Parker: Storing the Social Graph

Jonathan Kupferman
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California 93106
jkupferman@cs.ucsb.edu

Kurt Kiefer
Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, California 93106
kekiefer@gmail.com

ABSTRACT
Online social networking applications are becoming commonplace but tend to be very difficult to scale to large amounts of data and many concurrent users. Furthermore, the unfamiliar data model may not be a good fit for traditional relational databases. As web sites grow, they start to demand higher performance and more complex operations on the data. Naive approaches at storing the data quickly degrade the usability of a system, and custom solutions become necessary. We have designed Parker, a scalable distributed storage system designed specifically for social networking applications so as to address this concern. A specialized API makes application programming easy and less prone to error. Optimal routing techniques provide high performance and high availability in the face of increasing load as well at machine failures.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed databases*

General Terms
Performance,Reliability

1. INTRODUCTION
Online social networks have become immensely popular in recent years. In 2008, Facebook, YouTube, and MySpace were three of the five most visited web sites in the United States[1]. It is estimated that in early 2009, Facebook received 80 billion unique page views per month, an average of over 30,000 per second. Even if only a fraction of these page views involve a query to a database, handling such a large number of requests requires a high performance storage system.

While social networks are not the only websites dealing with large amounts of traffic, they are unique in the type of data that they host. Social networks are generally thought of as a graph structure with users as nodes and relationships as the links between them, hence the name "the social graph." While a useful model, graph structures are well known to be difficult to store and query in an efficient manner.

Making things more difficult is the fact that the graph structure needs to be queried in real time. Unlike other graph storage systems, minimal latency is crucial since many websites enforce stringent response time requirements.

Fortunately, the data model of the typical online social network has a number of exploitable properties. Users are individual points of interest within the social graph, and lookups on these points of data make up the bulk of web queries. The data model is straightforward: users maintain profiles with some information about themselves and also maintain a set of friends. These types of elements fit nicely into certain non-traditional database models. Relaxed consistency for the common read is also a convenient property. If a user updates his profile, others should eventually see the updates, but these changes do not have to be seen immediately.

This paper presents Parker, a highly scalable, highly available, persistent storage system specifically for applications running online social networks. It is designed to handle very large amounts of traffic and data through its unique system architecture and exploitation of the essential properties of social networking data.

2. REQUIREMENTS
API The system should provide a simple API which allows users (application developers) to perform many of the most common operations required by online social networks. Aside from simple query methods, there must also be higher level operations provided for the user. The API should also be extensible to allow new and more complex queries to be implemented.

Data model Since social networks vary widely in type of data they store per user, it should be simple to modify the data model such that fields can be added or removed as necessary. The system should also support versioning such that changes can be made to the data model without requiring the entire system to be redeployed.

Performance The system should be able to handle hundreds of requests per second in a datacenter-like environment. It should also be able to service incoming

requests from many distinct machines simultaneously.

Scalability As the amount of data or requests increases, the addition of nodes to the system should allow the system to maintain if not surpass its current performance. Simlarly, as any piece of the system becomes a bottleneck, the addition of servers with the same role should alleviate the issue.

Fault-tolerance As the number of machines in a distributed system increases so does the chance that one of these machines will fail. Thus, the system should be able to handle these occurrences as faults and continue functioning even in the face of multiple machine failures.

3. RELATED WORK

The work presented here stems from a large body of research done by both the distributed system and database communities. Peer-to-peer distributed systems like Chord[16], Tapestry[21], and Pastry[13], provide intelligent mechanisms for doing look-ups in large-scale distributed systems. By maintaining a small amount information about ones "neighbors" in the system, requests can be routed to the destination within O(log n) hops. Since these systems are designed for P2P use, it is assumed that there will be large number of failures and a high churn rate among nodes in the network.

Beehive[12] improves on the above systems by providing an average-case O(1) lookup performance. Since many query patterns have a long-tailed or Zipfian distribution, the system proactively replicates data so that the average lookup can be completed in constant time and the worst-case in O(log n) time.

While the previously mentioned systems could be used to store graph data in the same key-value manner that Parker does, there is a significant performance penalty inherent in their P2P designs. As discussed in section 6.2 there is a significant overhead cost associated with performing RPC that would be greatly magnified as a result of even the average case O(log n) hops required per lookup. Thus even in a data center environment, a noticeable increase in latency would make such a system impractical for use with real-time web applications. While Beehive could provide an improvement, it is not clear that queries for online social networks follow a Zipfian distribution.

Systems like Amazon's Dynamo[6], Google's BigTable[4], and Yahoo's PNUTS[5] all provide high-performance, scalable storage systems, each with a different query model. Unlike the previously discussed P2P systems, these systems were designed for a data center-like environment where nodes join or depart the network very infrequently, nodes can communicate with low latency, and there are no malicious nodes. Unfortunately, these systems are all proprietary and are not available for public use. While it would be feasible to build or leverage an open-source system modeled after any of these systems, there are additional design considerations which need to be taken into account.

