PNUTS: Yahoo!'s Hosted Data Serving Platform

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein,
Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni
Yahoo! Research

ABSTRACT

We describe PNUTS, a massively parallel and geographically distributed database system for Yahoo!'s web applications. PNUTS provides data storage organized as hashed or ordered tables, low latency for large numbers of concurrent requests including updates and queries, and novel per-record consistency guarantees. It is a hosted, centrally managed, and geographically distributed service, and utilizes automated load-balancing and failover to reduce operational complexity. The first version of the system is currently serving in production. We describe the motivation for PNUTS and the design and implementation of its table storage and replication layers, and then present experimental results.

1. INTRODUCTION

Modern web applications present unprecedented data management challenges, even for relatively "simple" tasks like managing session state, content meta-data, and user-generated content such as tags and comments. The foremost requirements of a web application are *scalability*, consistently good *response time* for geographically dispersed users, and *high availability*. At the same time, web applications can frequently tolerate *relaxed consistency guarantees*. We now examine these requirements in more detail.

Scalability. For popular applications such as Flickr and del.icio.us, the need for a scalable data engine is obvious [4]. We want not only architectural scalability, but the ability to scale during periods of rapid growth by adding resources with minimal operational effort and minimal impact on system performance.

Response Time and Geographic Scope. A fundamental requirement is that applications must consistently meet Yahoo!'s internal SLAs for page load time, placing stringent response time requirements on the data management platform. Given that web users are scattered across the globe, it is critical to have data replicas on multiple continents for low-latency access. Consider social network applications—alumni of a university in India may reside in North America and Europe as well as Asia, and a particular user's data may be accessed both by the user from his home in London as well as by his friends in Mumbai and San Francisco. Ideally, the data platform should guarantee fast response times to geographically distributed users, even under rapidly changing load conditions brought on by *flash crowds*, *denial of service attacks*, etc.

High Availability and Fault Tolerance. Yahoo! applications must provide a high degree of availability, with application-specific trade-offs in the degree of fault-tolerance required and the degree of consistency that is deemed acceptable in the presence of faults; e.g., all applications want to be able to read data in the presence of failures, while some insist on also being able to write in the presence of failures, even at the cost of risking some data consistency. Downtime means money is lost. If we cannot serve ads, Yahoo! does not get paid; if we cannot render pages, we disappoint users. Thus service must continue in the face of a variety of failures including server failures, network partitions and the loss of power in a co-location facility.

Relaxed Consistency Guarantees. Traditional database systems have long provided us with a well-understood model for reasoning about consistency in the presence of concurrent operations, namely *serializable transactions* [5]. However, there is a tradeoff between performance and availability on the one hand and consistency on the other, and it has repeatedly been observed that supporting general serializable transactions over a globally-replicated and distributed system is very expensive [17, 2, 1]. Thus, given our stringent performance and availability requirements, achieving serializability for general transactions is impractical. Moreover, based on our experience with many web applications at Yahoo!, general transactions are also typically unnecessary, since these applications tend to manipulate only one record at a time. For example, if a user changes an avatar, posts new pictures, or invites several friends to connect, little harm is done if the new avatar is not initially visible to one friend, etc. (especially if such anomalies are rare).

Given that serializability of general transactions is inefficient and often unnecessary, many distributed replicated systems go to the extreme of providing only *eventual con-*

[1]Author emails: {cooperb, ramakris, utkarsh, silberst, plb, nickpuz, dweaver, yerneni}@yahoo-inc.com
Hans-Arno Jacobsen's current affiliation: University of Toronto, jacobsen@eecg.toronto.edu

VLDB '08, August 24-30, 2008, Auckland, New Zealand

sistency [23, 12]: a client can update any replica of an object and all updates to an object will eventually be applied, but potentially in different orders at different replicas. However, such an eventual consistency model is often too weak and hence inadequate for web applications, as the following example illustrates:

EXAMPLE 1. *Consider a photo sharing application that allows users to post photos and control access. For simplicity of exposition, let us assume that each user's record contains both a list of their photos, and the set of people allowed to view those photos. In order to show Alice's photos to Bob, the application reads Alice's record from the database, determines if Bob is in the access list, and then uses the list of Bob's photos to retrieve the actual photo files from a separate serving store. A user wishes to do a sequence of 2 updates to his record:*

U_1: Remove his mother from the list of people who can view his photos

U_2: Post spring-break photos

Under the eventual consistency model, update U_1 can go to replica R_1 of the record, while U_2 might go to replica R_2. Even though the final states of the replicas R_1 and R_2 are guaranteed to be the same (the eventual consistency guarantee), at R_2, for some time, a user is able to read a state of the record that never should have existed: the photos have been posted but the change in access control has not taken place.

This anomaly breaks the application's contract with the user. Note that this anomaly arises because replica R_1 and R_2 apply updates U_1 and U_2 in opposite orders, i.e., replica R_2 applies update U_2 against a stale version of the record.

As these examples illustrate, it is often acceptable to read (slightly) stale data, but occasionally stronger guarantees are required by applications.

1.1 PNUTS Overview

We are building the PNUTS system, a massive-scale, hosted database system to support Yahoo!'s web applications. Our focus is on data serving for web applications, rather than complex queries, e.g., offline analysis of web crawls. We now summarize the key features and architectural decisions in PNUTS.

Data Model and Features PNUTS exposes a simple relational model to users, and supports single-table scans with predicates. Additional features include **scatter-gather** operations, a facility for **asynchronous notification** of clients and a facility for **bulk loading**.

Fault Tolerance PNUTS employs redundancy at multiple levels (data, metadata, serving components, etc.) and leverages our consistency model to support highly-available reads and writes even after a failure or partition.

Pub-Sub Message System Asynchronous operations are carried out over a **topic-based pub/sub system** called Yahoo! Messsage Broker (YMB), which together with PNUTS, is part of Yahoo!'s **Sherpa** data services platform. We chose pub/sub over other asynchronous protocols (such as gossip [12]) because it can be optimized for geographically distant replicas and because replicas do not need to know the location of other replicas.

Record-level Mastering To meet response-time goals, PNUTS cannot use write-all replication protocols that are employed by systems deployed in localized clusters [8, 15]. However, not every read of the data necessarily needs to see the most current version. We have therefore chosen to make **all high latency operations asynchronous**, and to support **record-level mastering**. Synchronously writing to multiple copies around the world can take hundreds of milliseconds or more, while the typical latency budget for the database portion of a web request is only 50-100 milliseconds. Asynchrony allows us to satisfy this budget despite geographic distribution, while record-level mastering allows most requests, including writes, to be satisfied locally.

Hosting PNUTS is a hosted, centrally-managed database service shared by multiple applications. Providing data management as a service significantly reduces application development time, since developers do not have to architect and implement their own scalable, reliable data management solutions. Consolidating multiple applications onto a single service allows us to amortize operations costs over multiple applications, and apply the same best practices to the data management of many different applications. Moreover, having a shared service allows us to keep resources (servers, disks, etc.) in reserve and quickly assign them to applications experiencing a sudden upsurge in popularity.

