PNUTS to Sherpa: Lessons from Yahoo!'s Cloud Database

Brian F. Cooper
brianfrankcooper@gmail.com

P.P.S. Narayan
ppsnarayan@gmail.com

Raghu Ramakrishnan
raghu@microsoft.com

Utkarsh Srivastava, Adam Silberstein, Philip Bohannon,
Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, Ramana Yerneni [1]

ABSTRACT

In this paper, we look back at the evolution of Yahoo!'s geo-replicated cloud data store from a research project called PNUTS to a globally deployed production system called Sherpa, share some of the lessons learned along the way, and finally, compare PNUTS with current operational cloud stores.

PVLDB Reference Format:
B.F. Cooper, P.P.S. Narayan, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS to Sherpa: Lessons from Yahoo!'s Cloud Database. *PVLDB*, 12(12) : 2300 - 2307, 2019.
DOI: https://doi.org/10.14778/3352063.3352146

1.INTRODUCTION

The vision of PNUTS in 2006 was to design and build a massively parallel, globally distributed database service for managing application state at internet scale, delivered as a cloud service for teams building Yahoo! applications. It was part of a broader evolution of the data ecosystem at Yahoo!, going beyond conventional relational database systems, to support three distinct scenarios:

- A highly available, globally replicated operational store to support the company's internet scale applications,
- Cost-effective storage of both application data and observational data (web logs, telemetry), and
- Extensive analytics to drive business decisions and improve operational efficiencies.

PNUTS addressed the first scenario. In addition, the company invested in building an object store called MObStor [1] (e.g., for holding attachments from Yahoo! Mail) and of course Hadoop [2] for analytics. The production version of PNUTS and Hadoop could be thought as Yahoo!'s operational database and data warehouse, respectively.

At the time we set out to design and build PNUTS, Yahoo! relied on a specialized backend service for managing user logins and on relational databases for other applications. There was an acute need for an operational store that was supported as a cloud service for developers to build on that offered sufficient flexibility to meet the needs of Yahoo!'s wide range of applications. The following were among the top asks : (1) A highly available cloud service rather than software that developers needed to host themselves, (2) low-latency geo-replication to handle Yahoo!'s global user base (which often traveled, requiring dynamic re-affinitization of related data across regions), and (3) simple semantics for updates with some user-control (e.g., changes to an object appear consistent; a writer always sees the result upon a subsequent read).

Surprisingly, there was willingness to sacrifice some powerful RDBMS features in exchange for such a service, especially given cost and performance considerations. Examples include ACID transactions across multiple objects and complex join queries. Therefore, we designed PNUTS to be a No SQL service, and this proved to be a good choice for Yahoo!'s operational needs.

Over time, as the service matured, we got requests for additional features such as sorted tables and secondary indexes, which suggest that the RDBMS feature set is indeed useful; it is simply the case that in many scenarios, developers are willing to trade some features for cost, performance and availability. (We discuss our learnings further in Section 4.3.)

Building a complex system required us to approach it in phases, and below we describe the journey of design, implementation and evolution of PNUTS over several years. We then reflect upon lessons learned and briefly look at the current landscape of public cloud data serving services. The productionized version of PNUTS is called "Sherpa," and we use the two names interchangeably throughout the paper.

2.DATA MODEL AND OPERATIONS

In this section, we outline the data model and operations supported in PNUTS.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352146

[1] Current affiliations: Apple—P.P.S. Narayan, Nick Puz; Facebook—Daniel Weaver; Google—Brian Cooper, Adam Silberstein, Utkarsh Srivastava, Ramana Yerneni; Grail—Philip Bohannon; Microsoft—Raghu Ramakrishnan; U. Toronto—Hans-Arno Jacobsen

2.1.Data Model

PNUTS supports a basic key-value data model in which values are simply records as in a traditional RDBMS. Records are automatically stored in partitions, which are re-balanced by the system for scale-out, and users can control data organization by optionally specifying a composite key for sorting a given record collection (table). Secondary indexes can also be specified for efficient scatter-gather operations.

2.2.Transactions

Transactions in PNUTS provide strongly consistent reads and writes to a single record. Each record goes through a time-ordered sequence of versions, where each write (insert, update or delete) creates a new version. All readers of the record will observe the same sequence of versions, regardless of which replica they read. We call this model "timeline consistency." We provide no guarantees about cross-record consistency.

The architecture supports a generalization of timeline consistency in which we allow a group of records identified by a common key value to be transactionally modified. Intuitively, all records in such a "record group" would be in the same physical partition and share the same timeline. This generalization was not implemented in Sherpa.

We chose the record timeline consistency model because it fit our requirements, while still providing high performance and scalability. The use cases we had for PNUTS could be satisfied with transactions updating a single record. For example, a user metadata database only requires updates to a single user's metadata at a time; a shopping listings database only requires updates to a single listing at a time.

