Banyan: Coordination-free Distributed Transactions over Mergeable Types

Shashank Shekhar Dubey[1], KC Sivaramakrishnan[1], Thomas Gazagnaire[2], and Anil Madhavapeddy[3]

[1] Indian Institute of Technology, Madras, India
[2] Tarides, France
[3] University of Cambridge Computer Laboratory, UK

Abstract. Programming loosely connected distributed applications is a challenging endeavour. Loosely connected distributed applications such as geo-distributed stores and intermittently reachable IoT devices cannot afford to coordinate among all of the replicas in order to ensure data consistency due to prohibitive latency costs and the impossibility of coordination if availability is to be ensured. Thus, the state of the replicas evolves independently, making it difficult to develop correct applications. Existing solutions to this problem limit the data types that can be used in these applications, which neither offer the ability to compose them to construct more complex data types nor offer transactions.

In this paper, we describe Banyan, a distributed programming model for developing loosely connected distributed applications. Data types in Banyan are equipped with a three-way merge function à la Git to handle conflicts. Banyan provides isolated transactions for grouping together individual operations which do not require coordination among different replicas. We instantiate Banyan over Cassandra, an off-the-shelf industrial-strength distributed store. Several benchmarks, including a distributed build-cache, illustrates the effectiveness of the approach.

1 Introduction

When applications replicate data across different sites, they need to make a fundamental choice regarding the consistency of data. Strong consistency properties such as Linearizability [20] and Serializability [9] makes it easier to design correct applications. However, strong consistency is often at odds with high performance. Strong consistency necessitates that all the replicas coordinate to agree on a global order in which any conflicting operations are resolved. The CAP theorem [17] and PACELC theorem [1] state that strongly consistent applications exhibit higher latencies when all the replicas are reachable, and they are unavailable when some of the replicas are unreachable. This limitation has spurred the development of commercial weakly consistent distributed databases for wide-area applications such as DynamoDB [2], Cassandra [3], CosmosDB [4] and Riak [32]. However, developing correct applications under weak consistency is challenging due to the fact that the operations may be reordered in complex ways even if issued by the same session [11]. Moreover, these databases only offer a limited set of sequential data types with a built-in conflict resolution strategies such as last-write-wins and multi-valued objects. Such built-in conflict resolution leads to anomalies such as write-skew [8] which makes it difficult (and often impossible) to develop complex applications with rich behaviours.

Rather than programming with sequential data types while reasoning about their semantics in a weakly consistent setting, an alternative strategy is to equip the data types with the ability to reconcile conflicts. Kaki et al. [23] recently proposed Mergeable Replicated Data Types (MRDTs) as a way to automatically derive correct distributed variants of ordinary data types. The inductively defined data types are equipped with an invertible relational specification which is used to derive a three-way merge function à la Git [18], a distributed version control system.

What does it take to make MRDTs a practical alternative to implementing high-throughput, low-latency distributed applications such as the ones that would be implemented over industrial-strength distributed databases? There are several key challenges to getting there.

Firstly, while MRDTs define merge semantics for operations on individual objects, Kaki et al. do not describe the semantics of composition of operations on multiple objects – i.e. transactions. Transactions are indispensable for building complex applications. Strongly consistent distributed transactions suffer from unavailability [1], whereas highly-available transactions [5] combined with weakly consistent operations often lead to incomprehensible behaviours [36].

Secondly, MRDTs impose significant burden on the storage and network layer to be able to support three-way merges to reconcile conflicts. Kaki et al. implement MRDTs over Irmin [21], a Git-like store for arbitrary objects, not just files. As with Git, in order to reconcile conflicts, three-way merges in MRDTs require the storage layer to record enough history to be able to retrieve the *lowest common ancestor* (LCA) state. For a distributed database, performance of the network layer is quite important for throughput and latency. Industrial-strength distributed databases use gossip protocols [24] to quickly disseminate updates in order to ensure fast convergence between the replicas. Git comes equipped with a remote protocol for transferring objects between remote sites using *push* and *pull* mechanisms. Unfortunately, directly using the Git remote protocols would mean that the client will have to name branches explicitly, complicating the programming model. The onus is on the client to ensure that all the branches that have updates are merged in order to ensure that there is convergence. This is undesirable.