1.2 Contributions

In this paper, we present the design and functionality of PNUTS, as well as the key protocols and algorithms used to route queries and coordinate the growth of the massive data store. In order to meet the requirements for a web data platform, we have made several fundamental—and sometimes radical—design decisions:

- An architecture based on *record-level*, asynchronous geographic replication, and use of a guaranteed message-delivery service rather than a persistent log.
- A *consistency model* that offers applications transactional features but stops short of full serializability.
- A careful choice of *features* to include (e.g., hashed and ordered table organizations, flexible schemas) or exclude (e.g., limits on ad hoc queries, no referential integrity or serializable transactions).
- Delivery of data management as a *hosted service*.

We discuss these choices, and present the results of an initial performance study of PNUTS. The first version of our system is being used in production to support social web and advertising applications. As the system continues to mature, and some additional features (see Section 6) become available, it will be used for a variety of other applications.

2. FUNCTIONALITY

In this section we briefly present the functionality of PNUTS, and point out ways in which it is limited by our desire to keep the system as simple as possible while still meeting the key requirements of Yahoo! application developers. We first outline the data and query model, then present the consistency model and notification model, and end with a brief discussion of the need for efficient bulk loading.

2.1 Data and Query Model

PNUTS presents a simplified relational data model to the user. Data is organized into tables of records with attributes. In addition to typical data types, "blob" is a valid data type, allowing arbitrary structures inside a record, but not necessarily large binary objects like images or audio. (We observe that blob fields, which are manipulated entirely in application logic, are used extensively in practice.) Schemas are flexible: new attributes can be added at any time without halting query or update activity, and records are not required to have values for all attributes.

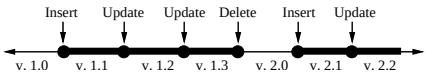
The query language of PNUTS supports selection and projection from a single table. Updates and deletes must specify the primary key. While restrictive compared to relational systems, single-table queries in fact provide very flexible access compared to distributed hash [12] or ordered [8] data stores, and present opportunities for future optimization by the system (see Section 3.3.1). Consider again our hypothetical social networking application: A user may update her own record, resulting in *point access*. Another user may scan a set of friends in order by name, resulting in *range access*. PNUTS allows applications to declare tables to be hashed or ordered, supporting both workloads efficently.

Our system is designed primarily for online serving workloads that consist mostly of queries that read and write single records or small groups of records. Thus, we expect most scans to be of just a few tens or hundreds of records, and optimize accordingly. Scans can specify predicates which are evaluated at the server. Similarly, we provide a "multiget" operation which supports retrieving multiple records (from one or more tables) in parallel by specifying a set of primary keys and an optional predicate, but again expect that the number of records retrieved will be a few thousand at most.

Our system, regrettably, also does not enforce constraints such as referential integrity, although this would be very desirable. The implementation challenges in a system with fine-grained asynchrony are significant, and require future work. Another missing feature is complex ad hoc queries (joins, group-by, etc.). While improving query functionality is a topic of future work, it must be accomplished in a way that does not jeapardize the response-time and availability currently guaranteed to the more "transactional" requests of web applications. In the shorter term, we plan to provide an interface for both Hadoop, an open source implementation of MapReduce [11], and Pig [21], to pull data out of PNUTS for analysis, much as MapReduce pulls data out of BigTable [8].

2.2 Consistency Model: Hiding the Complexity of Replication

PNUTS provides a consistency model that is between the two extremes of general serializability and eventual consistency. Our model stems from our earlier observation that web applications typically manipulate one record at a time, while different records may have activity with different geographic locality. We provide **per-record timeline consistency**: all replicas of a given record apply all updates to the record in the same order. An example sequence of updates to a record is shown in this diagram:



In this diagram, the events on the timeline are inserts, updates and deletes for a particular primary key. The intervals between an insert and a delete, shown by a dark line in the diagram, represent times when the record is physically present in the database. A read of any replica will return a consistent version from this timeline, and replicas always move forward in the timeline. This model is implemented as follows. One of the replicas is designated as the master, independently for each record, and all updates to that record are forwarded to the master. The master replica for a record is adaptively changed to suit the workload – the replica receiving the majority of write requests for a particular record becomes the master for that record. The record carries a sequence number that is incremented on every write. As shown in the diagram, the sequence number consists of the *generation* of the record (each new insert is a new generation) and the *version* of the record (each update of an existing record creates a new version). Note that we (currently) keep only one version of a record at each replica.

Using this per-record timeline consistency model, we support a whole range of API calls with varying levels of consistency guarantees.

- `Read-any`: Returns a possibly stale version of the record. However, unlike Example 1, the returned record is always a valid one from the record's history. Note that this call departs from strict serializability since with this call, even after doing a successful write, it is possible to see a stale version of the record. Since this call has lower latency than other read calls with stricter guarantees (described next), it provides a way for the application to explicitly indicate, on a per-read basis, that performance matters more than consistency. For example, in a social networking application, for displaying a user's friend's status, it is not absolutely essential to get the most up-to-date value, and hence `read-any` can be used.
- `Read-critical(required_version)`: Returns a version of the record that is strictly newer than, or the same as the `required_version`. A typical application of this call is when a user writes a record, and then wants to read a version of the record that definitely reflects his changes. Our write call returns the version number of the record written, and hence the desired read guarantee can be enforced by using a `read-critical` with `required_version` set to the version returned by the write.
- `Read-latest`: Returns the latest copy of the record that reflects all writes that have succeeded. Note that `read-critical` and `read-latest` may have a higher latency than `read-any` if the local copy is too stale and the system needs to locate a newer version at a remote replica.
- `Write`: This call gives the same ACID guarantees as a transaction with a single write operation in it. This call is useful for *blind* writes, e.g., a user updating his status on his profile.
- `Test-and-set-write(required_version)`: This call performs the requested write to the record if and only if the present version of the record is the same as `required_version`. This call can be used to implement transactions that first read a record, and then do a write to the record based on the read, e.g., incrementing the value of a counter. The test-and-set write ensures that two such concurrent increment transactions are properly serialized. Such a primitive is a well-known form of opti-

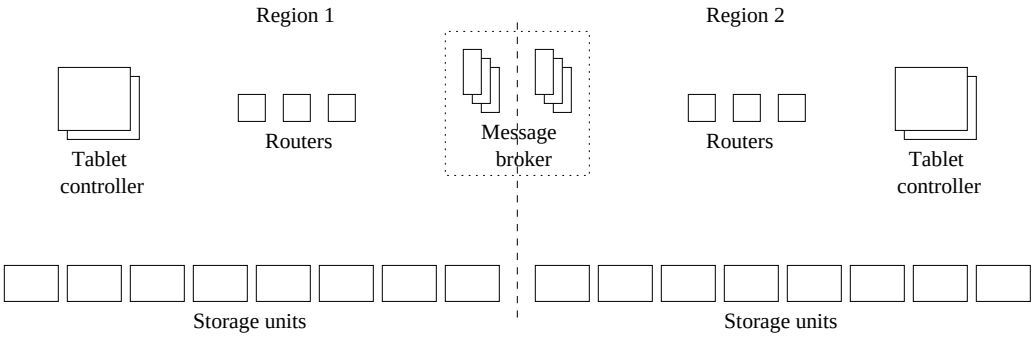


Figure 1: PNUTS system architecture

mistic concurrency control [5].