Similar to Parker, Dynamo provides a highly scalable distributed key-value storage system. While one could store a graph structure in such a system, there are some considerable drawbacks to this approach. First, since Dynamo does not have any semantic information about the data it stores, it cannot provide some essential functionality required for this type of data. One such example is support for transactions. In order to ensure that updates are not lost in the case of concurrent read-modify-write operations, one must ensure that each operation occurs as a transaction. While such functionality could be implemented on top of Dynamo, it would come with a performance penalty since achieving high performance in concurrent settings requires minimizing the size and time of transactions.

BigTable provides a more structured data format where data is divided into tables, to which columns can be dynamically added. As evidenced by its use for storing web documents, BigTable could likely be used to efficiently store a graph structure like that of social networks. While BigTable is designed for use both in offline data processing and real-time applications, we have found that our implementation is able to greatly outperform BigTable in terms of request rate. This is discussed in more detail in section 6.4.

Graph databases have been a popular area of research dating back to the early 90's, particularly for graph mining and biological studies[7, 19]. In both of these areas, it is often the case that one has a large static data set and the goal is to perform queries on the graph structure in an efficient manner. These databases are not typically designed to perform the low latency, real-time operations. Instead, these systems focus more on throughput and memory efficiency, and as a result, do not provide a satisfactory solution to the problem of storing and retrieving data for online social networking web applications.

4. ARCHITECTURE

Parker is designed as a two-tier horizontally scalable system, as shown in Figure 1. The lower layer is comprised of data servers whose responsibility it is to access and modify a local database. Above the data servers resides a layer of key servers. It is the task of the key servers to maintain the meta-information about the system like which data servers contain what pieces of data. In the simplest incarnation of the system, a single key server routes all user requests to any of three replicated data servers, each running a fast, local key-value store. The roles of key servers and data servers will be discussed in more detail in the following sections.

The system is designed to act transparently as a centralized, shared data storage resource for social networking data. It is primarily designed for use by web applications, where multiple application servers simultaneously access shared data to generate page content for users. In such a typical system, these application servers may exist on a single physical machine, or on many machines, and the machines may be located within one or many data centers. However, Parker can be used in other contexts where one needs high performance access to a very large graph structure.

Access to the social networking system by application servers is provided through a cross-platform API that implements queries specialized for social networking applications. The provided API can be accessed via C++, Java, Python, PHP, Ruby, Perl, and C#, as well as a number of other languages. The interface can be easily extended, providing additional

functionality beyond the original implementation – a typical scenario as an application's requirements evolve over time.

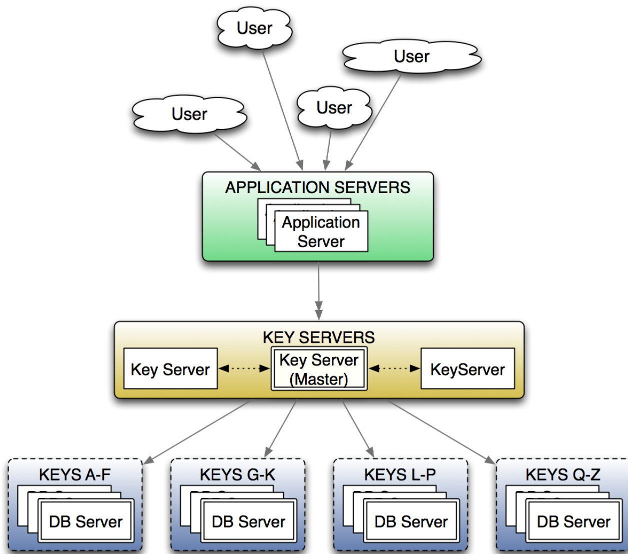

Figure 1: Overview of System Architecture

4.1 Data Servers

Groups of data servers function as a highly available and fault-tolerant replication group, with each data server maintaining a replica of the key space for which it is responsible. Within the set of data servers, a single server acts as a replication master in order to handle write requests. Any of the servers in the group can handle read requests. The model provides a selective eventual consistency model, as updates are not guaranteed to be immediately propagated to non-master data servers in the group. In other words, stale reads can occur on a replica which has not been informed of an update on the piece of data. However, the API does support the functionality to perform reads that are guaranteed to be consistent.

As the amount of data or load on the data servers increases, more servers may be required to handle the load. At any time, a new group of data servers can be started and dynamically assigned some portion of the data set.