The model provides scalability, because we can always add more independent computing resources to scale up the number of independent transactions we could execute. And by avoiding expensive protocols like two-phase commit, the model provides high performance.

When we were designing PNUTS, many replicated database systems provided either:

- *Full database consistency:* Transactions can read and write multiple records with ACID guarantees.
- *Best effort consistency:* Replicas were consistent in the common case, but could diverge even under normal (e.g., non-buggy) operation.

Timeline consistency provides a useful intermediate guarantee. Writes to a single record are always consistent, unlike best effort consistency. But the complex machinery and performance cost of providing full database consistency can be avoided.

As we built the system, we realized that some use cases could benefit from eventual consistency. These use cases had tighter requirements for write latency, and were able to tolerate "last writer wins" semantics in the case of concurrent writes. Thus, we added a mode that was weaker than timeline consistency: replicas may diverge from a "canonical" timeline of versions, but would eventually synchronize and store the same value.

2.3.Reads

The read API allows (indeed, forces) the client to make an explicit trade-off between latency and freshness. In particular, all reads are "consistent" (because they return a valid version from the canonical version timeline for the record). However, a client can choose to accept a potentially stale value in return for lower latency. The API provides three kinds of reads:

- *Read-latest:* returns the latest version of the record that existed at the time the read was initiated.
- *Read-critical:* returns a version of the record that is no staler than a client-specified version number.
- *Read-any:* returns any version of the record.

PNUTS records are geographically replicated, with one replica designated as the master (more details are in Section 3.3). A read-latest call must read from the master, since only the master knows for sure what the latest version is. If the master is far from the reader, the network propagation delay adds to the latency of the read. In contrast, read-critical can often read from the local replica, reducing the chance that a wide-area network hop is needed. Read-any can always read from the local replica.

2.4.Writes

There are two kinds of writes in the system:

- *Blind writes:* A write that always succeeds.
- *Test-and-set writes*: A write that only succeeds if the latest version of the record matches the version specified by the client.

Again, we have exposed to the client a trade-off between performance and transactional guarantees. Blind writes always succeed on the first try, but it is possible that your update is overwritten by another client writing concurrently. This is perfectly acceptable for many use cases; for example, a user updating their password in the user database is likely only performing one of these transactions at a time.

In contrast, many transactions need read-modify-write semantics: the data written depends on a previous read. An example is reading, and then incrementing, the view count on a shopping listing. Test-and-set writes allow clients to implement read-modify-write transactions by following these steps:

1. Read a record, including its version number V
2. Test-and-set write the record, specifying V

Step 2 will fail, effectively causing a transaction abort, if another write intervenes between step 1 and step 2. This optimistic concurrency mechanism does not require database locks, which helps performance by avoiding lock contention. However, as with all optimistic concurrency schemes, if there is high write contention, performance will decrease as many transactions have to abort in step 2 and be retried.

3.SYSTEM IMPLEMENTATION

We now describe the architecture and implementation of the system. The primary requirements that drove our design choices were:

- Elastic scalability

- Geographic replication
- Low latency reads and writes
- Strong, record-level consistency

3.1.Storage in a Single Data Center

In order to achieve scalability, we partition a database table into "tablets," each containing a set of records We then scatter these tablets across many independent servers, called "storage units," (SUs) in a data center. Reads and writes for a record are served by the storage unit that hosts the tablet containing that record. Storage units consist of an RDBMS database (originally InnoDB/MySQL) stored on locally-attached disk, and a software layer that implements our read and write protocols and interacts with MySQL storage. Each storage unit can hold many tablets from diverse tables. Because our workloads are easily partitionable along record boundaries, we achieve high scale simply by having many storage units.

Next, to make this scalability elastic, we make it possible to move tablets between storage units. If we need to increase the serving capacity of the system, we can add more storage units, and move some of the tablets to the new servers. Elastic scalability only works if we can alleviate hotspots: adding servers has limited value if most operations still hit a single server. Thus, we make the assignment of tablets to servers completely arbitrary (rather than, say, using a deterministic hash-based assignment). Thus, if a storage unit is overloaded, we move some of its tablets to less loaded servers to spread out the hotspot. We also split a tablet into smaller partitions if multiple hot rows are in the same tablet. The decision of which servers host which tablets is made by the "tablet controller," which is a singleton server (with a warm standby) in each data center.

The arbitrary assignment of tablets to storage units means we need a way to route requests for a record to the correct storage unit. This is accomplished by adding a layer of "routers" that know the mapping from record keys to tablets, and from tablets to servers. A client that wants to read or write a record contacts a router, which looks up the tablet and storage unit in its internal data structures and forwards the request to the appropriate server. The mapping of tablets to servers is determined by the tablet controller, and the routers merely cache a copy of this mapping.

Tables can be configured as hash or ordered tables. For hash tables, we divide the space of hash values into intervals, and each interval corresponds to a tablet. We hash each record key and find the interval that the key hash falls into; that is the tablet that owns the record. Ordered tables are implemented in a similar manner, except without hashing: tablets correspond to intervals of the key space, and the interval containing a key determines the tablet that owns the record. In this way, we can use the same infrastructure for defining tablet boundaries, splitting tablets, and assigning keys to tablets for both hash and ordered tables.