Our API, in contrast to that of SQL, may be criticized for revealing too many implementation details such as sequence numbers. However, revealing these details does allow the application to indicate cases where it can do with some relaxed consistency for higher performance, e.g., `read-critical`. Similarly, a `test-and-set write` allows us to implement single-row transactions without any locks, a highly desirable property in distributed systems. Of course, if the need arises, our API can be packaged into the traditional `BEGIN TRANSACTION` and `COMMIT` for single-row transactions, at the cost of losing expressiveness. Note that our consistency guarantees are somewhat different than traditional guarantees such as *serializable*, *repeatable read*, *read committed*, *snapshot isolation* and so on. In particular, we make no guarantees as to consistency for multi-record transactions. Our model can provide serializability on a per-record basis. In particular, if an application reads or writes the same record multiple times in the same "transaction," the application must use record versions to validate its own reads and writes to ensure serializability for the "transaction."

In the future, we plan to augment our consistency model with the following primitives:

- `Bundled updates`: Consistency guarantees for write operations that span multiple records (see Section 6).
- `Relaxed consistency`: Under normal operation, if the master copy of a record fails, our system has protocols to fail over to another replica. However, if there are major outages, e.g., the entire region that had the master copy for a record becomes unreachable, updates cannot continue at another replica without potentially violating record-timeline consistency. We will allow applications to indicate, per-table, whether they want updates to continue in the presence of major outages, potentially branching the record timeline. If so, we will provide automatic conflict resolution and notifications thereof. The application will also be able to choose from several conflict resolution policies: e.g., discarding one branch, or merging updates from branches, etc.

2.3 Notification

Trigger-like *notifications* are important for applications such as ad serving, which must invalidate cached copies of ads when the advertising contract expires. Accordingly, we allow the user to *subscribe* to the stream of updates on a table. Notifications are easy to provide given our underlying pub/sub infrastructure (see Section 3.2.1), and thus have the same stringent reliability guarantees as our data replication mechanism.

2.4 Bulk Load

While we emphasize scalability, we seek to support important database system features whenever possible. *Bulk loading* tools are necessary for applications such as comparison shopping, which upload large blocks of new sale listings into the database every day. Bulk inserts can be done in parallel to multiple storage units for fast loading. In the hash table case, the hash function naturally load balances the inserts across storage units. However, in the ordered table case, bulk inserts of ordered records, records appended to the end of the table's range, or records inserted into already populated key ranges require careful handling to avoid hot spots and ensure high performance. These issues are discussed in [25].

3. SYSTEM ARCHITECTURE

Figure 1 shows the system architecture of PNUTS. The system is divided into *regions*, where each region contains a full complement of system components and a complete copy of each table. Regions are typically, but not necessarily, geographically distributed. A key feature of PNUTS is the use of a pub/sub mechanism for both reliability and replication. In fact, our system does not have a traditional database log or archive data. Instead, we rely on the guaranteed delivery pub/sub mechanism to act as our redo log, replaying updates that are lost before being applied to disk due to failure. The replication of data to multiple regions provides additional reliability, obviating the need for archiving or backups. In this section, we first discuss how the components within a region provide data storage and retrieval. We then examine how our pub/sub mechanism, the Yahoo! Message Broker, provides reliable replication and helps with recovery. Then, we examine other aspects of the system, including query processing and notifications. Finally, we discuss how these components are deployed as a hosted database service.

3.1 Data Storage and Retrieval

Data tables are horizontally partitioned into groups of records called *tablets*. Tablets are scattered across many servers; each server might have hundreds or thousands of tablets, but each tablet is stored on a single server within a region. A typical tablet in our implementation is a few

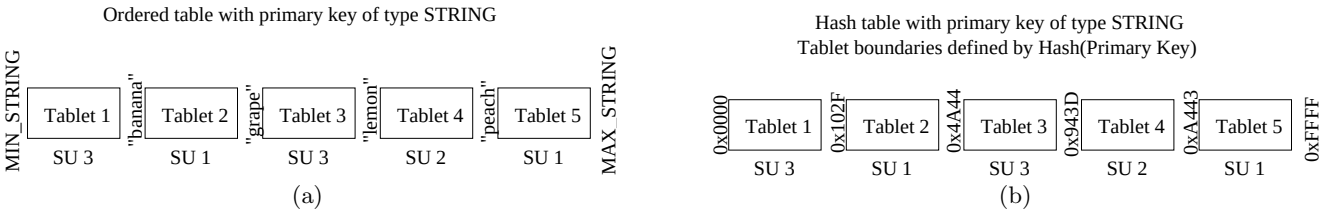


Figure 2: Interval mappings: (a) ordered table, (b) hash table.

hundred megabytes or a few gigabytes, and contains thousands or tens of thousands of records. The assignment of tablets to servers is flexible, which allows us to balance load by moving a few tablets from an overloaded server to an underloaded server. Similarly, if a server fails, we can divide its recovered tablets over multiple existing or new servers, spreading the load evenly.

Three components in Figure 1 are primarily responsible for managing and providing access to data tablets: the **storage unit**, the **router**, and the **tablet controller**. Storage units store tablets, respond to `get()` and `scan()` requests by retrieving and returning matching records, and respond to `set()` requests by processing the update. Updates are committed by first writing them to the **message broker**, as described in the next section. The storage unit can use any physical storage layer that is appropriate. For hash tables, our implementation uses a UNIX filesystem-based hash table implemented originally for Yahoo!'s user database. For ordered tables, we use MySQL with InnoDB because it stores records ordered by primary key. Schema flexibility is provided for both storage engines by storing records as parsed JSON objects.

In order to determine which storage unit is responsible for a given record to be read or written by the client, we must first determine which tablet contains the record, and then determine which storage unit has that tablet. Both of these functions are carried out by the **router**. For ordered tables, the primary-key space of a table is divided into intervals, and each interval corresponds to one tablet. The router stores an *interval mapping*, which defines the boundaries of each tablet, and also maps each tablet to a storage unit. An example is shown in Figure 2a. This mapping is similar to a very large root node of a B+ tree. In order to find the tablet for a given primary key, we conduct a binary search over the interval mapping to find the tablet enclosing the key. Once we find the tablet, we have also found the appropriate storage server.