Contributions. In this paper, we present Banyan, a programming model for building loosely connected distributed applications that provides coordination-free transactions over MRDTs. Banyan provides per-object causal consistency, and the transaction model is built on the principles of Git-like branches. Rather than relying on Git remote protocol for dissemination across replicas, we instantiate Banyan on top of Cassandra, an industrial-strength, off-the-shelf distributed store [26]. Unlike Git, Banyan does not expose named branches explicitly, and ensures eventual convergence. Importantly, Banyan only relies on eventual consistency, and Banyan can be instantiated on any eventually consistent key-value store. Extensive evaluation shows that Banyan makes it easy to build complex high-performance distributed applications.

The rest of the paper is organised as follows. We motivate the Banyan model by designing a distributed build cache in the next section. Section 3 describes the Banyan programming model. Section 4 describes the instantiation of Banyan on Cassandra. We evaluate the instantiation of Banyan on top of Cassandra in section 5. Sections 6 and 8 present the related work and conclusions, respectively.

2 Motivation: A Distributed Build Cache

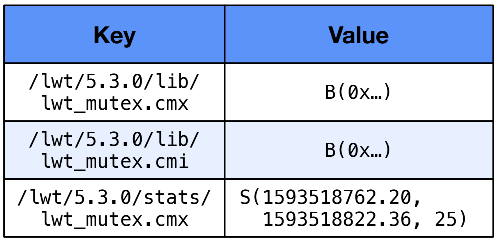
A distributed build cache enables a team of developers and/or a continuous integration (CI) system to reuse the build artefacts between several builds. Such a facility is provided by modern build tools such as Gradle [19] and Bazel [7], which can store and retrieve build artefacts from cloud storage services such as Amazon S3 or Google Cloud Storage. Consider the challenge of building a distributed build cache for OCaml packages. Let us assume that the builds are reproducible – that is, independent builds of the same source files yield the same artefact. In addition to storing the artefacts, it would be useful to gather statistics about the artefacts such as creation time, last accessed time and number of cache hits. Such information may be used in the cache eviction policy or replicating artefacts across several sites for increased availability. While an artefact itself is reproducible, care must be taken to ensure that the statistics are consistent. For the sake of exposition, we will assume that all the build hosts use the same operating system and compiler version.

2.1 Mergeable types

Let us build this distributed cache using Banyan, implementing it in OCaml. At its heart, Banyan is a distributed key-value store. The keys in Banyan are *paths*, represented as list of strings. The values are algebraic data types equipped a merge function that reconciles conflicting updates. In this example, we will use the following schema: `[<pkg_name>; <version>; <kind>; <filename>]` for the keys, where `<kind>` is either `lib` indicating binary artefact or `stats` indicating statistics about the artefact. The value type is given below:

```
type timestamp = float
type value =
  | B of bigarray  (*binary artefact*)
  | S of timestamp (*created*) * timestamp (*last accessed*)
       * int (*hits*)
```

The value is either a binary artefact or a statistics triple. Figure 1 shows the slice of the build cache key-value store. The cache stores the artefacts (`cmx` and `cmi` files) produced as a result of compiling the source file `lwt_mutex.ml` from the package `lwt` version `5.3.0`. The build cache also stores the statistics for every artefact. The example shows that the `lwt_mutex.cmx` was accessed 25 times. When several developers and/or CI pipelines are running concurrently on different hosts, they may attempt to add the same artefact to the store, or, if the artefact is already present, retrieve it from the cache and update the corresponding artefact statistics. It would be unwise to synchronize across all of the hosts for updating the store, and suffer the latency hit and potential unavailability. Hence, Banyan only writes an update to one of the replicas. The replicas asynchronously share the updates between each other, and resolve conflicting updates using user-defined three-way merge function. The merge function for the build cache is given below.



Key	Value
/lwt/5.3.0/lib/ lwt_mutex.cmx	B(0x…)
/lwt/5.3.0/lib/ lwt_mutex.cmi	B(0x…)
/lwt/5.3.0/stats/ lwt_mutex.cmx	S(1593518762.20, 1593518822.36, 25)

Fig. 1: A slice of the build cache key-value store.

```
1 let merge (lca: value option) (v1: value) (v2: value) : value =
2   match lca, v1, v2 with
3   | None, B a1, B a2 (* no lca *)
4   | Some (B _), B a1, B a2 -> assert (a1 = a2); B a1
5   | None, S(c1,la1,h1), S(c2,la2,h2) -> (* no lca *)
6       S(min c1 c2, max la1 la2, h1 + h2)
7   | Some(S(_,_,h0)), S(c1,la1,h1), S(c2,la2,h2)->
8       S(min c1 c2, max la1 la2, h1 + h2 - h0)
9   | _ -> failwith "impossible"
```

