A Cloud-native Architecture for Replicated Data Services

Hemant Saxena
University of Waterloo

Jeffrey Pound
SAP Labs, Waterloo, Canada

Abstract

Many services replicate data for fault-tolerant storage of the data and high-availability of the service. When deployed in the cloud, the replication performed by these services provides the desired high-availability but does not provide significant additional fault-tolerance for the data. This is because cloud deployments use fault-tolerant storage services instead of the simple local disks that many replicated data services were designed to use. Because the cloud storage services already provide fault-tolerance for the data, the extra replicas create unnecessary cost in running the service. However, replication is still needed for high-availability of the service itself.

In this paper, we explore types of replicated data services and how they can be mapped onto various classes of cloud storage. We then propose a general architectural pattern that can be used to: (1) limit additional storage resulting in monetary cost saving, (2) while keeping the same performance for the service, and (3) maintaining the same high-availability of the services and the durability guarantees for the data. We prototype our approach in two popular open-source replicated data services, Kafka and Cassandra, and show that with relatively little modification these systems can be deployed for a fraction of the storage cost without affecting the availability guarantees, durability guarantees, or performance.

1 Introduction

Infrastructure-as-a-Service (or, cloud) providers have become the defacto standard for deploying services of all kinds. However, migrating services engineered for bare metal servers to the cloud can come with varying degrees of difficulty, specifically when leveraging the highly available cloud storage. When these services get deployed on the cloud, they often end up using cloud storage similar to local on-premise storage, and this mismatch introduces inefficiencies in the system.

In this paper we focus on an important class of data management systems, we refer to as *replicated data services*. These services manage state (e.g. database state) and use replication for fault-tolerance and high-availability. They include a variety of back-end services used to build various applications that power the digital world; including replicated relational databases (e.g., PostgreSQL/XC), scalable key-value stores (e.g., Cassandra [20]), and ingest pipelines (e.g., Kafka [20]). Many of these services were originally engineered for *on-premise* deployments and share a common property of their monolithic architecture: they all manage their own disk. In particular, they manage their own copy of some shared state.

It is certainly possible to deploy an existing replicated service to the cloud without changes to the service itself. Cloud providers have gone to great lengths to make this an easy task. Storage services can be exposed as block devices or network attached file systems, giving the abstraction of the local disk our services were designed to manage. However, if we analyze the end-to-end architecture of this type of deployment there are two significant problems.

Redundant replication of storage: Cloud storage services provide fault-tolerance and high-availability using their own internal data replication. Replicated services also replicate data to provide the same properties. This additional application-level replication provided by the replicated service has little advantage over what the cloud storage already provides. For example, consider running multiple copies of a service within a single availability zone (AZ) to tolerate host failure or network partitions. If the storage service is available to all hosts within the AZ, then storing multiple copies of data within that storage service does not increase data availability. Furthermore, the storage systems themselves already guarantee durability of stored data under various types of failure scenarios. In some cases, the application-level replication is still needed. For example, if a storage service is not available across multiple AZs within a geographic region, then application-level replication is required to preserve data availability under AZ failure. However, if a storage service is available in all AZs, then storing multiple copies of data within that storage service again becomes redundant.

Storage service characteristics: Cloud provided storage services have significantly different performance characteristics compared to each other and the on-premise physical disks. For example, storage I/O latency for on-premise deployment (using local disk) is orders of magnitude lower than the I/O latency when using cloud storage. Data centric services, like RDBMSs and scalable key-value stores, have gone to great lengths to optimize I/O performance on local disks. Furthermore, different storage services have different availability properties. Some are only available to a single host at a time (e.g., AWS EBS [1]), some can be shared among hosts within a single availability zone or data centre (e.g., Google Cloud Filestore [15]), and others can be shared among hosts across availability zones in a geographic region (e.g., AWS EFS [2]). These availability properties can influence the architecture of a cloud-native replicated data service design, and can influence requirements on when data needs to be replicated in order to achieve particular fault-tolerance guarantees. Each cloud storage service presents a new class of storage device with its own unique performance and availability characteristics.

Addressing these problems poses some difficult challenges. To overcome the redundant replication problem, we need to reconsider the storage architecture of our replicated services in order to provide service availability without unnecessary data replication. While reconsidering the storage architecture, we must also consider the different performance characteristics of cloud storage services to best exploit their properties.