4.2 Key Servers

Key servers are responsible for maintaining the meta-information about the system, in particular the list data servers and the mapping between users and these servers. When a request is received by a key server it first must determine which data server group is responsible for a particular user, and then route that request to an appropriate data server. If request routing by the key servers becomes a bottleneck, additional key servers can also be added to the system. All key servers use a fault tolerant group communication protocol to ensure that information contained on each key server is consistent. Thus, any key server can route a request to the correct data server based on its local information.

In general, key servers are low-overhead processes and can be located anywhere. The system easily supports hundreds of

```
struct User {
    1: i64 uid,
    2: ProfileData profile,
    3: list<FriendData> friends_list
}

struct ProfileData {
    1: string name,
    2: i16 age
}

struct FriendData {
    1: i64 uid
}
```

Figure 2: User Data Model Definition

key servers. Thus, web application servers can run instances of key servers locally which will reduce network latency and improve overall performance of the system. Section 5.3 discusses this optimization in more detail.

5. IMPLEMENTATION

Parker has been developed in C++ over a three month period. The following section will discuss how the system is implemented along with the open-source tools used to create it.

5.1 Data Model

The system implements a very simple and intuitive data model whereby there are many users, each of which is referred to by a unique user ID. All data associated with a user is contained within the user object, including node information (e.g. name) and link information (e.g. friend relationships).

The user data model is defined using the Thrift framework, a part of the Apache Incubator project [14]. Thrift is a multi-language code generator designed for easy cross-platform interoperability. Given a C-like definition file of structures (i.e. objects), Thrift generates the necessary serialization code for the structures to be used interchangably across the desired languages.

Figure 2 is the Thrift definition used to specify a user object that is stored in the database. While simple, the user object captures the most important features required for storing users in a social network. In addition to their user ID, each user object stores a ProfileData object along with a list of FriendData objects. The ProfileData object captures all of the information associated with a particular user such as their name, age, and gender. A FriendData object contains all of the information that a user stores about their friends, in this case simply their friends user IDs. Since social networks vary highly in the type of information they store for a particular user, we provide a minimal User model which can be easily extended to store all of the information required by a particular social network. This can be done by simply updating the structure definition file.

```
get_user(userId, highConsistency)
get_profile(userId,highConsistency)
get_friends_list(userId, offset, limit, highConsistency)

put_user(userId, user)
put_profile(userId, profile)
add_friend(userId, friend)
delete_friend(userId, friend)

get_all_users()
get_random_users(numUsers)
shortest_path(userId, friend, maxHops)
get_neighborhood(userId, mapHops)
```

Figure 3: Sample API Calls

5.2 External API

In Figure 3 we provide a sample of the API calls provided by Parker. In general, the API provides three types of methods: reads, writes, and aggregation methods. The design of the API is based on the most common operations required by an online social network. A noteworthy feature is the optional highConsistency flag provided for read calls. Inspired by Yahoo's PNUTS, this feature allows the programmer to specify if the data must be consistent, or if consistency can be sacrificed for the sake of performance. Among other things, this feature can be used to provide write-then-read consistency (e.g. read your own writes). An additional feature is the inclusion of offset and limit parameters when retrieving friends lists such that common requirements like pagination can be easily and efficiently achieved.

Parker uses Thrift to define the interface to both the data servers and key servers. Thrift code generators create a heterogeneous and high performance call mechanism similar to that of CORBA[18]. For performing RPC, Thrift-generated code creates a user-defined service with a callable stub as well as a server interface.

5.3 Key Server

For most methods, the key servers act as a proxy between the client and data server, ensuring that a particular request is routed to the appropriate data server. This means that in the case of writes, it routes all requests to the master data server of a replication group, and reads are randomly distributed across all data servers in the group. For aggregate methods it is the responsibility of the key server to perform all of the queries necessary to gather the required data and then return the result to the user.

In order to efficiently store the location of users, an in-memory range tree of the possible keys is maintained on each key server. This data structure allows a key server to quickly determine the direct route to the appropriate data server, without any additional guidance from other key servers. Arbitrary ranges of 64-bit integers representing unique user IDs can be stored in the tree, meaning it is possible to support servers for single or small groups of nodes, along with larger groups. A lookup on the data structure has $O(\log n)$ complexity with n as the number of data server replication groups.

Since user IDs tend to be monotonically increasing integer values, mapping these values directly into the key space would result in a heavily skewed distribution of data across the data servers. To prevent this, we use hashing to evenly distribute user IDs over the entire range of the key space.

In order to improve performance, each key server maintains a pool of active TCP connections to data servers to be used when forwarding requests. This allows requests to be immediately forwarded to data servers as opposed to incurring the delay of TCP setup. See section 6.2 for more information about TCP connection overhead in Thrift as measured in Parker.