For hash-organized tables, we use an n-bit hash function $H()$ that produces hash values $0 \leq H() < 2^n$. The hash space $[0...2^n)$ is divided into intervals, and each interval corresponds to a single tablet. An example is shown in Figure 2b. To map a key to a tablet, we hash the key, and then search the set of intervals, again using binary search, to locate the enclosing interval and thus the tablet and storage unit. We chose this mechanism, instead of a more traditional linear or extensible hashing mechanism, because of its symmetry with the ordered table mechanism. Thus, we can use the same code to maintain and search interval mappings for both hash and ordered tables.

The interval mapping fits in memory, making it inexpensive to search. For example, our planned scale is about 1,000 servers per region, with 1,000 tablets each. If keys are 100 bytes (which is on the high end for our anticipated applications) the total mapping takes a few hundred megabytes of RAM (containing one key and storage unit address per tablet.) Note that with tablets that are 500 MB on average, this corresponds to a database that is 500 terabytes in size. For much larger databases, the mapping may not fit in memory, and we will have to use a mapping that is optimized for disk-based access.

Routers contain only a cached copy of the interval mapping. The mapping is owned by the **tablet controller**, and routers periodically poll the tablet controller to get any changes to the mapping. The tablet controller determines when it is time to move a tablet between storage units for load balancing or recovery and when a large tablet must be split. In each case, the controller will update the authoritative copy of the mapping. For a short time after a tablet moves or splits, the routers' mappings will be out of date, and requests will be misdirected. A misdirected request results in a storage unit error response, causing the router to retrieve a new copy of the mapping from the controller. Thus, routers have purely soft state; if a router fails, we simply start a new one and do not perform recovery on the failed router. The tablet controller is a single pair of active/standby servers, but the controller is not a bottleneck because it does not sit on the data path.

The primary bottleneck in our system is disk seek capacity on the storage units and message brokers. For this reason, different PNUTS customers are currently assigned different clusters of storage units and message broker machines (as a simple form of quality-of-service and isolation from other customers). Customers can share routers and tablet controllers. In the future, we would like all customers to be able to share all components, so that sudden load spikes can be absorbed across all of the available server machines. However, this requires us to examine flexible quota and admission control mechanisms to ensure each application receives its fair share of the system.

3.2 Replication and Consistency

Our system uses asynchronous replication to ensure low-latency updates. We use the Yahoo! **message broker**, a publish/subscribe system developed at Yahoo!, both as our replacement for a redo log and as our replication mechanism.

3.2.1 Yahoo! Message Broker

Yahoo! Messsage Broker (YMB) is a topic-based pub/sub system, which together with PNUTS, is part of Yahoo!'s **Sherpa** data services platform. Data updates are considered "committed" when they have been published to YMB. At some point after being committed, the update will be

asynchronously propagated to different regions and applied to their replicas. Because replicas may not reflect the latest updates, we have had to develop a consistency model that helps programmers deal with staleness; see Section 2.2.

We are able to use YMB for replication and logging for two reasons. First, YMB takes multiple steps to ensure messages are not lost before they are applied to the database. YMB guarantees that published messages will be delivered to all topic subscribers even in the presence of single broker machine failures. It does this by logging the message to multiple disks on different servers. In our current configuration, two copies are logged initially, and more copies are logged as the message propagates. The message is not purged from the YMB log until PNUTS has verified that the update is applied to all replicas of the database. Second, YMB is designed for wide-area replication: YMB clusters reside in different, geographically separated datacenters, and messages published to one YMB cluster will be relayed to other YMB clusters for delivery to local subscribers. This mechanism isolates individual PNUTS clusters from dealing with update propagation between regions.

YMB provides partial ordering of published messages. Messages published to a particular YMB cluster will be delivered to all subscribers in the order they were published. However, messages published to different YMB clusters may be delivered in any order. Thus, in order to provide timeline consistency, we have developed a per-record mastership mechanism, and the updates published by a record's master to a single YMB cluster are delivered in the published order to other replicas (see the next section). While stronger ordering guarantees would simplify this protocol, global ordering is too expensive to provide when different brokers are located in geographically separated datacenters.

3.2.2 Consistency via YMB and mastership

Per-record timeline consistency is provided by designating one copy of a record as the master, and directing all updates to the master copy. In this **record-level mastering** mechanism, mastership is assigned on a record-by-record basis, and different records in the same table can be mastered in different clusters. We chose this mechanism because we have observed significant write locality on a per-record basis in our web workloads. For example, a one week trace of updates to 9.8 million user ids in Yahoo!'s user database showed that on average, 85 percent of the writes to a given record originated in the same datacenter. This high locality justifies the use of a master protocol from a performance standpoint. However, since different records have update affinity for different datacenters, the granularity of mastership must be per-record, not per tablet or per table; otherwise, many writes would pay expensive cross-region latency to reach the master copy.

All updates are propagated to non-master replicas by publishing them to the message broker, and once the update is published we treat it as committed. A master publishes its updates to a single broker, and thus updates are delivered to replicas in commit order. Because storage units are so numerous, we prefer to use cheaper, commodity servers instead of expensive, highly reliable storage. By leveraging the reliable publishing properties of YMB, we can survive data-loss failures on storage units—any "committed" update is recoverable from a remote replica, and we do not have to recover any data from the failed storage unit itself.

Updates for a record can originate in a non-master region, but must be forwarded to the master replica before being committed. Each record maintains, in a hidden metadata field, the identity of the current master. If a storage unit receives a `set()` request, it first reads the record to determine if it is the master, and if not, what replica to forward the request to. The mastership of a record can migrate between replicas. If a user moves from Wisconsin to California, the system will notice that the write load for the record has shifted to a different datacenter (using another hidden metadata field in the record that maintains the origin of the last N updates) and will publish a message to YMB indicating the identity of the new master. In the current implementation, $N = 3$, and since our region names are 2 bytes this tracking adds only a few bytes of overhead to each record.

In order to enforce primary key constraints, we must send inserts of records with the same primary key to the same storage unit; this storage unit will arbitrate and decide which insert came first and reject the others. Thus, we have to designate one copy of each tablet as the *tablet master*, and send all inserts into a given tablet to the tablet master. The tablet master can be different than the record level master assigned to each record in the tablet.

3.2.3 Recovery

Recovering from a failure involves copying lost tablets from another replica. Copying a tablet is a three step process. First, the tablet controller requests a copy from a particular remote replica (the "source tablet"). Second, a "checkpoint message" is published to YMB, to ensure that any in-flight updates at the time the copy is initiated are applied to the source tablet. Third, the source tablet is copied to the destination region. To support this recovery protocol, tablet boundaries are kept synchronized across replicas, and tablet splits are conducted by having all regions split a tablet at the same point (coordinated by a two-phase commit between regions). Most of the time in this protocol is spent transferring the tablet from one region to another. Note that in practice, because of the bandwidth cost and latency needed to retrieve tablets from remote regions, it may be desirable to create "backup regions" which maintain a back-up replica near serving replicas. Then, recovering a table would involve transferring it from a "region" in the same or a nearby datacenter, rather than from a geographically distant datacenter.