Contributions: In this paper, we provide a general classification of the replication approaches used by a selection of popular replicated services and analyze how the approaches fit the characteristics of the various types of storage services provided by the major cloud providers. We then describe how a well-known architectural pattern originally designed for efficiently handling mixed read/update workloads, the *main-delta model* [19], can be adapted to various classes of replicated services to solve the redundant replication problem when engineering these services for cloud deployment. We refer to the new architecture as *cloud-native replicated services*. The main-delta architecture naturally lends itself to the decoupled compute and storage model available in the cloud. Furthermore, this design allows us to tune the size of I/Os by adapting the policy a cloud-native replicated service uses to *merge* its deltas. This allows us to simultaneously address the performance problem of using cloud storage services.

We support our analytical solution by implementing it in two popular replicated services, falling into two very different categories of replication type. Apache Kafka (Section 3.1) and Apache Cassandra (Section 3.2). We find that the implementation overhead to adapt these existing monolithic architectures to our proposed cloud-native design is very small, requiring only a few hundred lines of Java code modifications.

2 Problem & Solution Overview

2.1 Problem of redundant replication

Replicated data services provide *application-level* replication of data for high read throughput, fault-tolerance, and high-availability. On the other hand, cloud storage provides *storage-level* replication of the data for the same reasons. When replicated services are deployed on the cloud the data replication quadruples due to the two independent levels of replication that become the part of the whole system. We call this the problem of *redundant replication*. In Figure 1, we demonstrate this problem in an example scenario where an application is using cloud storage, with application-level and storage-level replication factors of 3, and the data (e.g., key-value $(a,1)$) is stored nine times.

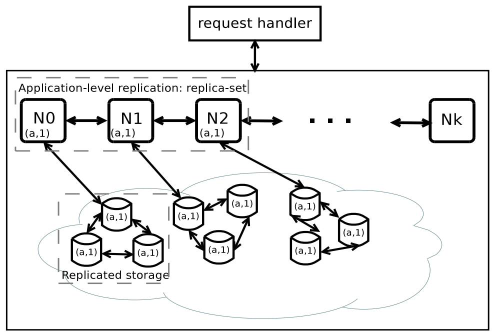

Figure 1: Present day architecture of replicated services on cloud.

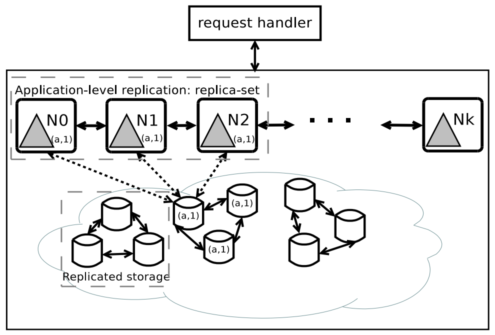

Figure 2: *Cloud-native architecture* of replicated services on cloud.

To solve the problem, we propose a solution that ensures *only one application-level replica of the data is stored on the cloud storage, while maintaining the high-availability guarantees of the systems.* Note that, our assumption is that this single replica is highly-available and durable as per the guarantees of the cloud service provider and hence is not a single point of failure. Our solution is based on the *main-delta* architecture, which we describe next. Following that are the details of our solution.

Main-Delta architecture overview: As the name suggests, the architecture has two partitions where data is stored: a *main* and a *delta*. The *main* data partition is static and read only, hence it is generally read-optimized. Whereas the *delta* partition is write-optimized, and allows for insert and read operations. Data in the main is physically modified by creating a new main. A new main is created by merging outstanding delta operations. The merging of the delta is done periodically.

2.2 Solution: Cloud-native architecture

In our solution, the *main* partition is stored on persistent cloud storage, and the *delta* partition is stored locally within the application, either in-memory or on-disk. We allow the delta partition to be replicated at the application-level whereas the main partition is replicated only once at the application-level. This is outlined in Figure 2 where triangles inside the nodes $N0$, ... Nk represent the deltas. All the nodes in the replica-set access the same *main* partition stored on cloud storage (which is internally replicated). The deltas from one or more replica nodes are merged with the main periodically. This design exploits the fault-tolerance properties of cloud storage services and avoids unnecessary data redundancy, as such we call this architecture a *cloud-native architecture* for replicated data services.