5.4 Data Server

When a data server receives a request from a key server, the data server has the responsibility of performing the appropriate operation on the underlying storage system. For example, a get_user call would result in the data server performing a query on the database, retrieving the desired user object, and then returning it to the caller. Similarly, an add_friend call would require fetching the user object, updating its friends list, and then writing that user object back to the database.

Storage at Parker's data servers requires a high-performance database supporting features like replication, transactions, and recovery. For this purpose we chose Berkeley DB [11] which was originally developed by Sleepycat Software, and now Oracle. Berkeley DB is a key-value storage system supporting all of the features previously described. Since user objects are serializable and each user has a unique ID, each user can be stored in the database as a (userId,userObject) key-value pair. Berkeley DB also supports sequential record retrieval which is used by aggregation methods like *get_all_users*. While ACID compliant, Berkeley DB is highly configurable such that any or all of these properties can be removed to meet a user's requirements.

All data is stored in Berkeley DB, which is configured to maintain the ACID properties and is replicated to some number of other data servers. This number can be configured based on a user's requirements, and for our purposes we replicate data three times. So as to maintain consistency, all writes must be acknowledged by enough machines to satisfy the quorum. In the case of three machines, the master must receive acknowledgement from at least one replica for a write to succeed. When the master database process or its server fails to operate, an election is held such that a new database process can be named master.

Since Parker provides persistent storage it is important that as new data server groups are assigned a part of the key space, they will receive all data currently held within that key space. In order to achieve this, the source data server must perform a checkpoint of its data and then send it over the wire to the new data server. Once the transfer is complete and the data has been loaded onto the new data server one last check is performed to ensure that any intervening updates are also propagated to the new server. Once up-to-date the data server is considered synchronized and is able to begin servicing requests. In order to minimize the amount of data transferred over the wire, the data is compressed prior

to being sent.

5.5 Maintaining Consistency

The key servers in the social networking system are responsible for tracking changes to the system and routing application requests to the appropriate data servers. Because multiple key servers can exist in the system, it is important that application servers be guaranteed consistently up-to-date and accurate access to the network, regardless of which key server they communicate with. Careful attention has been paid to ensure that arrival and departure of key servers and data servers in the network does not adversely affect the consistency or performance of the system, and that the emergence of network partitions does not corrupt the data structures of the key servers. In order to accomplish this high level of dependability and consistency, the key servers employ the Spread Toolkit [15] for guaranteeing reliable, totally ordered broadcast communication and group membership protocols.

The Spread Toolkit, or simply, Spread, is a ring-based wide-area group communication and membership system in many ways similar to Totem [3]. It provides varying levels of reliable broadcast communication as well as fault detection and membership protocols. Many of these concepts are necessary for distributed systems and can be found in other toolkits such as Horus [17] or Trans/Total [9]. Spread implements reliable messaging and membership to provide the strongest guarantees possible in such a system: total ordering of messages and extended virtual synchrony. Parker utilizes these provisions for total ordering and virtual synchrony to ensure that data structures remain consistent across all key server instances as the layout of the network changes.

Reliable group communication or broadcast protocols typically provide some guarantee that every non-faulty member of a group receives the same set of broadcast messages as all the other non-faulty members of the group with some order. In reliable broadcast, if a process in the system sends or receives a message, it is assured that all other working processes in the group (those supposed to receive the message) were guaranteed to have received it.

When these broadcast messages might alter the state of a process, simply receiving a message might not be a strong enough guarantee; ordering of these messages between processes becomes essential in making sure that all machines maintain a consistent state. The weakest level of ordering possible is per-process FIFO ordering, where all messages from a single process are guaranteed to be received in the same order they were sent. Strengthening these guarantees, the next level of reliable broadcast message ordering is causal ordering. Causal ordering additionally provides that causally related messages are delivered in a consistent order. Causal ordering relationships [8] describe the temporal relationship of messages that can be proved sequentially ordered due to the progression of processes consuming those messages. For example, if a process P1 broadcasts a message M1, and it is received by P2, the process P2 can subsequently broadcast a message M2. It can then be said that M1 causally precedes M2. Causal ordering guarantees that all processes in the system will receive M1 before M2 if they are causally related in the manner described. It should be noted that causal ordering of two messages contrasts the case where two messages are said to be concurrent: if P2 sent M2 before it received M1 and P1 send M1 before it received M2, there is no way to infer whether M1 or M2 was created first in "actual" time. The final, and strongest, level of reliable broadcast ordering guarantees is total ordering of messages. Total ordering strengthens the guarantees of causal ordering and also guarantees that even concurrent messages will be received in the same order by all processes in the system. Within the total ordering of messages can be a total ordering of events as well, and this represents the concept of virtual synchrony. Virtual synchrony provides that network configuration changes be considered part of the normal delivery of messages to the application [10], such that all processes perceive all types of network activity in the same order. Delivery of messages is guaranteed to all non-faulty members of the (primary) group, and total message ordering is maintained even in the emergence of network partitions or configuration changes. This is particularly important when processes can fail or the network can partition, because such events might otherwise effect consistency of processes if not taken into account.