3.2.Replication

So far, we have described the storage of records in a single dat center. In fact, tablets (and thus records) are replicated to multiple data centers. This geographic replication is achieved by log-shipping: a storage unit writes updates for a tablet to a write-ahead log, which is then forwarded to servers in other data centers hosting replicas of that tablet. We decided to implement our write-ahead log using a pub/sub service that is separate from the storage unit. This service is called the Yahoo! Message Broker (YMB). We used a separate pub/sub service for a few reasons:

- YMB provides reliable publishing and delivery of messages. It does this by storing copies of published messages on multiple servers in the data center. Thus, we ensure that redo records in the log are fault tolerant without having to make the storage unit itself more complex.
- The replication mechanism is a natural fit for pub/sub, and thus it made architectural sense to separate serving (e.g., the storage unit) from replication (e.g., YMB).
- Separating the write-ahead log from the storage unit allows us to deal with some consequences of storage unit failures by writing directly to YMB, as described in the next section.
- We were able to repurpose an existing Yahoo! pub/sub infrastructure for the first version of the YMB, accelerating development of the system.

Because of our desire for low-latency transactions, we elected to consider writes committed once they had been acknowledged by the local YMB, rather than waiting for the actual replication to other data centers to occur. This is in contrast to some other systems [3,4] that consider writes committed only after a wide-area consensus protocol (like Paxos [5]) had completed among multiple data centers. The trade-off we made was to accept that some catastrophic failures (like a data center burning down) would lead to data loss, in return for lower write latency in the common case.

The system provided a "selective replication" option both for tables and records. Tables need not have copies in all data centers or replicas. Depending on the needs of the application a table is replicated to at least two data centers. And over time this replication footprint could be changed, administratively, to add or remove data centers.

Similarly, some records simply wouldn't be replicated to some data centers. This allowed us to save resource costs: if a particular record was never accessed from a region, we would not pay the bandwidth and storage costs of keeping an up-to-date copy in that region. Also, for regulatory reasons, some data cannot be stored in certain places, and selective replication allowed us to support that.

3.3.Record Masters

Our timeline consistency model is implemented by assigning a "master" copy on a per-record basis. For example, the master of Brian's record may be in California, while the master of Raghu's record is in Washington. The master for a record can change over time: if Raghu goes on a business trip to England, and the application writes to his record now come from his new locality instead of Washington, PNUTS will automatically move the mastership (using a robust, fault-tolerant protocol).

Mastership is recorded in the record itself in a hidden field. When a storage unit receives a request to write a record, it can only execute that write if it is the master. Otherwise, the write is forwarded to the actual master. Similarly, storage units decide whether they can serve read-latest reads based on whether they are the master.

Changes to the mastership use the same write mechanism as normal writes. That is, to change mastership, the master writes the identity of a different replica in the "mastership" field of the record. All replicas, including the new master, will see this update and thus

learn of the change. In theory, this reduces the complexity of the system, since we only need a single mechanism for both writes and mastership changes. In practice, as we implemented the system, we found that handling failures forced us to re-introduce some complexity. If a storage unit hosting a replica had a hard failure, such that it could no longer commit writes, then writes to the record would be blocked until we forced a mastership change to another replica. This was accomplished by having the tablet controller write a mastership change directly to the YMB write-ahead log. Doing this properly is hard, because it requires correctly detecting that the storage unit has failed and will not come back to life and start committing writes again.

In order to provide primary key constraints, we need a master for records that do not yet exist. That master will ensure that test-and-set writes inserting a record for the first time are correctly executed. Since the record does not exist, we cannot use the record to store the master's identity. Instead, mastership is assigned at the tablet level. Replicas of a tablet store the same data, and we assign one tablet in each set of replicas as the master for any newly inserted records in that set. Tablet mastership can also change. As with record mastership, a hard failure of a storage unit results in the tablet controller forcing a mastership change by writing directly to YMB.

3.4.Failure handling

We now summarize how the system deals with different types of failures.

3.4.1.Storage unit failures

Storage units can fail in a variety of ways. The machine could experience a fail-stop failure, like a crashed server process or a failed power supply. A disk could become corrupted. Faulty RAM could corrupt in-memory state. For most of these failure modes, we rely on standard single-server fault tolerance: RAID arrays to tolerate loss of a single disk, checksums to deal with memory corruption, and so on. However, a key principle of the system is that it survives complete machine failures, so data is replicated to another machine in another data center. Whenever there is a full storage unit failure, reads can fail over to the other copy, and (after a record mastership change) so can writes.

