster. This process is very similar to how Dynamo pushes data updates out to replica nodes (DeCandia and Hastorun pg.212).

Dynamo is used by a number of critical Amazon services which have “strict operational requirements” (DeCandia and Hastorun pg.205). At every point in its design, Dynamo focuses on avoiding dependencies between nodes as much as possible. The trade-off is that client applications are tightly coupled to Dynamo and require a great deal of additional logic to cope with eventual consistency and procedural schema maintenance. The payoff is a system that has no single point of failure and is capable of scaling to handle the needs of one of the largest e-commerce systems in the world.

4.3 Facebook’s Cassandra

Facebook is one of the most popular social networking websites in the world; millions of people use Facebook every day to connect with their friends. Facebook's applications require a datastore that can manage billions of writes per day. To meet these requirements, Facebook engineers have designed a distributed datastore that combines Google BigTable's structured data model with the eventually consistent, decentralized storage model of Amazon's Dynamo. Like other popular column-oriented systems, Cassandra was also designed to achieve incremental scalability using commodity hardware.

Cassandra is a fusion of ideas from both BigTable and Dynamo. However, unlike BigTable and Dynamo, Cassandra is entirely open source. It borrows the data and durable storage models of BigTable (SSTables and Bloom filters) and the eventual consistency model from Dynamo. Unlike BigTable, Cassandra does not depend on a distributed file system, nor does it require a cluster management system. Cassandra also differs from Dynamo in an important way, in that Cassandra does not require a trusted environment. It does provide some

level of security with authentication and authorization methods, overcoming a critical downside in the design of Dynamo.

Client access for reading and writing data to Cassandra is performed through its Thrift API (Featherston pg.8). The underlying communication protocol uses Remote Method Invocation, which is commonly referred to as RMI. Schema design and maintenance is again entirely programmatic. The Thrift API does not appear to be as mature as BigTable and Dynamo's respective API's. Consequently, designing client applications that interact with Cassandra can be quite challenging (Featherston pg.8).

Cassandra has a novel solution to handling the consistency requirements for client applications. It allows the client to specify its required level of consistency for a particular operation. Supported levels are zero, one, quorum, all, and any (Featherston pg.6). The zero and all specifications are only available for writes. This allows for a much more flexible client-side implementation, where developers can limit the complexity of their applications by requiring higher levels of consistency if needed. Cassandra's eventual consistency model does open up the opportunity for multiple versions of a data item. It resolves this by using a technique called read-repair (Featherston pg.7). This means that if there are any conflicts in versions found during the processing of a read request, Cassandra will return the most recent copy.

Of the column-oriented datastores that were covered, Cassandra is the least mature, but offers the most diverse set of functionality. It was recently approved as a top-level Apache Foundation project in March of 2010 (Featherston pg.1). It is unique in that it allows client applications to specify their consistency requirements and is also capable of performing replication based on a rack aware and rack unaware strategy. It is designed to have no limit to its

scalability and given some time to mature, as BigTable and Dynamo have, it will likely turn into an even more robust system.

5 Motivating Example

In order to help illustrate the core differences between relational databases and column-oriented datastores, an example use case will be discussed. The use case covers a solution for storing and analyzing rsyslog formatted log files. The motivation for the storage of log files is to allow system administrators to analyze the files for trends and anomalies. Regular log analysis is a best practice for system administrators. Tools which aid in the parsing and loading of log files into a relational database do exist. However, there do not appear to be any widely available tools for loading log file data into a column-oriented datastore.

rsyslog is a logging daemon for Unix-like systems. It implements and extends the syslog protocol and provides several logging facilities and priorities. In addition to providing the basic functionality for application logging, rsyslog can act as a centralized log server, supports encrypted communications, and a robust set of log formatting capabilities. rsyslog's formatting capabilities are centered on a set of log message properties (The Property Replacer). When rsyslog prepares to output a particular message, it will use the specified properties to format the message, thus replacing each property with a value. In the opinion of the author, this behavior is lends itself well to a key-value pair data model.