When bootstrapping the system for the first time, a key server is first initialized, and registers as a process to the Spread daemon running on its local machine. It also joins a logical group called keyserv. The next step in starting the system involves the initialization of a number of data servers. Data servers are brought online and register their availability for work through a broadcast message to keyserv. Every key server in keyserv, on receipt of the broadcast message from the data server, executes a fixed, deterministic algorithm for determining how to insert the data server in its internal data structures that describe the layout of the network. This deterministic allocation scheme relies on the strong total ordering guarantees of broadcast messaging in Spread, and it invariantly holds that the key servers are able to maintain the same set of data structures independent of one another.

Regardless of the number of key servers registered in the keyserv group, a single master key server is responsible for communicating back to new data servers via out-of-band channels as needed to instruct database initialization. Additionally, the master is also responsible for shipping the current state message, containing the most up-to-date information about the layout of the system to new or rejoining key servers. All constituents of keyserv know the master to have the highest membership number (as determined by Spread), with one exception: if a newly joining key server is assigned the highest membership number, the subsequent second key server should be considered the master. The current state message from the master key server to the new key server causally follows the join of the new key server and causally precedes any additional network changes, whereby the total ordering of events guarantees receipt by the new member before the network layout might change again.

It should be noted that data servers, in order to broadcast to keyserv, can register with a Spread daemon running locally or on another machine, such as on that of a key server. This configuration has proved particularly useful in networks where subsequent additions of machines are not guaranteed

Method	Request Time(ms)
heart_beat	0.2451688
get_user	0.5082938
get_profile	0.3084188
get_friends_list	0.3338142
exists	0.259961
put_user	14.14775
put_profile	18.46594
add_friend	17.89232
get_all_users	33.3176
get_random_users	28.62866

Table 1: Latency of API Calls

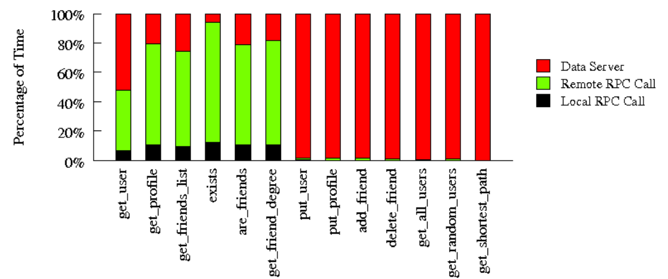

Figure 4: System Overhead

to be in the same subnet, such as on Amazon's EC2. As such, it is not required that the entire network layout be known a priori in order for Parker to work.

6. PERFORMANCE

The following section discusses the performance of Parker.

6.1 Average Latency

Evaluating the baseline performance of the fundamental calls in the system was done using a generated data set of 10,000 users. It is not the intention of this test to demonstrate the scalability of the system with a realistic data set or number of servers. As such, the system setup for this test consisted of a single key server local to the test application, and three data servers on the same 100 MBps LAN. The test was performed on 2.4 GHz Intel Core 2 Duo machines. A total of 5 runs of 20 seconds for each individual call were averaged for the calculation of the relative speeds of calls.

Table 1 shows the resulting speed of the calls in the system. A few interesting trends are apparent. First, all the read calls are approximately the same duration, as are all the write calls. The average read call clocks in at about 0.34 ms, and the average write call is nearly 50 times slower, at about 18.4 ms. The large difference between these two types of calls is likely a result of two important factors. First, because of the small data set used for testing, most of the read calls are cache hits and thus avoid the latency of accessing the disk. This is not an unreasonable scenario, as cache size is a configurable property of the database, and machine memory can be arbitrarily large, with data sets spread out across large numbers of machines. Second, the write calls not only don't complete until they are persisted to physical medium, they also have to receive at least one acknowledgement of a successful write to disk from a replica. This price may seem high, but guarantees the durability of writes in the presence of many types of data server failures.

The next section details just how much time was spend on these different calls in terms of RPC overhead and network latency.

6.2 Connection Overhead

Next, we set out to determine where the time was being spent in performing the API calls. In order to provide a baseline for performance of the system, we have evaluated the performance of RPC call overhead in Thrift. The analysis of this data led to the rational behind the recommended co-location of key servers and application servers for maximum possible system performance and confirmed our suspicions that retaining sets of active connections between key servers and data servers can greatly improve performance. These measurements were taken on four lab machines with 2.4 GHz Intel Core 2 Duo processors on a 100 MBps LAN. Three of the machines ran data servers while the fourth ran both the key server and test application. The tests were averaged over 5 runs of a continuous 20 seconds per call.