3.4.2 YMB broker failures

The YMB can also experience failures, either in the server (a crash) or in the on-disk storage. Again, we utilize redundancy. Unlike the storage unit, there is no easy way to replicate YMB state across data centers, since it is the YMB itself that is carrying out the replication task. As a result, we replicate at the disk level: each YMB update is written to multiple disks on multiple servers. Then, an individual YMB machine failure is not sufficient to cause a committed transaction to be lost.

3.4.3 Whole data center failures

Occasionally, entire data centers fail, for example due to a natural disaster or network outage. One reason to replicate data to multiple data centers in multiple geographic regions is to survive such a large-scale event. A natural disaster which devastates a geographic region can be tolerated because data has been replicated elsewhere on the globe. One limitation of our approach is that a transaction is considered committed when it is written to the YMB in the same data center as the storage unit where the write originated. This means that transactions which had not yet been replicated outside the data center may be lost in the case of a whole data center failure. We made this trade-off because data center failures are rare, and committing only to a YMB in the same data center reduces write latency.

3.4.4 Lost pub/sub messages

It is possible that a message sent by the YMB fails to arrive at the destination broker. For example, a network issue might cause the message to timeout, or the receiving broker may crash before persisting the message. Because the YMB stores messages locally until receiving positive acknowledgement of delivery everywhere, we can recover from a lost message by simply redelivering it.

4.EVOLUTION FROM PNUTS TO SHERPA

The PNUTS system was described in two 2008 papers—the system overview [6] and the description [7] of sorted tables, one of the distinctive features of the system. Subsequent PNUTS papers reported on parallelizing range queries [8], view maintenance [9], user-controlled geo-replication [10], and a benchmark designed to evaluate cloud-serving systems such as PNUTS [11].

On June 1st, 2009, we announced the launch of the production version Sherpa [12], going public with Yahoo!'s investment in a globally replicated hosted NoSQL cloud service. The architecture of Sherpa was, and even today is, closely aligned with the PNUTS paper [6]. The loosely coupled architecture of the storage units (SU), routers, tablet controller and messaging system was well-suited for horizontal scalability at the various layers. Application engineers used the client SDKs to connect and perform basic RESTful operations of Get (read), Set (write, insert), Delete and Scan. Version 1.0 of Sherpa supported both timeline consistency and eventual consistency for distributed hash tables.

4.1.The Journey

In 2010, we announced [13] support for distributed ordered tables, described in [7], so that applications can access a set of records based on a primary key prefix (e.g., retrieve the last 10 status updates by user=ppsn). Under the covers, ordered tables were partitioned by primary key ranges rather than the hash ranges of the primary key. While hashing has a nice property of uniform distribution, primary key based partitioning leads to imbalance of load and size of tablets. Therefore, it was imperative that we built a load balancer, which to this day, automatically balances the space utilization and IO load of the SUs across a cluster. Each SU keeps track of the "heat" metric for itself and its tablets based on measurements like latency, requests, # of records, and size of data. The load balancer used this heat metric to move tablets from hot SUs to cold ones, keeping the overall heat in the system near the average. The tablet move algorithm enabled balancing without any impact to applications requests for the data. The load balancer could also perform splits or merges on tablets as another option to balance the heat metric.

Within two years of launch, Sherpa was storing and serving data for properties and applications within Yahoo such as yahoo.com, Yahoo! News, Mobile, Social, Video, Sports, and Shopping. As bigger applications started using Sherpa, we decided to build isolation mechanisms into the multi-tenant hosted architecture. With the introduction of SU banks, tables could be isolated to a subset of SUs, such that applications with certain read-write patterns would not negatively impact smaller applications in the cluster.

One of the original use cases envisioned for PNUTS was user data storage. Adhering to local legal and jurisdictional considerations for global applications with global users meant building the right

data governance knobs. Selective record replication was an important feature we introduced for applications to choose where to store full copies of records dynamically. In 2011 [14], Sherpa became the de facto store for all new user data in Yahoo which required data governance rules.

The original design of the system did not support versioning, and primarily relied on multiple copies of data to recover from disk, node, cluster or data center failure. We soon realized that it was equally important to protect the data from inadvertent bugs introduced either by Sherpa engineers or application engineers that may accidentally corrupt or erase data. In 2011 [14], we introduced the capability to perform regular table backups to off-line storage, from which applications could restore records or tables to a point in time.

By 2012, we introduced secondary indexes with materialized fields that enabled applications to go beyond simple primary key lookup queries. The indexes were maintained asynchronously, so that application perceived latency for updates were fast. While the ideas were novel, index maintenance added a layer of complexity to our consistency algorithms, and also made it complex for application engineers to reason with CAP trade-offs during reads.

If we had to call out one pivotal feature of Sherpa in the last 10 years, it would be providing a self-service portal to the Yahoo application engineers. Up until 2014, provisioning of tables and applications on the multi-tenant platform was via a manual on-boarding process, after a review of the application workload and careful capacity planning. In late 2014, inspired by external cloud providers, we decided to make the application developer experience frictionless, and opened up Sherpa usage with a click of a button to anyone inside Yahoo. In the six months after the launch of the portal to the company, we witnessed a 300% increase in tables being created and used in production.