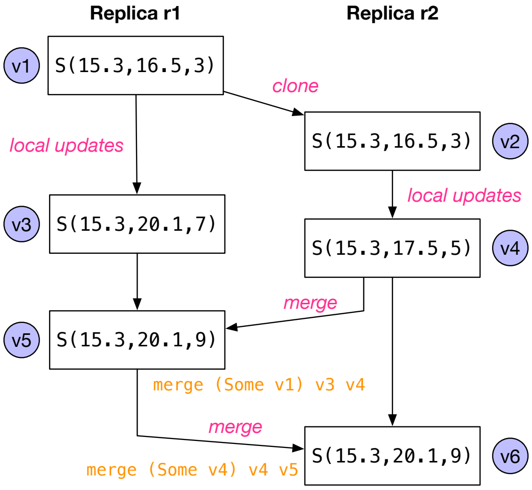


Fig. 2: Merging conflicting statistics updates.

The key idea here is that Banyan tracks the *causal history* of the state updates such that it is always known what the *lowest common ancestor* (LCA) state is, if one exists. This idea is analogous to how Git tracks history with the notion of *branches*. The merge function is applied to the LCA and the two conflicting versions to determine the new state. In the case of build cache, since the builds are reproducible, the binary artefacts will be the same (line 4). The only interesting conflicts are in the statistics. The merge function picks the earliest creation timestamp, latest last accessed timestamp, and the sum of the new cache hits since the LCA in the two branches and the original value at the LCA, if present (lines 5–8).

Figure 2 shows how the merge function helps reconcile conflicts. The arrows capture the happens-before relationship between the states. Assume that replica `r2` starts off by cloning the branch corresponding to replica `r1`. Subsequently both `r1` and `r2` performed local updates. The remote updates are reconciled by calling the merge function on each of the conflicting values. The value `v5` is obtained with merging the values `v3` and `v4` with `v1` as LCA. Importantly, observe that the cache hit count is `9` in `v5` which corresponds to the sum of `3` hits in the initial state, `4` additional hits in `r1` and `2` additional hits in `r2`. At this point, `r1` has all the changes from `r2`, but the vice-versa is not true. Subsequently, when `r1` is merged into `r2`, both the replicas have converged.

```
let compile s (* session *) =
  let ts = Unix.gettimeofday () in
  let lib = ["lwt";"5.3.0";"lib"] in
  let stats = ["lwt";"5.3.0";"stats"] in
  refresh s >>= fun () ->
  read s (lib @ ["lwt_mutex.cmx"]) >>= fun v ->
  match v with
  | None ->
      let (cmx, cmi, o) = ocamlopt "lwt_mutex.ml" in
      write s (lib @ ["lwt_mtex.cmx"]) (B cmx) >>= fun _ ->
      write s (stats @ ["lwt_mutex.cmx"]) (S (ts,ts,0))
                                              >>= fun _ ->
      ... (* similarly for cmi and o files *)
      publish s >>= fun _ ->
      return (cmx, cmi, o)
  | Some cmx ->
      read s (stats @ ["lwt_mutex.cmx"])
                                 >>= fun (Some M(c,la,h)) ->
      write s (stats @ ["lwt_mutex.cmx"]) (S (c,ts,h+1))
                                               >>= fun _ ->
      read s (lib @ ["lwt_mutex.cmi"]) >>= fun (Some cmi) ->
      read s (lib @ ["lwt_mutex.o"]) >>= fun (Some o) ->
      ... (* update stats for cmi and o file *)
      publish s >>= fun _ ->
      return (cmx, cmi, o)
```

Fig. 3: Compiling `lwt_mutex.ml`.

2.2 Transactions

Now that we the mergeable value type for the build cache, let us see how we can compile `lwt_mutex.ml` using Banyan. Figure 3 shows the code for compiling `lwt_mutex.ml`. In Banyan, the clients interact with the store in *isolated* sessions. A session can fetch recent updates using the **`refresh`** primitive and make *all* the local updates visible to other sessions using the **`publish`** primitive. During

refresh, any conflicting updates are resolved using the three-way merge function associated with the value type.

In order to compile `lwt_mutex.ml`, we first refresh the session to get any recent updates. Then, we check whether the `lwt_mutex.cmx` file is in the build cache. If not, the source file is compiled, and the resultant artefacts (`cmx`, `cmi`, `o` files) and the corresponding entries for updated statistics are written to the store. Finally, the all the local updates are published.