3.3 Other Database System Functionality

3.3.1 Query Processing

Operations that read or update a single record can be directly forwarded to the storage unit holding (the tablet that contains) the record. However, operations that touch multiple records require a component that generates multiple requests and monitors their success or failure. The component responsible for multi-record requests is called the **scatter-gather engine**, and is a component of the router. The scatter-gather engine receives a multi-record request, splits it into multiple individual requests for single records or single tablet scans, and initiates those requests in parallel. As the requests return success or failure, the scatter-gather engine assembles the results and then passes them to the client. In our implementation, the engine can begin stream-

ing some results back to the client as soon as they appear. We chose a server-side approach instead of having the client initiate multiple parallel requests for several reasons. First, at the TCP/IP layer, it is preferable to have one connection per client to the PNUTS service; since there are many clients (and many concurrent processes per client machine) opening one connection to PNUTS for each record being requested in parallel overloads the network stack. Second, placing this functionality on the server side allows us to optimize, for example by grouping multiple requests to the same storage server in the same web service call.

Range queries and table scans are also handled by the scatter gather engine. Typically there is only a single client process retrieving the results for a query. The scatter gather engine will scan only one tablet at a time and return results to the client; this is about as fast as a typical client can process results. In the case of a range scan, this mechanism simplifies the process of returning the top-K results (a frequently requested feature), since we only need to scan enough tablets to provide K results. After returning the first set of results, the scatter-gather engine constructs and returns a *continuation object*, which allows the client to retrieve the next set of results. The continuation object contains a modified range query, which, when executed, restarts the range scan at the point the previous results left off. Continuation objects allow us to have cursor state on the client side rather than the server. In a shared service such as PNUTS, it is essential to minimize the amount of server-side state we have to manage on behalf of clients.

Future versions of PNUTS will include query optimization techniques that go beyond this simple incremental scanning. For example, if clients can supply multiple processes to retrieve results, we can stream query results back in parallel, achieving more throughput and leveraging the inherently parallel nature of the system. Also, if a client has specified a predicate for a range or table scan, we might have to scan multiple tablets in order to find even a few matching results. In this case, we will maintain and use statistics about data to determine the expected number of tablets that must be scanned, and scan that many tablets in parallel for each set of results we return.

We have deliberately eschewed complex queries involving joins and aggregation, to minimize the likelihood of unanticipated spikes in system workload. In future, based on experience with the system and if there is strong user demand, we might consider expanding the query language, since there is nothing in our design that fundamentally prevents us from supporting a richer query language.

3.3.2 Notifications

PNUTS provides a service for notifying external systems of updates to data, for example to maintain an external data cache or populate a keyword search engine index. Because we already use a pub/sub message broker to replicate updates between regions, providing a basic notification service involves allowing external clients to subscribe to our message broker and receive updates. This architecture presented two main challenges. First, there is one message broker topic per tablet, and external subscribers need to know which topics to subscribe to. However, we want to isolate clients from knowledge of tablet organization (so that we can split and reorganize tablets without having to notify clients). Therefore, our notification service provides a mechanism to subscribe to all topics for a table; whenever a new topic is created due to a tablet split, the client is automatically subscribed. Second, slow clients can cause undelivered messages to back up on the message broker, consuming resources. Our current policy is to break the subscriptions of slow notification clients and discard their messages when the backlog gets too large. In the future, we plan to examine other policies.

3.4 Hosted Database Service

PNUTS is a hosted, centrally-managed database service shared by multiple applications. To add capacity, we add servers. The system adapts by automatically shifting some load to the new servers. The bottleneck for some applications is the number of disk seeks that can be done concurrently; for others it is the amount of aggregate RAM for caching or CPU cycles for processing queries. In all cases, adding more servers adds more of the bottleneck resource. When servers have a hard failure (such as a burnt out power supply or RAID controller failure), we automatically recover by copying data (from a replica) to other live servers (new or existing), carrying out little or no recovery on the failed server itself. Our goal is to scale to more than ten worldwide replicas, each with 1,000 or more servers. At this scale, automated failover and load balancing is the only way to manage the operations load.

This hosted model introduces several complications that must be dealt with. First, different applications have different workloads and requirements, even within our relatively narrow niche of web serving applications. Therefore, the system must support several different workload profiles, and be automatically or easily tunable to different profiles. For example, our mastership migration protocol adapts to the observed write patterns of different applications. Second, we need performance isolation so that one heavyweight application does not negatively impact the performance of other applications. In our current implementation, performance isolation is provided by assigning different applications to different sets of storage units within a region.

4. PNUTS APPLICATIONS

In this section, we briefly describe the Yahoo! applications that motivated and influenced PNUTS. Some of these applications are currently running on PNUTS, while others are planned to do so in future.

User Database Yahoo!'s user database has hundreds of millions of active user IDs, and billions of total IDs. Each record contains user preferences, profile information and usage statistics, as well as application-specific data such as the location of the user's mail home or their Yahoo! Instant Messenger buddy list. Data is read and possibly written on every user page view, and thus the volume of traffic is extremely high. The massive parallelism of PNUTS will help support the huge number of concurrent requests. Also, our asynchrony model provides low latency, which is critical given the need to read and often write the user database on every page view. User data cannot be lost, but relaxed consistency is acceptable: the user must see his own changes but it is fine if other users do not see the user's changes for some time. Thus, our record timeline model is a good fit. The user database also functions well under a hosted service model, since many different applications need to share this data.

Social Applications Social and "Web 2.0" applications require a flexible data store that can support operations geared around information sharing and connections between users. The flexible schemas of PNUTS will help support rapidly evolving and expanding social applications. Similarly, the ordered table abstraction is useful to represent connections in a social graph. We can create a relationship table with a primary key that is a composite of (Friend1, Friend2), and then find all of a user's friends by requesting a range scan for all records with key prefixed by a given user ID. Because such relationship data (and other social information) is useful across applications, our hosted service model is a good fit. Social applications typically have large numbers of small updates, as the database is updated every time a user posts a photo or writes a blog post. Thus, scalability to high write rates, as provided by our parallel system, is essential. However, the dissemination of these updates to other users does not have to be real time, which means that our relaxed consistency model works well for this application. The first set of production applications running on PNUTS are social applications.

Content Meta-Data Mass storage of data such as mail attachments, images and video is a challenge at Yahoo! and many other web companies. While PNUTS is not optimized to provide bulk storage, it can play a crucial role as the metadata store of a distributed bulk storage system. While a distributed bulk filesystem may store the actual file blocks, PNUTS can manage the structured metadata normally stored inside directories and inodes. One planned customer of PNUTS will use the system as such a metadata store, utilizing its scalability and low latency to ensure high performance for metadata operations such as file creation, deletion, renaming and moving between directories. The consistency model of PNUTS is critical to properly managing this metadata without sacrificing scalability.