The requirements for our use case include the ability to provide structured storage for log messages from multiple hosts. For the sake of limiting our scope, we'll focus on handling IPtables firewall messages that are sent to rsyslog from various local hosts. We will assume that we have configured rsyslog to appropriately filter messages that are prefixed with “iptables:” and we will be making use of the following rsyslog message properties: msg, fromhost-ip, and

timestamp. We'll also be interested in the formatted IPtables message, which includes other key-value pair data (ex: SRC=<IP address>, DPT=<port number>, etc.). The vast majority of our data will be character strings.

The main goal of using a database to store these log messages will be to identify trends in dropped or rejected packets. This will help us to identify potentially malicious hosts so that we can deal with them appropriately. It will also help us to analyze usage patterns of the various clients that use our services. In short, we'll use this operational data to help us better understand the state of our firewalls.

5.1 Relational Data Model Implementation

There are two choices to be made in the relational data model design. One choice would include the following entities: remote_host and log_message. We know that each log message will include the msg, fromhost-ip and timestamp properties and each of our hosts will have a host_ip, hostname, and description property. The host entity will have a one-to-many relationship with log_message. The result will be two tables: one for remote_host and one for log_message. The primary-key of the remote_host table will be the host_ip. The primary-key of the log_message table will be a composite of host_ip and a timestamp for the message. We assume that log messages will be serialized on the host, so each timestamp will be unique for each host. A diagram of each data model is included in the Appendix.

This design assumes that the format of our log messages will remain static, which isn't entirely accurate for IPtables messages. Each protocol (icmp, udp, tcp, etc) includes a handful of different keys in their messages. We've included each rsyslog property and IPtables message key as a column in the log_message table. The alternative design could allow for flexibility by introducing additional entities: rsyslog_property and iptables_key. With this design,

remote_hosts would have a one-to-many relationship with the log_message entity. The rsyslog_property and iptables_key entities would both have a one-to-many relationship with the log_message.

This solution provides for more flexibility, however, it will result in a log_message table whose row count will be the number of keys times the number of log messages. A lightly loaded server may produce 5,000 messages in a day. This would result in 90,000 rows for one host in a single day (assuming we have 18 message fields). The size of this table would quickly grow to the point that it would become difficult to maintain. Each log message insert would incur some overhead since an insert for each key would need to be performed. Another primary downside to this method is that we now have to assume that each rsyslog_property value and iptables_key value are of the same length, which makes this design rather inefficient. Ideally, we would have a design that would allow for a flexible handling of properties and keys, but remain performant and avoid a table design that would grow rapidly in row count.

5.2 Column-oriented Data Model Implementation

The column-oriented data model lends itself quite well to handling this sort of semi-structured data. Utilizing Cassandra's data model will provide a nested column-family structure for us to store our data. One choice for the logical data model would be to create a column family for each host, keyed by the host's IP address for their primary network interface. Each host would need a column key for hostname, a description of that host, and the MAC address for the primary physical network interface. This column family yield a list of the distinct hosts for which log messages will be stored.

Another column family could be used to store each log message for each host. Our key for this column family will be the concatenation of host IP address and the timestamp on the log

message. In this case, there is really no use for versioning each of the log messages, so a new instance of this column family will be created for each log message. For each message, we'll have column keys for every part that that will be parsed out of the original rsyslog formatted message. One key for each rsyslog property will be used along with a key for each part of the IPtables message.

Each protocol will have a slightly different list of keys. Cassandra supports this problem nicely by allowing us to add any number of column keys to a column family. This is in contrast with a relational data model where we would have to declare all possible message fields before hand, but only a subset of fields would be needed for each message. Given all the above requirements, the physical data model would be implemented as follows:

```
{
      "host_ip": {
            HOSTNAME:
            DESCRIPTION:
            MAC:
      }
}
{
      "host_ip"+"timestamp" {
            MSG:
            FROMHOST-IP:
            IN:
            OUT:
            MAC:
            SRC:
            DST:
            LEN:
            TOS:
            PREC:
            TTL:
            ID:
            PROTO:
            SPT:
            DPT:
            WINDOW:
            RES:
            FLAG:
      }
}
```