The all or nothing property of **refresh** and **publish** is critical for the correctness of this code. Observe that when the artefact is locally compiled, all the artefacts and their statistics are published atomically. This ensures that if a session sees the `cmx` file, then other artefacts and their statistics will also be visible. Thus, Banyan makes it easy to write highly-available, complex distributed applications in an idiomatic fashion.

3 Programming Model

In this section, we shall describe the system and programming model of Banyan from the developers point-of-view. The Banyan store consists of several replicas, which are fully or partially replicated [13]. The replicas asynchronously distribute updates amongst themselves until they converge. The key property that enables Banyan to support mergeable types and isolated transactions is that Banyan tracks the history of the store in the same way that Git tracks the history of a repository.

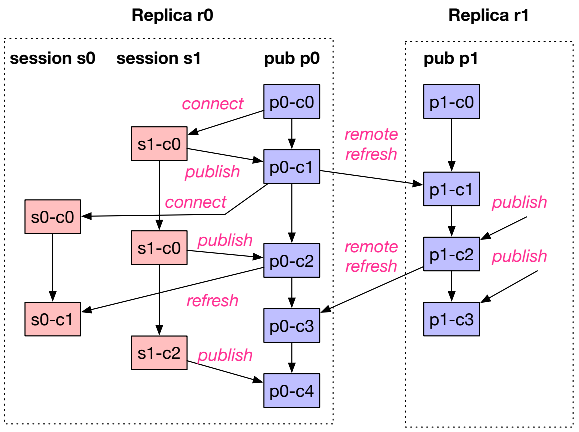


Fig. 4: Banyan system and programming model.

Figure 4 presents the schematic diagram of the system and programming model. Each replica has a distinguished public branch `pub`, which records the history of the changing state at that replica. Each node in this connected history graph represents a *commit*. Whenever a new client connection is established, a new branch is forked off the latest commit in the public branch. Any reads or writes in this session are only committed to this branch unless explicitly published. This ensures the isolation property of each session. The figure shows the creation of two sessions in the replica `r0`.

The simplified Banyan API is given below:

```
type config  (* Store configuration *)
type session
type key = string list
type value   (* Type of mergeable values in the store *)

val connect : config -> session Lwt.t
val close   : session -> unit Lwt.t
val read    : session -> key -> value option Lwt.t
val write   : session -> key -> value -> unit Lwt.t
val publish : session -> unit Lwt.t
val refresh : session -> unit Lwt.t
```

When a client connects to a Banyan store, a new session is created, which is rooted to one of the replicas in the store. Every write creates a commit in the session performing the write. As previously explained, Banyan permits the sessions to atomically `publish` their updates and `refresh` to obtain latest updates. The `publish` operation squashes all the local commits since the previous `refresh` or `publish` to a single commit, and then *pushes* the changes to the public branch on the replica to which the session is rooted. The `refresh` operation *pulls* updates from the public branch into the current sessions branch. Both `publish` and `refresh` may invoke the merge function on the value type if there are conflicts. The objects that written to each replica are asynchronously replicated to other replicas. Banyan offers causal consistency for operations on each key.

Periodically, the changes from other public branches are *pulled* into a replica's public branch (remote refresh). This operation happens *implicitly* and asynchronously, and does not block the client on that replica. When a session is closed, the outstanding writes are implicitly published. Similarly, when a session is connected, there is an implicit refresh operation.

Observe that both the local and the remote refresh operations are non-blocking – it is always safe for `refresh` to return with updates only from a subset of public branches. The only push operation is due to `publish`. When pushing to a branch, it is necessary to atomically update the target branch to avoid concurrency errors. The key observation is that only the session that belongs to a replica can push to the public branch on that replica. This can be achieved with replica-local concurrency control and does not require coordination among the replicas. Hence, Banyan transactions do not need inter-replica coordination, and hence, are available.

When a particular replica goes down, the sessions that are rooted to that replica may not have enough history to be able to `refresh` and `publish` to other replicas. In particular, `refresh` and `publish` will need to discover the LCA in the case of conflicting updates. Since the objects are asynchronously replicated across the replicas, the recent writes to the replica that went down may not have been replicated to other replicas. Hence, Banyan requires sticky availability [5] – the sessions need to reach the logical replica to which it originally connected. In practice, with partial replication, a logical replica may be represented by a set of physical servers. As long as one of these physical servers is reachable, the system remains available for that session.