Modelling the replicated services as main-delta systems is fortunately straightforward. While not explicitly designed this way, many existing replicated data services already have this model internally, for example Kafka and Cassandra. Generally speaking we treat their in-memory data buffers as deltas and on-disk files as main. The challenge is in how the deltas should be merged with the main. This is highly dependent on the replication semantics and the failure guarantees of a particular system; we discuss this in Section 3.

3 Cloud-native architecture

There are several potential solutions to the problem of redundant replication. We generally cannot control storage-level replication in the cloud, as these services are controlled by the cloud provider. However, we can change how we do application-level replication. One simple approach is to remove application-level replication. Since the storage service already provides fault-tolerance via replication, there is no need for the application to replicate for fault-tolerance. The drawback of this approach is that it results in the loss of *availability* of the service. If the single running instance is unreachable, e.g., due to process crash, machine failure, or network partition, then the entire service is unavailable.

An alternative solution is to have multiple copies of the service share a single primary copy of the data on the cloud storage service. This way availability is maintained by having multiple instances of the services running without actually using application-level replication. There are two drawbacks to this approach. The first is that all writes to the system need to be persisted to the cloud storage service to ensure no data is lost if a service fails. For systems like Kafka and Cassandra that buffer writes in memory and flush them to storage in batches this introduces significant latency. The second drawback is for services that are engineered as shared-nothing architectures and that have multi-writer designs (e.g. Cassandra). These services would require concurrent writes to shared data, which would require re-engineering the storage of the system to coordinate concurrent updates to the shared storage, and introduces contention in the system.

We observe that main-delta systems have a desirable property that can be exploited in cloud deployments: they have a large read-only *main* data segment, which is periodically rebuilt to incorporate a batch of *deltas* in a process called the *delta merge*. Because the data in main is read-only, it would be possible for multiple instances of a replicated service to share a single main, without introducing the same contention described above with multiple instances sharing a single primary copy of the data. Only the occasional delta-merge process would need to be coordinated. Deltas, on the other hand, are kept relatively small by the recurring delta merges and are expunged after the merge takes place. Each replica can maintain its delta using the application-level replication. Delta could be kept on a local disk, in a private area of the cloud storage service, or in-memory depending on the environment and durability guarantees of the system.

Deciding which node within a replica-set merges the delta to main depends on the replication policy used by the application. Therefore, before providing the details about the delta-merge strategy, it is important to understand the replication strategies used in practice.

Application-level replication strategies: We classify these into three categories based on the strategies seen in practice.

(1) **Single-writer/single-reader:** this has a single master node in a replica-set, and all the read and write operations are handled by the master node. The role of replicas in this strategy is only to provide fault-tolerance and high-availability. For example, Kafka [18].

(2) **Single-writer/multi-reader:** the writes are handled by a single master but the reads can be handled by any replica node. The role of replicas here is to provide fault-tolerance, high-availability, and read scale-out. Examples are MongoDB [9], Redis [22], PostgreSQL, MemSQL [21], Aurora [23], and Pnuts [10].

(3) **Multi-writer/multi-reader:** reads and writes can be serviced by multiple nodes in a replica-set. The role of replicas here is to provide fault-tolerance, high-availability, and read and write scale-out. In some multi-writer/multi-reader systems, quorums of nodes are used to accept writes, which means that not all replicas in a replica-set are exact replicas of each other. Examples are Cassandra [20], CouchDB [12], PostgresXC, Dynamo [13], Spanner [11].

Depending on the replication strategy, the delta-merge strategy could be as simple as master node always merging the delta, as in the case of Kafka and for other single-writer/single-reader systems (in Section 3.1), or a more complex one, involving deltas of all replica nodes, as we will see in the case study of Cassandra or any other multi-writer/multi-reader system (in Section 3.2).

In addition to the delta-merge strategy, different replication strategies also determine which type of cloud storage can be used when using main-delta architecture for replicated services. Cloud storage can be classified into the following three categories:

(1) **Network attached block devices**: This storage is similar

to an on-premise disk; the storage is bound or attached to a single compute instance. That means only one instance at a time can mount the storage for reading and writing. Examples are Elastic Block Store (EBS) [1], Google Cloud's Persistent disk [14], and Disk Storage [6] by Azure.
(2) **NFS shared storage**: This is similar to a Network Files System (NFS) shared across multiple compute instances. Any number of compute instances can mount the storage, hence allowing multiple instances to simultaneously read and write the data. Examples are Elastic File System (EFS) [2] , Google Filestore [15], and Azure Files [7].
(3) **Object Stores**: This type of storage allows reading and writing named objects. This storage does not allow for in-place updates, data can be deleted and inserted again with new values. Examples are Amazon S3 [3], Google Cloud Storage Buckets [14], and Azure Blob Storage [5].