5.3 Example Input Data

```
Apr 23 18:30:50 zachhost kernel: 'iptables: ' IN=eth0 OUT=
MAC=00:00:00:00:00:00 SRC= 192.168.1.8 DST=192.168.1.2 LEN=200 TOS=0x00
PREC=0x00 TTL=54 ID=3000 PROTO=UDP SPT=53 DPT=31232

Apr 23 18:30:51 webserv1 kernel: 'iptables: ' 'IN=eth0 OUT= MAC=
00:00:00:00:00:00 SRC=192.168.1.8 DST=192.168.1.21 LEN=40 TOS=0x00 PREC=0x00
TTL=32 ID=3344 DF PROTO=TCP SPT=80 DPT=8972 WINDOW=65535 RES=0x00 SYN

Apr 23 18:32:08 webserv2 kernel: 'iptables: ' 'IN=eth0 OUT= MAC=
00:00:00:00:00:00 SRC= 192.168.1.8 DST=192.168.1.23 LEN=40 TOS=0x00 PREC=0x00
TTL=32 ID=561 DF PROTO=TCP SPT=8080 DPT=30215 WINDOW=65535 RES=0x00 SYN

Apr 23 18:33:12 webserv1 kernel: 'iptables: ' 'IN=eth1 OUT= MAC=
00:00:00:00:00:00 SRC= 192.168.1.8 DST=192.168.1.22 LEN=40 TOS=0x00 PREC=0x00
TTL=32 ID=3002 DF PROTO=TCP SPT=80 DPT=9651 WINDOW=65535 RES=0x00 SYN
```

5.4 Example Persisted Data

```
{
      192.168.1.2 : {
            HOSTNAME: zachhost
            DESCRIPTION: test server
            MAC: 00:00:00:00:00:00
      }
}
{
      192.168.1.23 : {
            HOSTNAME: webserv2
            DESCRIPTION: web application server
            MAC: 00:00:00:00:00:00
      }
}
{
      192.168.1.21 : {
            HOSTNAME: webserv1
            DESCRIPTION: web application server
            MAC: 00:00:00:00:00:00
      }
}
{
      192.168.1.2_Apr-23-18:33:12 {
            MSG:'iptables: ' IN=eth0 OUT= MAC=00:00:00:00:00:00 SRC=
192.168.1.8 DST=192.168.1.2 LEN=200 TOS=0x00 PREC=0x00 TTL=54 ID=3000
PROTO=UDP SPT=53 DPT=31232
            FROMHOST-IP: 192.168.1.2
            IN: eth0
            OUT:
            MAC: 00:00:00:00:00:00
            SRC: 192.168.1.8
            DST: 192.168.1.2
            LEN:200
            TOS:0x00
            PREC:0x00
            TTL:54
            ID:3000
```

```
            PROTO:UDP
            SPT:53
            DPT:31232
      }
}
{
      192.168.1.21_Apr-23-18:30:51 {
            MSG:'iptables: ' 'IN=eth0 OUT= MAC= 00:00:00:00:00:00
SRC=192.168.1.8 DST=192.168.1.21 LEN=40 TOS=0x00 PREC=0x00 TTL=32 ID=3344
DF PROTO=TCP SPT=80 DPT=8972 WINDOW=65535 RES=0x00 SYN
            FROMHOST-IP: 192.168.1.21
            IN: eth0
            OUT:
            MAC: 00:00:00:00:00:00
            SRC: 192.168.1.8
            DST: 192.168.1.21
            LEN: 40
            TOS: 0x00
            PREC: 0x00
            TTL: 32
            ID: 3344
            PROTO: TCP
            SPT: 80
            DPT: 8972
            WINDOW: 65535
            RES: 0x00
            FLAG: SYN
      }
}
{
      192.168.1.23_Apr-23-18:32:08 {
            MSG:'iptables: ' 'IN=eth0 OUT= MAC= 00:00:00:00:00:00 SRC=
192.168.1.8 DST=192.168.1.23 LEN=40 TOS=0x00 PREC=0x00 TTL=32 ID=561 DF
PROTO=TCP SPT=8080 DPT=30215 WINDOW=65535 RES=0x00 SYN
            FROMHOST-IP: 192.168.1.23
            IN: eth0
            OUT:
            MAC: 00:00:00:00:00:00
            SRC: 192.168.1.8
            DST: 192.168.1.23
            LEN: 40
            TOS: 0x00
            PREC: 0x00
            TTL: 32
            ID: 561
            PROTO: TCP
            SPT: 8080
            DPT: 30215
            WINDOW: 65535
            RES: 0x00
            FLAG: SYN
      }
}
{
      192.168.1.21_Apr-23-18:33:12 {
            MSG:'iptables: ' 'IN=eth0 OUT= MAC= 00:00:00:00:00:00 SRC=
192.168.1.8 DST=192.168.1.21 LEN=40 TOS=0x00 PREC=0x00 TTL=32 ID=3002 DF
```

```
PROTO=TCP SPT=80 DPT=9651 WINDOW=65535 RES=0x00 SYN
            FROMHOST-IP: 192.168.1.22
            IN: eth0
            OUT:
            MAC: 00:00:00:00:00:00
            SRC: 192.168.1.8
            DST: 192.168.1.22
            LEN: 40
            TOS: 0x00
            PREC: 0x00
            TTL: 32
            ID: 3002
            PROTO: TCP
            SPT: 80
            DPT: 9651
            WINDOW: 65535
            RES: 0x00
            FLAG: SYN
      }
}
```