Self-service provisioning required tracking usage for chargeback. Additionally, we decided to establish and implement quotas, and build throttling for protection from rogue applications. In 2015 [15], we launched a distributed and decentralized rate limiting solution which could scale to thousands of SUs, and respond to changing traffic patterns in real-time, while adding only a few microseconds per API request.

When we initially built Sherpa, each local SU was based on InnoDB/MySQL. In later years, two factors motivated us to look beyond MySQL. First, we noticed a demand for varying read-write access patterns. The high read workload was changing and the percentage of write workload was increasing. Second, both hardware and software technology was evolving, with interesting solutions in throughput/latency trade-offs for varying query workloads. Given our pluggable architecture of storage engines, we experimented and in 2015 [16] replaced InnoDB with RocksDB [17], which was able to support our write heavy workloads with lower cost, and at the same time provide similar latency characteristics.

Over time we encountered applications requiring lower latency SLAs, in the range of 1-2 millisecond response time for reads, much lower than the typical 10-20ms provided by the initial implementation of Sherpa. In 2015, we deployed a low-latency (LL) tier of the service, which was used to launch new advertising initiatives such as Gemini at Yahoo. Based on PCIe Flash SSDs, the LL tier had 5x lower 99%ile read latency, and was 6x cheaper than public cloud NoSQL implementations. Our LL implementation was the second big architectural change from the original PNUTS design, since we moved the routing logic directly into the client SDK, thus reducing a network hop for requests.

Finally, one of the key contributions of the PNUTS/Sherpa program was furthering the technology and innovations on distributed pub/sub messaging. Since PNUTS, Yahoo has built and open-sourced key pieces of distributed computing such as Apache ZooKeeper and Apache Bookkeeper. After years of research, development and testing at scale, we replaced the original messaging system for Sherpa with Pulsar, which was built using Zookeeper and Bookkeeper. Yahoo open sourced Apache Pulsar in 2016 [18], which is also used internally at Yahoo as a messaging service for applications across many properties such as Mail, Finance, Sports, and Advertising.

4.2.Scale and Numbers

The first deployment of Sherpa was in two US data centers, and each region had 25 storage units. Over the last ten years, this has grown tremendously. Sherpa now runs in 6 Yahoo data centers around the world, with two or more availability zones within each, thus leading to more than 14 replicas. The total number of storage units, as last known to the authors, exceeds 5000.

Almost every major web and mobile property at Yahoo depends on Sherpa as a globally distributed serving store. The tables have more than a trillion records in aggregate [16] and several PB of storage. Given the growth in usage, the transaction rates have peaked at more than 3 million reads per second, and hundreds of thousands of writes per second.

Running, deploying, upgrading and maintaining a system at this scale requires operational expertise developed over the years. As last known to the authors, the Sherpa team follows an agile development methodology with release milestones every six weeks. To contrast, Sherpa had just two releases six months apart in 2009.

The agility in building and deploying such a large-scale system was achieved through significant investment in testing and automation. Integration testing of all Sherpa components, including validating performance and scale, is done in under 12 hours. The tests include cross-data center replication and data integrity tests.

As new code is rolled out, in an automated but carefully phased approach, deployment scripts monitor for performance regressions and error rates. During upgrades, applications automatically get redirected to replica copies in the same geographic zone, thus maintaining high availability. A successful deployment of new software across the 6 data centers and thousands of servers is done in under 3 days. New features are typically rolled out with feature flags, and all changes are backwards compatible.

4.3.Learnings

Looking back after ten years, our vision of building PNUTS and Sherpa was successful because we failed many times along the way. With each failure, we heard the developer needs, saw the evolving use cases, observed the changing market trends, and hardened the system. The learnings allowed us to pivot our focus in making the system bigger, faster, better and richer in features.

4.3.1.Make it Frictionless

"It is important to remember your competitor is only a click away"

As mentioned earlier in this paper, having a frictionless self-service portal and dashboard was a fundamental adoption and growth

driver for Sherpa within Yahoo. Public cloud services such as AWS, Azure and GCP have made it easy for applications engineers to use their platforms. These engineers are setting the same ease of use expectations for internal enterprise platforms.

4.3.2.Design for your Customer Base

"Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple"

Building a large distributed system is fun, innovative and highly challenging. The urge to build unique and differentiated solutions ends up hiding the long-term costs of maintaining complexity. We spoke to engineers across Yahoo! early in the project, and the overwhelming ask was for a highly available, durable, low latency, geo-replicated serving system. We listened to this input, and some of our best decisions were the features we chose *not* to build—e.g., support for complex joins or cross-partition transactions—in order to focus on the core requirements.

Over time, requests for additional features were steady, and we added support for data versions, secondary indexes, sorted tables, etc. That said, we didn't always make the right trade-offs. Some features we invested in (e.g., selective record replication), while unique, did not get high adoption due to complex integration challenges and simpler alternatives (e.g., separate table per region.)