Compared to traditional transactions usually executed at a particular isolation level, `refresh` and `publish` permits more fine-grained, explicit control of visibility. In Banyan, transactions are delimited by `publish` operations, begin and end of sessions. For example, the set of writes performed between consecutive `publish` operations are made visible atomically outside the session. The transaction may abort if the three-way merge function throws an exception. However, in practice, the useful MRDTs are designed in such a way that a merge is always possible, and the failure of the merge function represents a bug. This idea of merge always being possible ensures *strong eventual consistency*, espoused by convergent replicated data types [33]. Banyan adds transactional support over strong eventual consistency.

The `publish` and `refresh` can be used to achieve well-known isolation levels. For example, consider parallel snapshot isolation (PSI) [35], which is an extension of snapshot isolation (SI) [8] for geo-replicated systems. Like SI, the transactions in PSI operate on a snapshot of the state at a replica. While SI precludes write-write conflicts, PSI admits them on mergeable types. Since all the data types in Banyan are mergeable types, every write-write conflict can be resolved. We can achieve PSI by `refresh`ing at that beginning of the transaction and `publish`ing at the end of the transaction with no intervening `refresh`es.

Similarly, we get monotonic atomic view (MAV) [5] isolation level if two consecutive `publish` operations are interspersed with `refresh`es. Since the `refresh`es may bring in new updates from committed transactions, the state of the transaction grows monotonically.

4 Implementation

In this section, we describe the instantiation of Banyan on Cassandra [3], a popular, industrial-strength, column-oriented, distributed database. Cassandra offers eventual consistency with a last-write-wins conflict resolution policy. Cassandra also offers complex data types such as `list`, `set` and `map` with baked-in conflict resolution policies. Given the richness of replicated data types, the available complex data types are quite limiting. Cassandra also offers lightweight transactions (distributed compare-and-update) implemented using the Paxos consensus protocol [27]. Lightweight transactions are limited to operate on only one object. Banyan does not use lightweight transactions since their cost is prohibitively high due to consensus. As mentioned previously Banyan only requires sticky availability, and so uses a replica-local lock for ensuring mutual exclusion when multiple sessions try to update the public branch on a replica concurrently.

By instantiating Banyan on Cassandra, we offload the concerns of replication, fault tolerance, availability and convergence to the backing store. On top of Cassandra, Banyan uses Irmin [21], an OCaml library for persistent stores with built-in branching, merging and reverting facilities. Irmin can be configured to use different storage backends, and in our case, the storage is Cassandra. Importantly, Cassandra being a distributed database serves the purpose of the networking layer in addition to persistent storage. While Irmin permits arbitrary branching and merging, Banyan is a specific workflow on top of Irmin which retains high availability.

4.1 Irmin data model

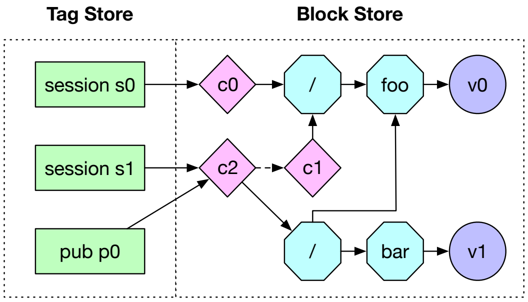


Fig. 5: A sample Irmin store. The rectangles are tags, diamonds are commit objects, octagons are tree object, and circles are blob objects.

The expressivity of Irmin imposes significant burden on the underlying storage. For efficiently storing different versions of the state as the store evolves, Irmin uses the Git object model. Figure 5 shows a snapshot of the state of the Irmin store. There are two kinds of stores: a mutable tag store and an immutable, content-addressed block store. The tag store records the branches and the commit that corresponds to this branch. In this example, we have three branches, `session s0`, `session s1` and `pub p0`.

The block store is content-addressed and has three different kinds of objects: commits, tree and blobs. A commit object represents a commit, and it may have several parent commits and a single reference to a tree node. For example, the commit `c2`'s parent is `c1`, and `c0` and `c1` do not have any parent commits. The tree object corresponds to directory entries in a filesystem, and recursively refer to other tree objects or a blob object. Unlike Git, Irmin allows blob objects to be arbitrary values, not just files. The blob objects may refer to other blob objects. In the session `s1`, reading the keys `["foo"]` and `["bar"]` would yield `Some v0` and `Some v1`, respectively.