5.5 Summary of the Example Solution

It is likely that a medium sized enterprise would generate hundreds of millions of log messages every day. Two primary requirements of a log database for such an enterprise will be: the ability to sustain a high write throughput and the ability to efficiently query the large volume of log data generated by servers. Relational data model 1 provides no flexibility and is not feasible for handling diverse log file formats. Relational data model 2 provides a high degree of flexibility, however, table size will grow rapidly.

To store each new log message in relational data model 2, we will need to perform an insert for each key-value pair. Requirement 1 says that we will want limit overhead for writes to keep throughput high, therefore, we will want to avoid any integrity constraints and indexes. However, requirement 2 says that we will need to be able to efficiently query the data. To query combinations of keys, we will have to join the log_message table onto itself for each additional key. Worse yet, without indexes, we will have to scan the entire table to perform joins and queries. As a result of our two main requirements, we have a Catch-22 with the relational solution.

The column-oriented solution achieves a high write throughput by only needing to write to a commit log and an in-memory buffer initially. The contents of an SSTable are not written to disk until the in-memory structure reaches a certain size. The commit log is also buffered and periodically flushed to disk to keep I/O as sequential as possible. The data model for this solution will only require one insert for each log message, yet will still give us the flexibility to only include the keys that were used; another benefit over relational model 2.

Querying with the column-oriented solution is also a strength. First of all, there will be no need to self-join tables to get combinations of keys within log messages. Secondly, we can use Bloom filters to only read blocks that contain the requested keys, which will help limit the number of I/O's. Lastly, our datastore supports dynamic partitioning which will further speed up queries given that the load for reads (and writes for that matter) can be distributed across multiple servers and executed in parallel.