For *single-writer/single-reader* replication the delta can be merged only by the master node and the reads are also served by the master node. Therefore, any cloud storage which allows for one or more compute nodes to read and write data is suitable. That is, all of the above types of storage can be used. For the *single-writer/multi-reader* replication the delta is merged only by master but the reads are served by all the replica nodes, therefore each node should have read access to the main. Hence, only *NFS shared storage* and *Object Stores* can be used to store the main. Similarly for *multi-writer/multi-reader* replication, the delta from all the nodes needs to be merged, and each node serves the reads. Therefore, all nodes need read and write access to the storage. Hence, only *NFS shared storage* and *Object Stores* can be used to store the main.

In the remainder of the section we show that the delta-merge architecture is general enough to be adopted for different replication strategies. The three replication strategies discussed above captures the spectrum of possible replication strategies that exist in replicated systems. For the proof of concept we implemented the main-delta architecture in two of the well known systems: Kafka [18] which follows the *single-writer/single-reader* replication strategy, and Cassandra [20] which follows the *multi-writer/multi-reader* with quorum writes replication strategy.

3.1 Case Study: Cloud-native Kafka

Main-delta in Kafka: Kafka internally implements an append only data model. Updates are treated as new values and appended to the existing data. Compaction runs in the background and deletes older versions of the same data. To support the append only architecture Kafka has in-memory buffers to which new values are appended, and these buffers are flushed to the persistent storage regularly where it is merged with the rest of the data. The append only architecture lends itself naturally to the main-delta architecture, where the in-memory buffers are equivalent to the deltas and the data stored on persistent storage is equivalent to the main.

Delta-merge: For every data partition, there is a fixed set of Kafka brokers, called replica-sets, owning the replicas of the partition. For every replica-set only a single copy of the main is stored on persistent storage, but every broker in the replica-set maintains its own delta. The multiple copies of the delta are kept synchronized by Kafka's synchronous replication protocol. The delta-merge strategy naturally follows from the fact that there is a single master broker per replica-set (i.e. *single-writer/single-reader* replication strategy). We employ the master replica to perform the delta-merge (flush the log tail) and read the main (persisted log) to/from persistent storage, and block all the other brokers from accessing the storage. This ensures that only the master broker's delta is merged to the main. To decide when the in-memory buffer should be flushed to main we explicitly manage the in-memory buffers. In our implementation we maintain fixed size Byte Buffers in memory as deltas. Once a buffer is full it is flushed by a background thread to the file located on a persistent storage. We maintain two buffers such that when one buffer is being flushed, the other buffer is available for writes.

Failure guarantees: In the case of master failure, Kafka has an algorithm to elect the new master broker. Write permissions to the persistent storage and also the permission to merge delta are enabled only on the master. The new master broker can also read the existing log from storage to answer read requests. The guarantees provided by Kafka at the time of master node failure depends on the replication policy configured for Kafka. Replication in Kafka can be configured to either synchronous or asynchronous replication. Our modifications towards main-delta architecture allow for same replication policies to hold, because the deltas still get updates according to the specified replication policy. Once the delta is merged it is persisted on fault-tolerant storage and all the guarantees of the storage service apply.

3.2 Case Study: Cloud-native Cassandra

Main-delta in Cassandra: Cassandra, in contrast to Kafka, is a peer-to-peer system with no notion of master or slaves. Cassandra supports quorum reads and quorum writes, where n (i.e. quorum count) out of the k (number of replicas) nodes must respond to the read or write request. Each Cassandra node writes data to an in-memory data structure called *memtable*, which are regularly flushed to the disk, and merged with on-disk structure called *sstable*. Similar to Kafka, Cassandra does not support in-place updates.