The choices we made for building blocks also influenced our thinking heavily, and arguably kept us from exploring avenues our customers would have benefited from. Our storage units were initially based on MySQL since our focus was on distributed aspects of the system rather than the individual nodes. We were careful to make the choice of local engine pluggable and this later allowed us to replace MySQL with RocksDB. However, we never explored the space of richly structured values in our key-value model of data. Given the huge success of Mongo, Cosmos DB, etc., it is surprising that we did not consider this dimension; it would likely have resonated well with many of the developers building on Sherpa. Arguably, the choice of a relational SU, even though it was pluggable, kept us from thinking about non-relational abstractions for the data model.

4.3.3.Building is Hard, Operating is Harder

"Events are called inevitable only after they have occurred"

Durability was a core principle while designing PNUTS and building Sherpa. When using Sherpa to store data, application engineers expected that the data is always accessible, and there is no data loss. Over the years we realized that data loss was inevitable. Data loss can be caused by a variety of reasons: failures, bugs in code, incorrect protocols, process oversights, missing tests, human error, data center or network outages. As the system matured, we built many tools and capabilities that made it possible for us to detect data loss/inconsistencies (e.g., offline record sampling and compare) and then fix (e.g., backup/restore) it automatically with little or no operator intervention.

4.3.4.Fail Fast and Make Different Mistakes

"Success is not final, failure is not fatal"

As Sherpa grew in usage and global footprint, we discovered bugs and protocols that were not robust. For example, we had designed and implemented an asynchronous reconciliation protocol via the messaging system. This broadcast protocol would enable storage units to find the correct latest version of the record in case of late or lost messages. However, we discovered a flaw in the protocol that led to flooding of our messaging system when it was already in backlogged situations, thus causing a self-inflicted DoS attack. We corrected the protocol by making it out-of-band rather than using the messaging system.

4.3.5.Think Big

"… and dream bigger"

In hindsight, this is perhaps the biggest lesson of all. Early on, we realized the value of having a geo-replicated operational cloud database, especially with the unique feature set (e.g., sorted tables, secondary indexes) that PNUTS supported. We had many conversations about making it available as a public cloud service, perhaps in conjunction with MOBStor, the internal storage service at Yahoo! for blobs. The discussions expanded to whether Yahoo! should get into public cloud computing. Unfortunately, none of these proposals came to pass. On a related note, the Sherpa journey was well under way before Yahoo! fully embraced open-source development, and the codebase had many dependencies on Yahoo! internal libraries. We never were able to create an open-source version of PNUTS. In hindsight, the impact could have been considerably more had we open-sourced the project or delivered it as a public cloud service.

5.CLOUD OPERATIONAL STORES

In recent years, we've seen rapid growth of cloud databases, including operational or transactional database systems such as PNUTS. The ones most closely related to PNUTS are Cosmos DB, Dynamo DB and Spanner. Cosmos DB and Dynamo DB are No SQL services like PNUTS, and Cosmos DB and Spanner offer native geo-replication support. All four of these services are designed from the ground up to be cloud-native, i.e., leverage benefits like elastic scalability of resources while tolerating various kinds of failure modes and latency/bandwidth challenges..

Aurora and SQL Hyperscale, while not focused on geo-replication, are noteworthy in that they embody cloud-native architectures designed to handle on-premise relational DBMS workloads.

5.2.Cosmos DB

Azure Cosmos DB [3] is a globally replicated key/value store. Cosmos DB is designed to allow customers to elastically and independently scale write and read throughput and storage across any number of data centers. It offers SLAs for throughput, latency, read and write availability.

Cosmos DB's concurrency model provides richer semantics than PNUTS, with five consistency models (strong, bounded staleness, session, consistent-prefix and eventual) and multi-item transactions that do not cross partitions (i.e., server boundaries).

Another notable feature of Cosmos DB is the rich underlying document model for values, with automatic schema-agnostic indexing. Like PNUTS, Cosmos DB does automatic sharding and load balancing, and offers strong support for geo-replication. Whereas PNUTS uses primary copy concurrency control, Cosmos DB uses nested consensus with dynamic quorums with a Bayou style multi-master replication protocol.

5.3.Dynamo DB

Amazon's Dynamo DB [19] is a single region replicated key/value store (i.e., the core replication protocol is scoped to a single region)

that can be configured across multiple regions by doing double writes. A defining feature of Dynamo DB is its "eventual consistency" model: if you write data, eventually all reads will observe that write. This is different than our timeline consistency model:

- *No bounded staleness:* Dynamo DB offers the option to read the latest written data, or to read stale data. But there is no way to specify a bound on the staleness of the data. PNUTS provides a read-critical operation that allows clients to decide how much staleness is acceptable.
- *No read-modify-write transactions:* All writes in Dynamo DB are blind writes. PNUTS provides a test-and-set write that can be used to implement per-record ACID transactions, as described above.