Observe that all the commits share the tree object `foo` and its descendents, thanks to the block store being content addressed. Content addressibility of the block store means that as the store evolves, the contents of the store are shared between multiple commits, if possible. On the other hand, updating a value in a deep hierarchy of tree objects would necessitate allocating a new spine in order to maintain both the old and the new versions. Thus, each write in Banyan will turn into several writes to the underlying storage.

4.2 Cassandra instantiation

For instantiating Banyan on Cassandra, we use two tables, one for the tag store and another for the block store. For the tag store, the key is a `string` (tag) and the value is a `blob` (hash of the commit node). For the block store, the key is a `blob` (hash of the content), and the value is a `blob` (content). Irmin handles the logic necessary to serialize and deserialize the various Git objects into binary `blob`s and back.

Cassandra replicates the writes to the tag and block tables asynchronously amongst the replicas. Each replica periodically merges the public branches of other replicas into its public branch to fetch remote updates. Due to eventual consistency of Cassandra, it may be the case that not all the objects from a remote replica are available locally. For example, the merge function may find a new commit from a remote replica, but the tree object referenced by a commit object may not available locally. In this situation, Banyan simply skips merging this branch in this round. Cassandra ensures that eventually the remote tree object will arrive at this replica and will be merged in a subsequent remote refresh operation. Thus, fetching remote updates is a non-blocking operation.

In Irmin, the tag store is updated with a compare-and-swap to ensure that concurrent updates to the same tag should be disallowed. Naively implementing this in Cassandra would necessitate the use of lightweight transactions and suffer prohibitive costs. By restricting the Banyan programming model (Section 3) such that entries in the tag store (in particular, the tag corresponding to the public branch of the replica) is only updated on that replica, we remove the necessity for lightweight transactions. Thus, we do not depend on any special features of Cassandra to realise the Banyan model, and Banyan can be instantiated on any eventually consistent key-value store.

4.3 Recursive merges

A particular challenge in making Banyan scalable is the problem of recursive merges. Consider a simple mergeable counter MRDT, whose implementation is:

```
let merge lca v1 v2 =
  let old = match lca with None -> 0 | Some v -> v in
  v1 + v2 - old
```

Consider the execution history presented in Figure 6 which shows the evolution of a single counter. The history only shows the interaction between two replicas, and does not show any sessions. Each node in the history is a commit. Since we want to focus on a single counter, for simplicity, we ignore the tree nodes and the node labels show the counter value.

Initially the counters are `0`, and each replica concurrently increments the counter by `4` and `5`. When the replicas perform remote refreshes, they invoke `merge None 4 5` to resolve the conflict updates yielding `9`. The LCA is `None` since there is no common ancestor.

Subsequently, the replicas increment the counters by `3` and `5`. Now, consider that the replicas merge each other's branches. When merging `12` and `14`, there are two equally valid LCAs `4` and `5`. Picking either one of them leads to incorrect result. At this point, Irmin merges the two LCAs using `merge None 4 5` to yield `9`, which is used as the LCA for merging `12` and `14`. This yields the value `17`. The result of merging the LCAs is represented as a rounded rectangle. Importantly, the result of the recursive merge `9` is not a parent commit of `12` and `14` (distinguished by the use of dotted arrows). This is because the commit nodes are stored in the content-addressed store, and adding a new parent to the commit node would create a distinct node, whose hash is different from the original node. Any other nodes that referenced the original commit node will continue to reference the old node. As a result, the recursive merges will need to be performed again for subsequent requests!

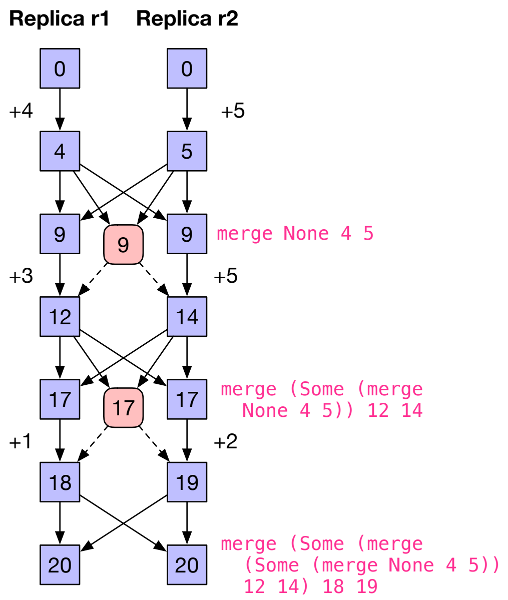

Replica r1
Replica r2
0
0
+4
+5
4
5
9
9
9
merge None 4 5
+3
+5
12
14
17
17
17
merge (Some (merge None 4 5)) 12 14
+1
+2
18
19
20
20
merge (Some (merge (Some (merge None 4 5)) 12 14) 18 19

Fig. 6: Recursive merge. Rounded rectangles are the results of recursive merges.