The *memtable* and *sstable* structures of Cassandra [20] naturally lends itself to the main-delta architecture. In-memory memtables are logically equal to the deltas, and on-disk multiple sstables together form the main. However, what is required is to ensure that only a single copy of the main exists for every partition. Ensuring that is not straightforward in systems like Cassandra, where the replication policy is driven

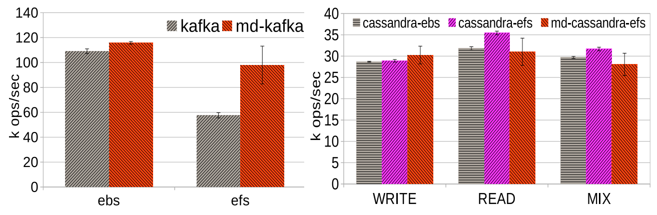

Figure 3: Left plot compares the producer throughput of *kafka* and *md-kafka*. Right plot compares throughput of *cassandra* and *md-cassandra*. Throughput is in thousand ops/sec.

by quorum writes. To implement the main-delta architecture within Cassandra we first need to decouple the storage from its processing engine. We move from per-node-storage to per-replica-set-storage. In per-node-storage, each node has its own set of memtables and dedicated storage for its sstables. In per-replica-set-storage, each node still has its own set of memtables but the sstables (which together form the main) have only a single copy per replica-set. These sstables, are stored on a shared storage (NFS or object store), and each node in the replica-set has read and write permissions to the sstables.

Delta-merge: Due to quorum writes it is uncertain that any single node in the replica-set has all the recent writes. Therefore, unlike Kafka, the delta merge needs to reconcile deltas (i.e. memtables in Cassandra) of all the nodes in the replica-set. To achieve this, we allow each node to independently flush its delta to the cloud storage whenever their delta is full, and a background compaction task merges the multiple copies of flushed deltas into a single combined delta and then append it to the main (which contains remaining sstables). Since each node can write its delta to the storage, this approach needs the main to be stored on shared storage that allows for multiple writes (i.e., NFS storage or object storage). The quorum reads are handled the same as in the original Cassandra, i.e., each node in the replica-set searches their deltas and the single shared main, and as soon as the quorum is satisfied the result is returned to the user.

Failure guarantees: Our modifications in Cassandra are mainly focused on the compaction job, which is a background process that is already part of Cassandra. This keeps rest of the architecture including failure handling unchanged. Therefore, the failure guarantees of the cloud-native Cassandra are same as original Cassandra.

4 Experimental Evaluation

We provide a preliminary evaluation of our cloud-native architecture implemented for Kafka [18], and Cassandra [20]. Our main goal is to show that we can use our architecture in practice and avoid redundant storage while maintaining same performance or improving it for certain storage types. We used Amazon Web Services (AWS) for the experiments. Kafka and Cassandra clusters had application-level replication of 3 and read/write quorum for Cassandra was 2. The clusters were hosted on EC2 instances, and we ran experiments on two types of storage: EBS [1], and EFS [2]. To generate workloads we used Kafka's and Cassandra's respective performance tool available with their source code [8, 16]. In the experiments, implementations with no modifications are labelled as *kafka* and *cassandra*, and implementations based on main-delta are labelled as *md-kafka*, and *md-cassandra*.

Figure 3 (left) shows the comparison of the throughput (thousand write operations per second) of two Kafka versions. The throughput of *md-kafka* is similar to the original Kafka in the case of EBS storage. In the case of EFS storage we achieve much higher throughput using our design (close to 2x) because the delta architecture inherently batches the write operations to the storage. We also observed 3x storage cost and space savings with both EBS and EFS storage.

In Figure 3 (right) we show the throughput comparison for Cassandra. As mentioned in Sec 3.2, the modified Cassandra (*md-cassandra*) requires storage type that allow for *multi-writer/multi-reader* systems, therefore we only use EFS storage for *md-cassandra*, labelled as *md-cassandra-efs*. However, original Cassandra can still use EBS storage where each Cassandra node has a dedicated EBS volume. The throughput of *md-cassandra-efs* is comparable to original Cassandra using EBS storage (*cassandra-ebs*) and original Cassandra using EFS storage (*cassandra-efs*), across three types of workloads: read-only, write-only, and mixed workload. The read throughput, however, is slightly worse likely due to contention on the single *main* file. Storage cost and space savings were slightly less than 3x (2.8x, averaged over 10 runs) because until the compaction job merges the deltas there exist 3 application-level replicas of them on the cloud storage. Tuning the compaction job to run frequently can bring the savings closer to 3x but with a higher load on the server. We used the default setting of running compaction for every 4 recently flushed sstables.