5.4.Spanner

Google's Spanner [4] is a geo-replicated relational database. There are several differences with PNUTS:

- *SQL:* Spanner's API is based on SQL. The PNUTS API provides CRUD operations.
- *Multi-item transactions:* Spanner provides ACID read-modify-write transactions that can involve any number of records in the database. PNUTS transactions are limited to a single record. While Spanner's transactions are more powerful than those in PNUTS, multi-record transactions can lead to unexpected performance problems since the risk of lock contention increases as the scope of the transaction increases.
- *Commits require WAN communication:* Spanner commits transactions by executing a multi-data center Paxos round. If the data centers are geographically distributed, this can lead to higher latencies than PNUTS transactions, which require only in-data center communication between the storage unit and YMB.

5.4.Aurora

AWS Aurora [20] is a relational DBMS based either on MySQL or PostgreSQL. Amazon provides more features, with the notable exception of geo-replication, but less scalability, than PNUTS.

Aurora provides the standard MySQL and PostgresSQL APIs, and support for all their operations, such as multi-record transactions and expressive queries in SQL. PNUTS provides a CRUD API with more limited reads and writes, but allows for tables to be sorted and sharded across multiple globally distributed locations, with near real-time update synchronization.

All operations in Aurora are confined to a single "cluster volume," which is a database that currently cannot grow larger than 64 terabytes (TB). PNUTS tables can grow to arbitrary size.

5.5.SQL Hyperscale

Microsoft's Azure SQL Hyperscale [21] is a relational DBMS based on SQL Server. Like Aurora, it provides more features than PNUTS, again, with the exception of global geo-replication.

Azure SQL provides all the capabilities of SQL Server (full transactional support, SQL queries, etc.), with a "Hyperscale" architecture that has no fixed size limits and has been tested at 100s of TBs.

6.RELATED WORK

The original PNUTS paper [6] lists a complete set of references. Here we highlight a few especially influential papers.

The Bigtable paper [22] introduced and popularized many concepts around very large-scale structured data storage as represented by the "NoSQL" movement. Many of the architectural principles of BigTable, such as the idea of tablets of data that can be moved between servers for load balancing, influenced the design of PNUTS.

The Dynamo paper [19] introduced the notion of a data store based on eventual consistency. Although our consistency model was based on the stronger notion of timeline consistency, we often discussed Dynamo during the design phase of PNUTS. In particular, we strove to understand the benefits, and costs, of adopting a consistency model that was stronger than eventual consistency. We also decided during the development of the system to support an eventual consistency mode as an option.

Distributed hash tables (DHT), like Chord [23] provided an alternative architecture for scalable distributed databases. Unlike PNUTS, which allows arbitrary assignment of tablets to servers, Chord uses a consistent hashing scheme to deterministically map data to servers. Very early in the design phase of PNUTS we considered a DHT-like model. However, we decided that the extra flexibility provided by arbitrary tablet assignment gave us advantages for load balancing and latency (as we could avoid searching the Chord ring for data).

7.KEY CONTRIBUTORS

PNUTS began as a Yahoo! Labs project, and quickly became a collaboration with the product team that led to the Sherpa production system. Contributors from Labs include Parag Agrawal, Phil Bohannon, Jianjun Chen, Brian Cooper, Sudarshan Kadambi, Nick Puz, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, Erwin Tam, Eric Vee and Ramana Yerneni. Rodrigo Fonseca, Hans-Arno Jacobsen and Ymir Vigfusson worked on PNUTS while visiting Yahoo! Labs, and Hector Garcia-Molina was at Stanford.

Many individuals in the Yahoo! product team, across engineering, infrastructure and operations organizations, were a core part of the team and contributed to building, deploying and running Sherpa. We would like to call out the key contributors to the Sherpa system.

In Engineering, the following were responsible for building the features listed in this paper: Andrews Albert, Kevin Athey, Craig Bair, Del Bao, Maurice Barnum, Subramanyeswara Bhavirisetty, Roger Bush, Jayadev Chandrasekhar, Rupesh Chhatrapati, John Corwin, Kevin Dalley, Amanveer Dhillon, Joe Francis, Zeke Huang, Mohsin Khan, Eun-Gyu Kim, Varad Kishore, Prashant Kumar, Rajesh Kumar, David Lomax, Mukund Madhugiri, Marco Mar, Patrick Marion, Brad McMillen, Matteo Merli, Masood Mortazavi, Monoreet Mutsuddi, Michi Mutsuzaki, P.P.S. Narayan, Chuck Neerdaels, Jothi Padmanabhan, Yi Pan, Varalaxmi Raveendar, Farooque Sayed, Priyanka Shah, Scott Simpson, Subbu Subramaniam, Prasant TR Rao, Adwait Tumbde, Aditya Umrani, Tao Wang, Daniel Weaver, Kenneth Yin and Michael Yuvrovitsky.