Listings Management Comparison shopping sites such as Yahoo! Shopping aggregate listings of items for sale from many sources. These sites provide the ability to search for items and sort by price, ratings, etc. Our ordered table can be used to store listings, sorted by timestamp, to allow shopping sites to show the most recent N items. Creating an index or view of the data (see Section 6) will allow us to retrieve items sorted by secondary attributes such as price. Also, the flexible schemas in PNUTS will make it easy to model the varied attributes of different kinds of products.

Session Data Web sites often maintain per-session state, and the large number of concurrent sessions active at a large site like Yahoo! means that a scalable storage system is needed to manage the state. Strong consistency is not required for this state, and to enhance performance, applications may decide to turn off all PNUTS consistency for session tables. Because managing session state is a common task across many different web applications, running PNUTS as a service allows applications to quickly use the session store without having to architect and implement their own solution.

5. EXPERIMENTAL RESULTS

We ran a series of experiments to evaluate the performance of our system. Our performance metric was average request latency, since minimizing latency is a primary goal of the system. We compared the performance of the hash and ordered tables. For these experiments, we set up a three-region PNUTS cluster, with two regions on the west coast of the United States and one region on the east coast. We ran our experiments on this test cluster instead of the larger production system so that we could modify parameters and code and measure the performance impact. In ongoing work we are gathering measurements on larger installations.

Component	*Servers/region*	*OS*
Storage unit	5	FreeBSD 6.3
Message broker	2	FreeBSD 6.3
Router	1	Linux RHEL 4
Tablet controller	1	Linux RHEL 4
Client	1	Linux RHEL 4

Region	*Machine*
West 1, West 2	Dual 2.8 GHz Xeon, 4GB RAM, 6 disk RAID 5 array
East	Quad 2.13 GHz Xeon, 4GB RAM, 1 SATA disk

Table 1: Machine configurations for different regions. Storage units and message brokers run on FreeBSD because of their use of BSD-only packages. The difference in machine configuration between the West and East areas results from our using repurposed machines for our experiments.

5.1 Experimental Setup

We used an enhanced version of the production code of PNUTS in our experiments. The current production version supports only hash tables; the enhanced system also supports ordered tables. The primary change needed to support ordered tables was to replace the storage engine (a Yahoo! proprietary disk-based hashtable) with MySQL using InnoDB to get good range scan performance within a storage unit. We also needed to modify the router to support lookup by primary key in addition to hash of primary key, and the tablet controller to create tablets named by key range instead of by hash range. The system is written primarily in C++ with some components, in particular the tablet controller and the administrative scripts, written in PHP and Perl. We set up three PNUTS regions, two in the San Francisco Bay Area in California (regions "West 1" and "West 2") and one in Virginia (region "East"). The configuration of each region is shown in Table 1.

Our database contained synthetically generated 1 KB records, replicated across three regions, each with 128 tablets. We generated workload against the database by running a workload process on a separate server in each region. Each process had 100 client threads, for a total of 300 clients across the system. For each client thread, we specified the number of requests, the rate to generate requests, the mix of reads and writes, and the probability that an updated record was mastered in the same region as the client (called the "locality"). The values we used for these parameters are shown in Table 2. In particular, the locality parameter is based on our measurements of the production Yahoo! user database. Each update overwrote half of the record. In most of our experiments the records to be read or written were chosen randomly according to a uniform distribution. We also experimented with a Zipfian distribution to measure the impact of skew on our system.

Total clients	300
Requests per client	1,000
Request rate	1200 to 3600 requests/sec (4 to 12 requests/sec/client)
Read:write mix	0 to 50 percent writes
Locality	0.8

Table 2: Experimental parameters

5.2 Inserting Data

Our experiment client issues multiple insertion requests in parallel. In the hash table case, record insertions are naturally load balanced across storage servers because of the hashing function, allowing us to take advantage of the parallelism inherent in the system. Recall that to enforce primary key constraints, we must designate a "tablet master" where all inserts are forwarded. We used 99 clients (33 per region) to insert 1 million records, one third into each region; in this experiment the tablet master region was West 1. Inserts required 75.6 ms per insert in West 1 (tablet master), 131.5 ms per insert into the non-master West 2, and 315.5 ms per insert into the non-master East. These results show the expected effect that the cost of inserting is significantly higher if the insert is initiated in a non-master region that is far away from the tablet master. We experimented with varying the number of clients and found that above 100 clients, increasing contention for tablets introduced higher latency and in some cases timeouts on insertions. We can add storage units to alleviate this problem. Insertion time for the ordered table showed a similar trend, requiring 33 ms per insert in West 1, 105.8 ms per insert in the non-master West 2, and 324.5 ms per insert in the non-master East. These measurements demonstrate that MySQL is faster than our hashtable implementation under moderate load. However, if the concurrent insert load on the system is too high, MySQL experiences a performance drop due to contention (in particular, concurrent inserts into the primary key index). For this reason, when inserting into the ordered table we only used 60 clients. Again, more storage units would help. (For performance on bulk-loading records into an ordered table in primary key order, see [25]; this operation requires special handling in order to achieve high parallelism.) Our storage unit implementation is not particularly optimized; for example out of the 75.6 ms latency per insert into the hash table, 40.2 ms was spent in the storage unit itself, obtaining locks, updating metadata structures and writing to disk. It should be possible to optimize this performance further.

5.3 Varying Load

We ran an experiment where we examined the impact of increased load on the average latency for requests. In this experiment, we varied the total request rate across the system between 1200 and 3600 requests/second, while 10 percent of the requests were writes. The results are shown in Figure 3. The figure shows that latency decreases, and then increases, with increasing load. As load increases, contention for resources (in particular, disk seeks) increases, resulting in higher latency. The exception is the left portion of the graph, where latency initially decreases as load increases. The high latency at low request rate resulted from an anomaly in the HTTP client library we used, which closed TCP connections in between requests at low request rates, requiring expensive TCP setup for each call. This library

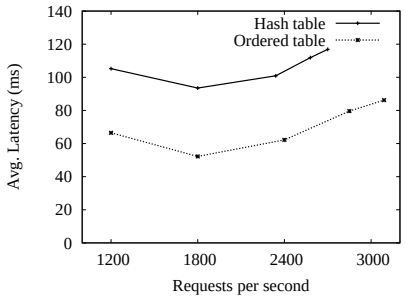


Figure 3: Impact of varying request rate on the average request latency.

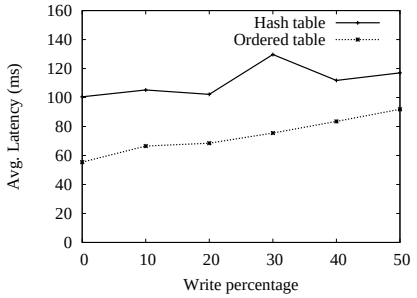


Figure 4: Impact of varying read:write ratio on the average request latency.

should be optimized or replaced to ensure high performance even for lower request rates. Also, while the largest request rate we targeted was 3,600 requests/second, the clients were only able to achieve 3,090 requests/second for the ordered table and 2,700 requests/second for the hash table. Because clients sent requests serially, the high latency limited the maximum request rate.