5 Conclusion and Discussion

In this paper we showed that existing replicated data services designed for on-premise infrastructure, when deployed on the cloud, end up with redundant replicas of the storage without providing significant additional fault-tolerance. We present a main-delta based cloud-native architecture for replicated data services which solves the problem of redundant replicas, allowing application owners to pay for *k* (application's replication factor) times less for storage while maintaining same performance, fault-tolerance, and availability. As a proof-of-concept, we implemented our solution in two popular replicated services, Apache Kafka and Apache Cassandra, and demonstrated that our approach is general, and could be applied to various types of replicated data services.

5.1 Discussion

While engineering cloud-native systems from the ground up is an active and important line of research, the question of how to migrate existing on-premise architectures to the cloud is both a pragmatic and urgent problem for many people. We raise the following points for discussion. First, *what are the different architectures in the design space of replicated services on the cloud?* We start with the default on-premise setting, where these services benefit from low latency storage I/O but need to design robust data replication strategies to handle failures. Next, this on-premise architecture can be moved to the cloud as-is where local storage is replaced by cloud storage. Although this is an easy and natural thing to do for many users, this design suffers from high over the network storage I/O and has the problem of redundant data replication as pointed out in this work. To avoid this we proposed the incorporation of main-delta architecture for managing storage. To keep the failure guarantees unchanged, this approach requires careful strategies for merging the deltas from the replicas to a single shared main (see Sections 3.1 and 3.2). Other architectures in the design space are built from scratch and have re-designed the replication strategy to suit cloud storage characteristics, such as fault-tolerance and high I/O latency, and lowered the critical path I/O latency by just keeping the log on the cloud storage (as in Amazon Aurora [23], and Microsft Socrates [4]). Since they are designed from scratch they can extract maximum performance from the cloud infrastructure. Whereas, our main-delta approach tries to port existing services to the cloud with minimal changes in their architecture. Due to this, there could be performance differences between our approach vs. systems designed from scratch and comprehensive empirical evaluation will be a promising future work.

Second, *how can replicated services benefit from the cloud storage, which is fault-tolerant and highly available?* In this work, we see these characteristics of the cloud storage as an opportunity to save on the storage cost for the replicated services, when deployed on the cloud. In a related work, SHADOW [17], the shared fault-tolerant storage has been exploited for high availability. SHADOW system stored the log on this shared fault-tolerant storage and allowed the active server to write transactions to the log and the hot standby server to read the log and make updates to the database.

Third, *what are the practical limitations of our approach?* We believe that the general methodology of retaining application-level protocols to sync in-memory state, combined with decoupled fault-tolerant storage provides a foundation that can be used to migrate on-premise architectures to cloud-native storage. The core challenge is in coordinating the delta-merge phase, in which the main memory state is flushed to disk. We have provided insight into this problem by exploring different classes of replicated systems, and by prototyping the approach in the two very different architectures found in Apache Kafka and Apache Cassandra.

References

[1] Amazon elastic block store. https://aws.amazon.com/ebs/.

[2] Amazon elastic file system. https://aws.amazon.com/efs/.

[3] Amazon s3. https://aws.amazon.com/s3/.

[4] 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. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1743–1756, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Azure blob storage. https://azure.microsoft.com/en-ca/services/storage/blobs/.

[6] Azure disk storage. https://azure.microsoft.com/en-ca/services/storage/disks/.

[7] Azure files. https://azure.microsoft.com/en-ca/services/storage/files/.

[8] The cassandra-stress tool. https://docs.datastax.com/en/archived/cassandra/2.1/cassandra/tools/toolsCStress_t.html.

[9] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[10] 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. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. 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. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[12] CouchDB. http://couchdb.apache.org/.

[13] 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. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[14] Google cloud persistent disk. https://cloud.google.com/compute/docs/disks.

[15] Google cloud filestore. https://cloud.google.com/filestore/.

[16] Kafka performance tool. https://github.com/apache/kafka/tree/trunk/core/src/main/scala/kafka/tools.

[17] J. Kim, K. Salem, K. Daudjee, A. Aboulnaga, and X. Pan. Database high availability using shadow systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 209–221, New York, NY, USA, 2015. ACM.

[18] J. Kreps. Kafka : a distributed messaging system for log processing. 2011.

[19] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proc. VLDB Endow.*, 5(1):61–72, Sept. 2011.

[20] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[21] Memsql. https://docs.memsql.com/operational-manual/v6.7/using-replication/.

[22] Redis. https://redis.io/.

[23] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, New York, NY, USA, 2017. ACM.