Consider that the replicas further evolve by incrementing `1` and `2`, yielding `18` and `19`. When these commits are merged on remote refresh, there are two LCAs `12` and `14`, which need to be merged. This in turn has two LCAs `4` and `5`, which need to be merged. Thus, every subsequent recursive merge, which is very likely since the replicas merge each other's branches, requires repeating all the previous recursive merges. This does not scale.

We solve this problem by having a separate table in Cassandra that acts as a cache, recording the result of LCA merges. Whenever Banyan encounters a recursive merge, the cache is first consulted before performing the merge. In this example, when `18` and `19` are being merged, Banyan first checks whether the two LCAs `12` and `14` are in the cache. They would not be. This triggers a recursive merge of LCAs `4` and `5`, whose result is in the cache, and is reused. The cache is also updated with an entry that records that the merge of the LCAs `12` and `14` is the commit corresponding to `17`.

4.4 Garbage collection

While traditional database systems only store the most recent version of the data, Banyan necessitates that previous versions of the data must also be kept around for three-way merges. While persistence of prior versions [15,16] is a useful property for audit and tamper evidence, the Banyan API presented here does not provide a way to access earlier versions. The question then is: when can those prior versions be garbage collected?

We have not yet implemented the garbage collector for Banyan on Cassandra, but we sketch the design here. Git is equipped with a garbage collector that considers that any object in the block store that is reachable from the tag store is alive. Unreachable objects are deleted. Our aim is to assist the Git-like garbage collector by pruning the history graph of nodes which will no longer be used. The key idea is that if a commit node will not be used for LCA computation, then that commit node may be deleted. Deleting commit nodes will leave dangling references from its referees, but Irmin can be extended to ignore dangling references to commit nodes.

For individual sessions, once the session is closed, the corresponding entry in the tag store, and all the commits by that session may be deleted. In the execution history in Figure 7, the commit node `s0-c0` may be deleted. The next question is when can commits on public branch be deleted. For each ongoing session in a replica, we maintain the latest commit in the public branch against which **`refresh`** was performed. The earliest of such commits in the public branch and its descendants must be retained, since they are necessary for the three-way merge. For example, in Figure 7, session `s1` refreshed against `p0-c2`, and `s1` is the only ongoing session. If `s1` publishes, then `p0-c2` will be the LCA commit.

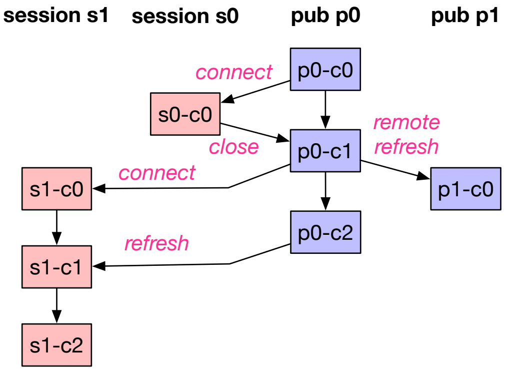


Fig. 7: Garbage collection. Here, the commits `p0-c0` and `s0-c0` may be deleted.

A similar reasoning is used for remote refreshes. When a commit in the public branch of a replica has been merged into the public branches of all the other replicas, then the ancestors of such commits will not be accessed and can be deleted. In Figure 7, assume that we only have two replicas. Since `p0-c1` was merged by the public branch `p1`, `p0-c1` will be the LCA commit for subsequent remote refreshes by `p1`. Given that `p0-c0` is neither necessary for remote refreshes nor for ongoing sessions, `p0-c0` can be deleted.

5 Evaluation

In this section, we evaluate the performance of Banyan's instantiation on Cassandra. Our goal is to assess the suitability of Banyan for programming loosely connected distributed applications. To this end, we first quantify the overheads of implementing Banyan over Cassandra. Subsequently, we assess the performance of MRDTs implemented using Banyan. And finally, we study the performance of distributed build cache (Section 2).