5.4 Varying Read/Write Ratio

Next, we examined the effect of the read/write mix on latency. We varied the percentage of writes between 0 and 50 percent of requests, while keeping the request rate at 1200 requests/second. The results are shown in Figure 4. As the proportion of reads increases, the average latency decreases. Read requests can be satisfied by a local copy of the data, while write requests must be satisfied by the master record. When a write request originates in a non-master region (20 percent of the time), it must be forwarded to the master region, incurring high latency. Thus, when the proportion of writes is high, there is a large number of high-latency requests, resulting in a large average latency. In particular, writes that had to be forwarded from the east to the west coast, or vice versa, required 324.4 ms on average compared to 92.0 ms on average for writes that could complete locally; these measurements are for the hash table but measurements for the ordered table were comparable. As the number of writes decreases, more requests can be satisfied locally, bringing down the average latency. This result demonstrates the benefit of our timeline consistency model: because most reads can be satisfied by local, but possibly stale data, read latency is low. Even the higher latency for

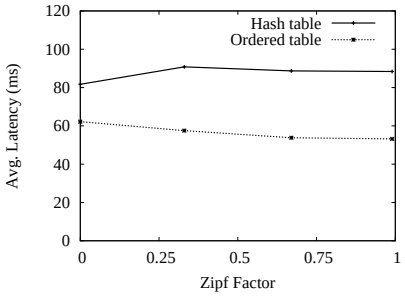


Figure 5: Impact of varying the skew of requests on the average request latency.

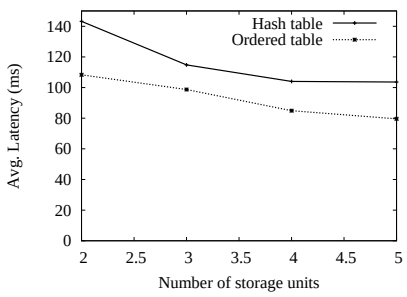


Figure 6: Impact of varying the number of storage units on the average request latency.

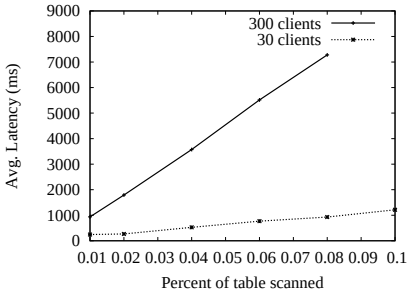


Figure 7: Impact of varying the size of range scans on the average request latency.

writes is mitigated by the fact that per-record mastering allows us to assign the master to the region with the most updates for a record. The "bump" observed at 30 percent writes for the hash table is an anomaly; on examining our numbers we determined that there was unusually high latency within the datacenter where our West 1 and West 2 regions reside, most likely due to congestion caused by other systems running in the datacenter.

5.5 Varying Skew

We also examined a workload where the popularity of records was non-uniform. It has been observed that web workloads often exhibit Zipfian skew [7]. We ran an experiment where the probability of requesting a record varied according to a Zipfian distribution, with the Zipf parameter varying from 0 (uniform) to 1 (highly skewed). The workload had 10 percent writes, and 1,200 requests/second. The results in Figure 5 show that for the hash table, average request latency first increases, and then decreases slightly with higher skew. More skew causes more load imbalance (and higher latency), but this effect is soon outweighed by better cache locality for popular records (resulting in lower latency). In the ordered table, the improved caching always dominates, resulting in better latency with more skew.

5.6 Varying Number of Storage Units

Next, we examined the impact of the number of storage units on latency. One of the main advantages of our system is the ability to scale by adding more servers. We varied the number of storage units per region from 2-5. The workload had 10 percent writes, and 1,200 requests/second. The results are shown in Figure 6. As the figure shows, latency decreases with increasing number of storage units. More storage units provide more capacity, in particular disk seek capacity, reducing the latency of individual requests. The decrease is roughly linear; we attribute the not-quite-linearity of the hash table result to experimental noise.

5.7 Varying Size of Range Scans

Finally, we examined the impact of range scan size on latency. This experiment was run only with the ordered table. The hash table inherently does not support range scans by primary key unless we do an expensive table scan. We varied the size of range scans between 0.01 percent and 0.1 percent of the total table. We also conducted runs with 30 clients (10 per region) and 300 clients (100 per region). The results are shown in Figure 7. As the figure shows, the time to range scan increases linearly with the size of the scan. However, the average completion time for the whole range scan is much higher when there are more clients running. Range scanning is a fairly heavyweight process, and with more range scans occurring concurrently, all of the system resources (disk, CPU and bandwidth) become overloaded.

6. LONG TERM VISION

PNUTS was designed from the ground up to support a variety of advanced functionality, such as indexes over data. However, building a complex system required us to build in phases, and we have first built the basic table storage and consistency layers described in this paper. In this section, we describe some advanced functionality under design and development; the goal is to support them in the production platform over the next year or two.

6.1 Indexes and Materialized Views

In order to support efficient query processing, it is often critical to provide secondary indexes and materialized views. In our system, indexes and views will be treated equivalently; an index is just a special case of a view that provides efficient look up or range scans on secondary attributes of the base table. Our indexes and views will be stored as regular ordered tables, but are asynchronously maintained by the system. An **index/view maintainer** will listen to the stream of updates from message broker, and generate corresponding updates. For example, if a user moves from Wisconsin to California, and we have an index on location, the maintainer will delete the Wisconsin index entry for the user and insert a California index entry for the user. Further research is needed to examine the semantic implications of answering queries using possibly stale indexes and views.

6.2 Bundled Updates

Several customers have expressed a need for an extension of the consistency guarantees we provide (Section 2.2). The extension, called *bundled updates*, provides atomic, non-isolated updates to multiple records. That is, all updates in the bundle are guaranteed to eventually complete, but other transactions may see intermediate states resulting from a subset of the updates. For example, if Alice and Bob accept a bi-directed social network connection, we need to update both Alice's and Bob's records to point to the other user. Both updates need to complete (and the application writer

would prefer not to check and retry to ensure this, as in the current system) but it is not critical to provide serializability; it is ok if Alice is temporarily a friend to Bob but not vice-versa. The challenges in implementing bundled updates are to ensure the timeline consistency guarantees described in Section 2.2 when the updates in the bundle are asynchronously and independently applied, and to provide a convenient mechanism for the client to determine when all updates in the bundle have completed.