The first source of overhead we investigated was the cost of RPC invocation itself. This cost, as measured between the co-located test application and the key server, does not include any data transfer beyond an in-memory copy, and represents the raw call speed. It was determined to be 0.03 ms.

Once a TCP/IP connection is open between the application server and key server (assumed to be a persistent connection), the next source of overhead is the request routing. Any call must be received by the key server and proxied to the appropriate data server over the network. Another 0.03 ms is also spent making the RPC call at the data server. The two approaches we considered for routing requests involved either opening new TCP/IP connections between the key server and data server for every request, or drawing from a pool of already established connections. Not including the RPC overhead incurred on the key servers or data servers, the time it takes to open a new TCP/IP connection for every call and also transport the necessary data over the network is 0.72 ms. Using this approach, the total call cost to to the application is 0.78 ms if the application server and key server are co-located, or about 0.95 ms if they are not. Using connection pooling between key servers and data servers, the network overhead is reduced significantly to 0.17 ms. This approach clearly reduces the remote call overhead significantly, putting the total call cost is 0.23 ms with co-location, or about 0.40 ms without.

Figure 4 is a comparison of the different API calls in Parker, with estimated times spent performing RPC and request routing operations on co-located key and application servers with pooled key server to data server connections. This figure considers network overhead for object data tranfer part of the necessary cost of data server operation. Calls such as exists, which do little network transfer, spend more than 95% of the total operation time being routed to the the data server, but even get_user, which is the most data

intensive of the read operations (returning an average of 1800 bytes), spends about 50% of the time performing just RPC and routing operations. This demonstrates that at least in the case of reads, the importance of optimizing TCP connection overhead is very high.

6.3 Average Request Rate

The final test was designed to determine how many requests the system can serve, and how well it scales with additional servers. These tests are run with increasing numbers of data servers along with increasingly larger number of threads to simulate more work. The trials were all run on Amazons Elastic Compute Cloud [2] with small instance types. For all trials there are three key servers, each with its own test application, along with a variable number of data servers each residing on a separate instance. The database contained a set of 50,000 users which is sufficiently small to ensure it can be held in memory. In order to ensure the data resembled real life user data, all user information was generated based on a sample of 1.2 million Facebook users collected in a previous study[20]. It should be noted that while the performance results show similar patterns to those described above, the rates differ somewhat because of the large differences in hardware and environment (e.g. network).

From the previous tests it is clear that the performance difference between write and read is quite substantial, thus for clarity we chose to test write and read calls separately. The results of the read tests are shown in Figure 6 and the write tests in Figure 5.

Starting with the write results, there are a few interesting observations to be pointed out. With three data servers and 20 threads, 241 write requests are completed per second. However, as the number of threads increases, the number of requests completed only increases slightly. We believe this plateau is a result of the disk(s) becoming the bottleneck. Even with larger numbers of incoming requests, the disk can only service so many requests at a time and is likely to be thrashing. Increasing to six data servers yields an increase of over 2x from three data servers. Furthermore, as the number of threads increases so does the number of requests completed, indicating that the disk bottleneck has not been reached, but is being approached. With nine servers the 20-thread result only demonstrates a small improvement over the same test with six servers, and is in fact worse than some tests with six. However, with increasing numbers of threads the benefits of the increased numbers of servers are shown with 120 threads, demonstrating a 4x improvement over three servers. While initially these results may seem somewhat unintuitive, the reasoning behind it is quite simple. As shown above, there is a large amount of overhead associated with a write call (e.g. sync to multiple disks, TCP) and as a result, low numbers of threads can only achieve so many requests given such high latency. As the number of threads increases more requests can be run in parallel which offsets some of the latency.

Next we test the performance of the read calls. Initially, we note that the baseline for read requests is significantly higher than that of write calls, particularly since reads can be serviced from memory and require no communication with other data servers. With requests being distributed