6 Conclusion

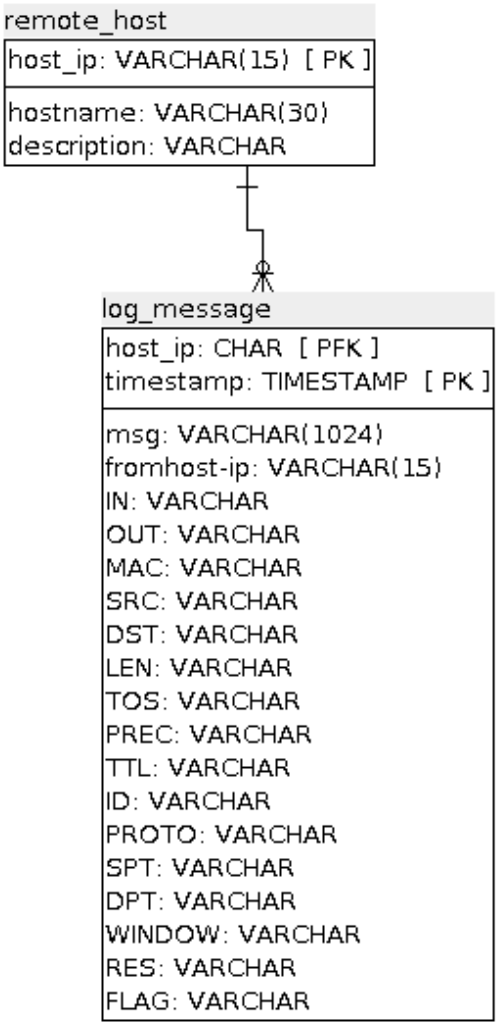
Solving problems at the scale of the Internet necessitates another level of engineering that most sites do not have to consider. Providing structured data storage for massively distributed systems requires a new way of thinking. Regardless of the specific technology that's being used, concepts such as asynchronous replication, dynamic partitioning and incremental scalability are perquisites for meeting the demands of Internet-scale applications. There are both relational and non-relational databases that will provide the previously listed scale-out functionality. The main consideration then becomes whether or not the requirements of your application require strict transactional processing and strong consistency.

Datastores such as Cassandra, Dynamo, and BigTable were built with specific problems in mind. Where relational databases are designed to provide a general purpose database, these

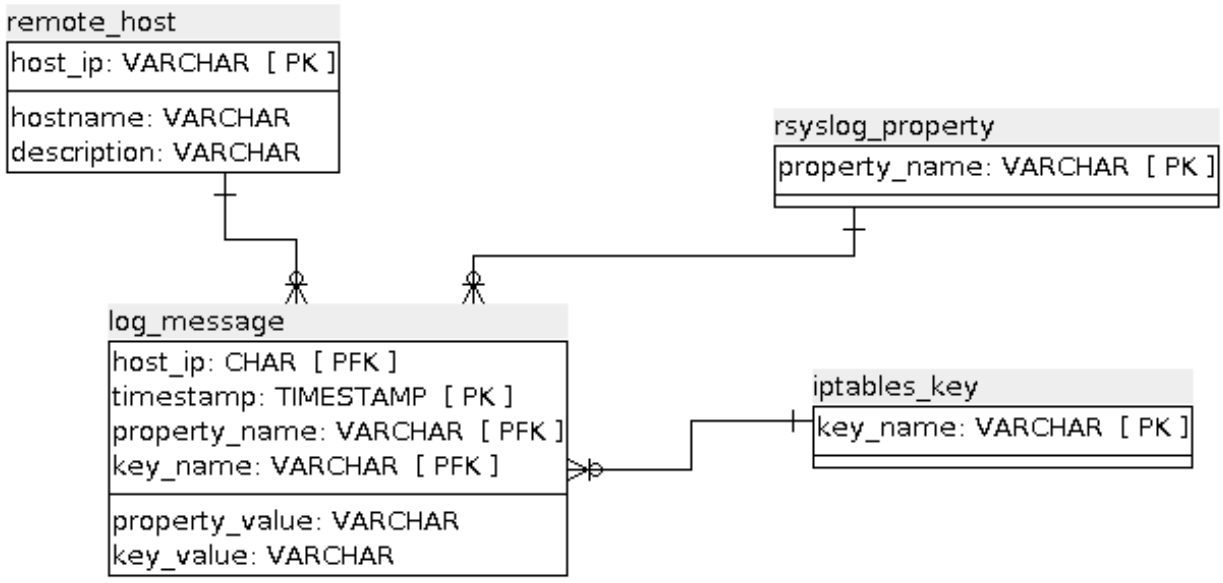
datastores were designed to meet the specialized needs of applications. It is important to keep that perspective when studying alternatives to something as mature as the relational database model. In practice, these sites are using a combination of relational databases and column-oriented datastores to solve their needs. Ultimately, it's about having the right frame of reference and understanding to be able to choose the right tool for the job.