Defining the roadmap aligned to business needs, while managing the ever-growing expectations of the engineers using Sherpa was not an easy task. This job was performed by the team of product managers: Satheesh Nanniyur, Toby Negrin and Sambit Samal.

Running Sherpa at scale with astonishing growth, especially in the initial years when the system was still maturing, required true grit, determination, passion and persistence. The operations team members, past and present— Mohammed Abdurahiman, Brian Adams, Shabana Azmi, Lohith BK, Jennifer Davis, Vivian Fernandez, Mike Marino, David Pippenger, Ludwig Pummer, Smritidhara Saha, Venkatasubramanian Venkataraman — have kept the system humming and growing for the last ten years.

Finally, Sherpa had key executive support through the years. Usama Fayyad, David Filo, Prabhakar Raghavan and Jay Rossiter from the executive team were strong sponsors of the technology.

8.ACKNOWLEDGEMENTS

We thank all the users of Sherpa, especially those who drove the design with their insightful asks.

9.REFERENCES

[1] N. Joneja. MObStor: Yahoo!'s unstructured data cloud. Yahoo! Developer Network Blog https://web.archive.org/web/20100612203230/http://developer.yahoo.net/blog/archives/2009/07/mobstor.html, July 17, 2009.

[2] E. Baldeschwieler. Yahoo! launches world's largest Hadoop production application. Yahoo! Developer Network Blog. https://web.archive.org/web/20160307081144/https://developer.yahoo.com/blogs/hadoop/yahoo-launches-world-largest-hadoop-production-application-398.html. Feb 18, 2008.

[3] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M.G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. Levandoski and D. Lomet. Schema-agnostic indexing with Azure DocumentDB. PVLDB 8(12): 1668-1679, 2015.

[4] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W.C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang and D. Woodford. Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. 31(3): 8:1-8:2, 2013.

[5] L. Lamport. The part-time parliament. ACM TOCS 16.2 pp. 133–169, 1998.

[6] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H-A. Jacobsen, N. Puz, D. Weaver and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. PVLDB 1(2): 1277-1288, 2008.

[7] A. Silberstein, B.F. Cooper, U. Srivastava, E. Vee, R. Yerneni and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. ACM SIGMOD, 765 – 778, 2008.

[8] Y. Vigfusson, A. Silberstein, B.F. Cooper and R. Fonseca. Adaptively parallelizing distributed range queries. PVLDB 2(1), 682 – 693, 2009.

[9] P. Agrawal, A. Silberstein, B.F. Cooper, U. Srivastava and R. Ramakrishnan. Asynchronous View Maintenance for VLSD Databases. ACM SIGMOD, 179 – 192, 2009.

[10] S. Kadambi, J. Chen, B.F. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam and H.G. Molina. Where in the World is My Data? PVLDB 4(11): 1040-1050, 2011.

[11] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears. Benchmarking cloud serving systems with YCSB. ACM SOCC, 143 – 154, 2010.

[12] T. Negrin, Moving to the cloud. Yahoo! Developer Network Blog http://web.archive.org/web/20090605065217/https://developer.yahoo.net/blog/archives/2009/06/sherpa.html June 1, 2009.

[13] P.P.S. Narayan. Sherpa update. Yahoo! Developer Network Blog https://web.archive.org/web/20110126145148/https://developer.yahoo.com/blogs/ydn/posts/2010/06/sherpa_update/ June 8, 2010.

[14] S. Samal. Sherpa grows and scales in 2011. Yahoo! Developer Network Blog https://web.archive.org/web/20160729043838/https://developer.yahoo.com/blogs/ydn/sherpa-grows-scales-2011-50931.html Sep 16, 2011.

[15] V. Kishore and A. Sethuraman. Cloud Bouncer - Distributed rate limiting at Yahoo!. https://yahooeng.tumblr.com/post/111288877956/cloud-bouncer-distributed-rate-limiting-at-yahoo Feb 17, 2015.

[16] S. Nanniyur. Sherpa Scales New Heights. https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights June 4, 2015.

[17] RocksDB: A persistent key-value store for fast storage environments. http://www.rocksdb.org

[18] J. Francis and M. Merli. *Open-sourcing Pulsar, Pub/sub Messaging at Scale*, https://yahooeng.tumblr.com/post/150078336821/open-sourcing-pulsar-pub-sub-messaging-at-scale Sept 7, 2016.

[19] 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 SOSP, 205 – 220, 2007.

[20] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, X. Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. ACM SIGMOD, 1041 – 1052, 2017.

[21] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U.F. Minhas, N. Prakash, V. Purohit, H. Qu, C.S. Ravella, K. Reisteter, S. Shrotri, D. Tang and V. Wakade. Socrates: The New SQL Server in the Cloud. ACM SIGMOD, 1743 – 1756, 2019.

[22] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes and R.E. Gruber. Bigtable: A distributed storage system for structured data. ACM OSDI, 205 – 218, 2006.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM, 149 – 160, 2001.