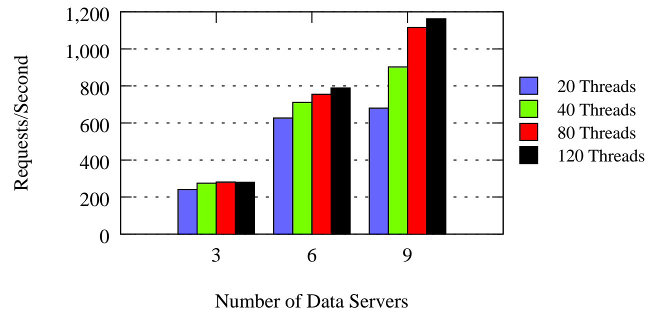

Figure 5: Scaling Write Calls

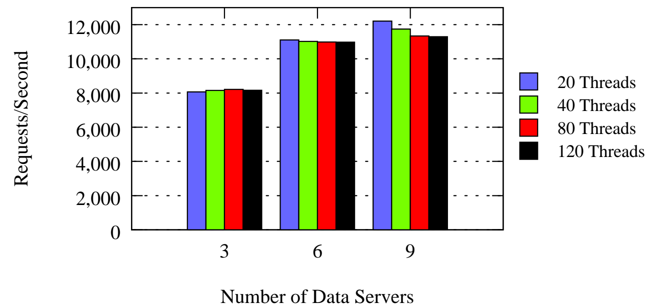

Figure 6: Scaling Read Calls

randomly across three data servers they are able to sustain 8,000 requests per second. Unexpectedly, doubling the number of data servers only yields an improvement of approximately 25%, and, similarly, the increase to nine servers yields only a 31% improvement. Such a firm plateau despite the increased number of data servers indicates that it is not the data servers creating a bottleneck. Similarly increasing the number of threads does not improve performance, indicating that it is not a latency issue as in the write case. From this we infer that the key servers have now become the bottleneck. We believe that with a larger number of key servers, the benefits of additional data servers will be demonstrated.

One also might note the slight decrease in read performance with the addition of threads, particularly with nine data servers. As discussed previously, key servers maintain a pool of open connections to data servers such that requests can be serviced more quickly. The size of that pool begins at 20 connections. As the number of threads increases beyond 20, there are more occasions when a pooled connection is not available, and the resulting overhead of creating a new connection is incurred more frequently. However, since the newly created connection will be added to the pool that overhead only occurs a small portion of the time.

6.4 Performance Comparison

While we cannot compare our system directly to other systems like BigTable or PNUTS, we can make some observations as to how our performance results compare to theirs. It should be noted that the environments under which our tests were run are very different from those used to run either of these systems, but we still believe that comparing the numbers will shed light on how well our system stacks

up.

We begin with the performance of BigTable. Using a single tablet server and a GFS cell of 1786 machines as backing storage, BigTable is able to achieve 10,811 random-reads (in-memory) per second and 8,850 writes per second. All reads and writes are done on 1000-byte records.

Even though we were unable to test our system with such a large number of machines, we can instead compare those results to our largest test running three key servers and nine data servers. While only running a total of 12 machines, Parker was able to achieve over 12,000 random reads per second and 1,163 writes per second. Though Parker does outperform BigTable in terms of reads, it is steadfastly beaten in terms of write throughput. There are two things to note in the write-case. First, the users in our data set were on average over 1800-bytes versus the 1000-byte records used by BigTable. Second, since writes are heavily IO bound, with a larger number of machines (and therefore disks) the performance improves greatly. In the worst-case for writes, Parker was able to attain 280 writes per second on three data servers. Thus we conjecture that with an additional 595 data servers (1786/3 as a result of replication) that we would be able to attain write rates of 16,660 per second (280*595), almost doubling that of BigTable.

We can also compare our performance results to that of Yahoo's PNUTS. While their performance evaluation was run on 30 machines running in three separate data centers, all writes occur at a single data center with 10 machines and thus we use these, best-case results. For insertion, their performance benchmarks show a latency of 33 ms for an ordered table and 131.5 ms for a hash table.

We can contrast these results to the values seen in Section 6. Even in the worst case of write performance (delete_friend), Parker took 23 ms, which is significantly shorter then either of the insertion latencies of PNUTS. Similarly, if we compare PNUTS latencies to those of the put_user call (more analogous the insertion case), we see that it only takes 14 ms, or less than half of that for the best case of PNUTS.

Based on these comparisons we believe that Parker is in fact faster then both systems for storing social network data for real-time applications.

7. DISCUSSION

7.1 Routing Advantages

Many similar systems rely on the mechanism of consistent hashing for maintaining the mapping between partitions of the key space and the machines on which they reside. Although the consistent hashing scheme has the advantage of being simple to implement, it has some downfalls. First, it achieves fault tolerance whereby server failures result in a normally harmless cache miss on a backup machine. This however is not appropriate for Parker since durability is to be provided. Having all key servers know exactly which machines are responsible for any piece data allows us to easily route requests, even in the presence of failures, to machines that are guaranteed to hold up-to-date copies of that data. Furthermore, when a failed server comes back online, it can be instructed to retrieve any changes to the data set from one of the remaining machines. The second advantage of this scheme is that by putting the key servers in charge of network layout, the system can potentially control resource allocation for particularly hard-hit nodes, for example by further splitting a particular range of keys or by assigning extra read-only data servers to reduce load on the primary ones.