7 Appendix

Relational Data Model 1



Relational Data Model 2



Works Cited

"ACID." *Wikipedia, the Free Encyclopedia*. Web. 10 Apr. 2011. <http://en.wikipedia.org/wiki/ACID>.

Chang, Fay, 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." *OSDI'06: Seventh Symposium on Operating System Design and Implementation*. Seattle, WA. Google, Inc., Nov. 2006. Web. 25 Jan. 2011. <http://labs.google.com/papers/bigtable.html>.

Codd, E. F. *The Relational Model for Database Management: Version 2*. Boston, MA: Addison-Wesley, 1990. *ACM Digital Library*. Association for Computing Machinery. Web. 10 Apr. 2011. <http://0-portal.acm.org.millennium.lib.cyut.edu.tw/citation.cfm?id=77708>.

"Column-oriented DBMS." *Wikipedia, the Free Encyclopedia*. Web. 7 Feb. 2011. <http://en.wikipedia.org/wiki/Column-oriented_DBMS>.

DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's Highly Available Key-value Store." Proc. of SOSP '07 Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, Stevenson, WA. Amazon.com, 2007. Web. 7 Mar. 2011. <http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf>.

Eaton, Chris. "DB2 PureScale Scalability - Part 3." Web log post. *An Expert's Guide to DB2 Technology*. Toolbox for IT, 8 Apr. 2011. Web. 11 Apr. 2011. <http://it.toolbox.com/blogs/db2luw/db2-purescale-scalability-part-3-45443>.

Featherston, Dietrich. "Cassandra: Principles and Application." *D2fn.com*. 3 Aug. 2010. Web. 25 Jan. 2011. <http://d2fn.com/2010/08/03/cassandra-paper.html>.

Gerhards, Rainer. "Syslog Parsing in Rsyslog." *Rsyslog.com*. 23 Sept. 2008. Web. 25 Apr. 2011. <http://www.rsyslog.com/doc/syslog_parsing.html>.

Gilbert, Seth, and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." *ACM SIGACT News* 33.2 (2002): 51-59. Web. 19 Feb. 2011. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf>.

Lakshman, Avinash, and Prashant Malik. "Cassandra: a Decentralized Structured Storage System." *ACM SIGOPS Operating Systems Review* 44 (Apr. 2010): 35-40. *CiteSeerX*. Facebook, Apr. 2010. Web. 2 Mar. 2011. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.161.6751&rep=rep1&type=pdf>.

"MySQL :: MySQL 5.0 Reference Manual." *MySQL :: Developer Zone*. Web. 12 Apr. 2011. <http://dev.mysql.com/doc/refman/5.0/en/index.html>.

Orenstein, Gary. "What the Heck Are You Actually Using NoSQL for?" *High Scalability - High Scalability*. 6 Dec. 2010. Web. 05 Apr. 2011. <http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html>.

Pritchett, Dan. "BASE: An Acid Alternative." Web log post. *ACM Queue*. Association for Computing Machinery, 28 July 2008. Web. 17 Apr. 2011. <http://queue.acm.org/detail.cfm?id=1394128>.

"Relational Model." *Wikipedia, the Free Encyclopedia*. Web. 05 Apr. 2011. <http://en.wikipedia.org/wiki/Relational_model>.

Stonebraker, Michael. "The "NoSQL" Discussion Has Nothing to Do With SQL | Blog@CACM." *Communications of the ACM*. 4 Nov. 2009. Web. 9 Feb. 2011. <http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext>.

"The Property Replacer." *Rsyslog.com*. Web. <http://www.rsyslog.com/doc/property_replacer.html>.

Yao, Jingguo. "Cassandra Architectural Overview." *Cassandra Wiki*. Apache.org, 27 Dec. 2010. Web. 27 Apr. 2011. <http://wiki.apache.org/cassandra/ArchitectureOverview>.