6.3 Batch-Query Processing

Although PNUTS is optimized for web OLTP workloads, we believe that it can also serve as a data store for batch and bulk processing, such as that provided by MapReduce or Pig. This requires further investigation of how a scan-oriented bulk workload interacts with a seek-oriented serving workload. It may be necessary to separate PNUTS replicas into "batch" and "serving," and optimize them separately for the different workloads. Also, parallel batch systems optimize their execution based on the current location of data, and therefore we may need to provide hooks for accessing tablets directly, bypassing routers.

7. RELATED WORK

Distributed and Parallel Databases

Distributed and parallel databases have been around for decades [13]. Much of the work has focused on distributed query processing [19] and distributed transactions [16]. While our thinking has been heavily influenced by this work, our simplified query language and weakened consistency model means that many prior techniques are not applicable.

Recent interest in massively scalable databases has produced several systems, each optimized for a different point in the design space. Google's BigTable [8] provides record-oriented access to very large tables, but to our knowledge there have been no publications describing support for geographic replication, secondary indexes, materialized views, the ability to create multiple tables, and hash-organized tables. Amazon's Dynamo [12] is a highly-available system that provides geographic replication via a gossip mechanism, but its eventual consistency model does not adequately support many applications, and it does not support ordered tables. *Sharding* is a technique for partitioning a database over a large number of commodity database machines, and is used, for example, in the data architecture for Yahoo!'s Flickr website. However, sharding systems do not typically provide automated data migration between machines or shard splitting, both of which are needed to minimize operational cost. Other large scale distributed storage systems include Amazon's S3 and SimpleDB services, and Microsoft's CloudDB initiative, but there is little information publicly available about the architecture of these systems.

Distributed Filesystems

Distributed filesystems are another option for large scale storage. Examples include Ceph [27], Boxwood [20], and Sinfonia [3]. These systems are designed to be object stores rather than databases. Boxwood does provide a B-tree implementation, but the design favors strict consistency over scalability, limiting the scale to a few tens of machines. Similarly, Sinfonia provides a simplified transaction type called "minitransactions" which are based on an optimized two-phase commit protocol. Even this optimized protocol requires communication among participants and limits scalability. PNUTS aims to provide much richer database functionality at scale than these systems.

Distributed Hash Tables

Widely distributed hash tables (DHTs) based on peer-to-peer architectures include Chord [26], Pastry [24] and a host of other systems. While DHTs themselves provide primarily object routing and storage, higher level filesystems [9] and database systems [18] have been built on top of them. These systems do not provide an ordered table abstraction, and focus on reliable routing and object replication in the face of massive node turnover. Since PNUTS runs in managed datacenters, we can focus instead on high performance and low latency using simpler routing and replication protocols.

Data Routing

Distributed data systems require a mechanism for finding data that has been spread across hosts. Several investigators have argued for using a hash function that automatically assigns data to servers [14, 28]. We decided to use the direct mapping approach instead for two reasons. First, in an ordered table, some tablets may become hotspots (e.g., tablets holding the most recent data in a date range). We need the flexibility provided by direct mapping to move hot tablets to underloaded servers. Second, we need a router layer to abstract away the actual location of data from clients so that we can change the servers or data partitioning without impacting clients. Since we need a routing layer anyway, using direct mapping adds no additional network latency.

Asynchronous Replication and Consistency

While serializability can be provided in distributed databases using two-phase commit protocols [16] or optimistic concurrency control [6], full serializability limits scalability [17]. At our scale, serializability is just not feasible, and so we provide a weaker, but still meaningful, consistency model (record-level timeline consistency).

Asynchronous replication has been used in other systems to provide both robustness and scalability. Master-slave replication is the canonical example [22, 10]. PNUTS extends master-slave replication by making the granularity of mastership be the record, by permitting mastership to change temporarily or permanently, and by ensuring timeline consistency (but not stronger semantics) despite storage server failures or mastership changes. Epidemic replication [23] is another example, and variants of epidemic replication are used in gossip systems such as Dynamo [12]. Epidemic replication usually provides "eventual consistency," where updates are committed in different orders at different replicas, and replicas are eventually reconciled. In this model, it is possible to read updates that will later be rolled back during reconciliation. In the default timeline consistency model of PNUTS, such "dirty data" is not visible, which is a key requirement for many of our applications. PNUTS does provide a relaxed consistency mode that is similar to eventual consistency for availability reasons, and applications can choose which mode they want.

8. CONCLUSIONS

PNUTS aims for a design point that is not well served by existing systems: rich database functionality and low latency at massive scale. While developing PNUTS, we have had to balance several tradeoffs between functionality, performance and scalability. In particular, we have chosen asynchronous replication to ensure low write latency while providing geographic replication. We developed a consis-

tency model that provides useful guarantees to applications without sacrificing scalability. We built a hosted service to minimize operations cost for applications, and focused on automated tuning, optimization and maintenance of the hosted service. We also limited the features to those that were truly needed and could be provided preserving reliability and scale. These design decisions have resulted in several novel aspects of our system: per-record timeline consistency to make it easier for applications to cope with asynchronous replication; a message broker that serves both as the replication mechanism and redo log of the database; and a flexible mapping of tablets to storage units to support automated failover and load balancing. The resulting system provides a useful platform for building a variety of web applications, as demonstrated by the applications that have already been built, or are in development, using PNUTS. Experimental results show that the system performs well under a variety of load conditions, and that our ordered table implementation provides high performance for both individual record lookups and range scans.

The first version of our system has entered production, serving data for some of Yahoo!'s social applications. This production instance uses hash-organized tables. Notably, ordered tables are not yet provided. While we have a first implementation of this functionality, and conducted experiments comparing ordered and hash tables (see Section 5), ordered tables will only enter production later this year. Other described functionality is currently implemented in a simplified form. For example, our load balancer currently follows a simple policy of keeping the number of tablets roughly equal on all storage units, but does not account for varying load between tablets. As we continue to add enhanced functionality, such as indexes, PNUTS will expand its role as the hosted data serving platform for Yahoo!'s web applications.

9. REFERENCES

[1] Eventually consistent. http://www.allthingsdistributed.com/2007/12/-eventually_consistent.html.

[2] Trading consistency for scalability in distributed architectures. http://www.infoq.com/news/2008/03/ebaybase, 2008.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[4] P. Bernstein, N. Dani, B. Khessib, R. Manne, and D. Shutt. Data management issues in supporting large-scale web services. *IEEE Data Engineering Bulletin*, December 2006.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proc. VLDB*, 1980.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. INFOCOM*, 1999.

[8] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[9] F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, 2001.

[10] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proc. VLDB*, 2006.

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

[12] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[13] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *CACM*, 36(6), June 1992.

[14] I. Stoica et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. ACM STOC*, 1997.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. SOSP*, 2003.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[17] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2007.

[18] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proc. VLDB*, 2003.

[19] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[20] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

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

[22] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *VLDB*, 1999.

[23] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP*, 1997.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[25] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proc. SIGMOD*, 2008.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM*, 2001.

[27] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. OSDI*, 2006.

[28] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proc. Supercomputing (SC)*, 2006.