Due to the large remote connection overhead for reads as demonstrated in Section 6.2, it is apparent that any extra hops would have an unnecessary negative effect when the speed of a cached read is as quick as we have measured it to be. When key servers are located on the same machine as application servers, the cost of sending a request to the key server is very low. Due to this, we conjecture that the routing in Parker is optimal. This constant time routing is a property that cannot be matched by various distributed hashing techniques, which typically require O(log n) hops.

7.2 Database Performance

The high standards of database durability can ultimately be relaxed and tuned as applications demand. Currently, both local and remote writes to disk are required for a transaction to succeed, but this need not be the case if write performance becomes an issue. For example, it would be possible to require only an acknowledgment of receipt, not of write to disk, by another server in order for a write to succeed. In the face of multiple machine failures, durability would suffer, but the total time of write calls should be nearly cut in half. Further optimization might yield that not even immediate write to local disk be required, either. The only extra overhead beyond the cost of a read would be that of the acknowledgment.

8. FUTURE WORK

We believe there are quite a few avenues available to improve upon the performance of Parker. First, we believe that the addition of caching between the key servers and data servers could greatly improve read performance. In particular, we believe that if caching were employed by the key service layer that reads could be serviced directly from cache, thereby eliminating the cost of serialization and reducing network overhead even more. One difficult issue with caching is ensuring old data is expired promptly so as to not serve stale data. This issue can be easily overcome in this context since write requests come through key servers, upon receiving a write request the key server can immediately expire the data forcing the request to go to a data server. We believe this improvement is an easy win in terms of performance.

Most of the aggregation methods provided by the key servers are single threaded and employ simple algorithms. Since many of these operations are embarrassingly parallel, the addition of threads in these queries will likely greatly improve performance. Similarly, more advanced algorithms could be employed to both improve performance and memory efficiency.

In the related work section we discussed Beehive which proactively replicates "hot" data. We believe the same principal could be applied to social network data. Our system currently supports the dynamic addition of read-only replicas

to any set of data servers. One could potentially maintain a pool of unused data servers which be added as read-only replicas to any overloaded data server groups.

Applying the Beehive concept in another context, while studying the social networking data we noticed in shortest hop queries that there are frequently a large number of paths between users. It is also very likely that a number of these paths traverse through a small set of the most highly connected users. Since all paths of the same length are equivalent to each other, one could devise a search algorithm which attempts to exploit this fact. For example, performing a bi-directional search between the two users covering only a small set of the most popular users could yield faster results. Particularly in social networks with high degrees of connectivity, the traditional bread-first search is impractical.

9. CONCLUSIONS

We have presented a novel architecture for storing social network data in a distributed environment. The system is more than suitable for use as the primary data store for a web application. While previous systems have been created to store graph information, we have shown that storing graph nodes as key-value pairs allows the system to have a natural design while achieving high performance in a real-time environment. An additional benefit is that such an architecture is easily scalable to various types of load.

Our implementation of Parker has shown that such a system can perform well with modest number of machines and can scale to larger number of servers. It is easily extended to meet the requirements of most online social networks. Thus we believe that it is suitable for use building in building the next great online social network.

10. ACKNOWLEDGMENTS

We would like to thank members of UCSB's CURRENT lab for providing us with Facebook data.

11. REFERENCES

[1] Top sites in united states, June 2009. `http://www.alexa.com/topsites/countries/US`.

[2] E. Amazon. Amazon elastic compute cloud. *aws. amazon. com/ec2.*

[3] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems (TOCS)*, 13(4):311–342, 1995.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDIâĂŹ06)*, 2006.

[5] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment archive*, 1(2):1277–1288, 2008.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[7] I. Fischer and T. Meinl. Graph based molecular data mining-an overview. In *2004 IEEE International Conference on Systems, Man and Cybernetics*, volume 5.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. 1978.

[9] P. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, 1990.

[10] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, pages 56–65, 1994.

[11] M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

[12] V. Ramasubramanian and E. Sirer. Beehive: O (1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Symposium on Networked Systems Design and Implementation (NSDIâĂŹ04)*.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 11, pages 329–350. Citeseer, 2001.

[14] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Technical report, Facebook, Palo Alto, CA, April 2007. URL http://developers. facebook. com/thrift/thrift-20070401. pdf. 2.1. 2.

[15] L. Spread Concepts. The Spread toolkit.

[16] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM New York, NY, USA, 2001.

[17] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[18] S. Vinoski and I. Inc. CORBA: integrating diverse applications within distributedheterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[19] T. Washio and H. Motoda. State of the art of graph-based data mining. *Acm Sigkdd Explorations Newsletter*, 5(1):59–68, 2003.

[20] C. Wilson, B. Boe, A. Sala, K. Puttaswamy, and B. Zhao. User interactions in social networks and their implications. In *Proceedings of the fourth ACM european conference on Computer systems*, pages 205–218. ACM New York, NY, USA, 2009.

[21] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. *Computer*, 74, 2001.