5.1 Experimental setup

For the experiments, we use a Cassandra cluster with 4 nodes within the same data center. Each Cassandra node runs on a baremetal Intel®Xeon®E3-1240 CPU, with 4 physical cores, and 2 hardware threads per core. Each core runs at

3.70GHz and has 128KB of L1 data cache, 128KB of L1 instruction cache, 1MB L2 cache and 8MB of L3 cache. Each machine has 32GB of main memory. The machines are unloaded except for the Cassandra node. The ping latency between the machines is 0.5ms on average. The clients are run on a machine with the same configuration in the same data center.

For the experiments, Cassandra cluster is configured with a replication factor of 1, read and write consistency levels of `ONE`. Hence, the cluster maintains a single copy of each data item, and only waits for one of the servers to respond to return the result of read and write to the client. These choices lead to eventual consistency where the reads may not return the latest write. The cluster may be configured with larger replication factor for better fault tolerance. However, stronger consistency levels are not useful since Banyan enforces per-key causal consistency over the underlying eventual consistency offered by Cassandra. In fact, choosing strong consistency for reads and writes in Cassandra does not offer strong consistency in Banyan since the visibility of updates in Banyan is explicitly controlled with the use of **`refresh`** and **`publish`**.

5.2 Baseline overheads

Given that Banyan has to persist every version of the store, what is the impact of Banyan when compared to using Cassandra in a scenario where Cassandra would be sufficient? We measure the throughput of performing 32k operations, with 80% reads and 20% writes with different numbers of clients. The keys and values are 8 and 128 byte strings, respectively. For Banyan, we use last-writer-wins resolution policy, which is the policy used by Cassandra. The results are presented in Figure 8.

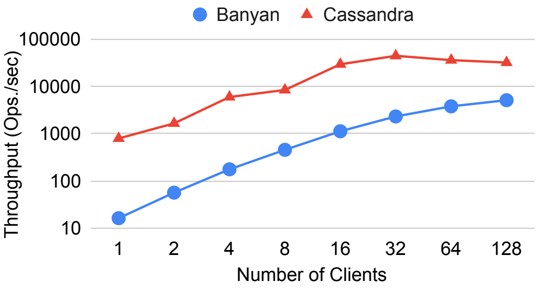


Fig. 8: Performance comparison between Banyan and Cassandra on LWW string value.

With 1 client, Banyan performs 16 operations per second, while Cassandra performs 795 operations per second. Cassandra offers 50× more throughput than Banyan with 1 client. This is due to the fact that every read (write) performs 4 reads (3 reads and 4 writes) to the underlying store to create and access the tag, commit and tree nodes. Banyan additionally includes marshalling and hashing overheads for accessing the content-addressed block store. Cassandra does not include any of these overheads. Luckily, Banyan overheads are local to a client, and hence, can be easily parallelized. With 1 client, the cluster is severely under utilized, and the client overheads dominate. With increasing number of clients, the cluster is better utilized. At 128 clients, Cassandra performs 31274 operations per second where as Banyan performs 5131 operations per second, which is a slowdown of 6.2×. We believe that these are reasonable overheads given the stronger consistency and isolation guarantees, and better programming model offered by Banyan.

At the end of 32k operations, Cassandra uses 4.9MB of disk space, while Banyan uses 1.8GB of disk space. As mentioned earlier (§4.4), we have yet to implement garbage collection for Banyan– once implemented, we expect this space usage will come down significantly.

5.3 Mergeable Types

Counter We begin with the counter data type discussed in Section 4.3. How does a Banyan counter perform on when concurrently updated by multiple clients? For the experiment, the value type is a counter that supports increment, decrement and read operations. The clients perform 32k increment or decrement operations on a key randomly selected from a small key space. Each client refreshes and publishes after every 100 operations. By choosing a small key space, we aim to study the scalability of the system with large number of conflicts.

Figure 9 shows the performance result for two key spaces of size 1024 and 4096 keys. With 1 client, there are no conflicts. The conflicts increases with increasing number of clients. We get a peak throughput of 1814 (2027) operations per second with a key space of 1024 (4096) keys. Observe that the number of conflicts is considerably lower with 4096 keys when compared to 1024 keys. As a result, the throughput is higher with 4096 keys. The result shows that the throughput of the system is proportional to the number of conflicting operations.

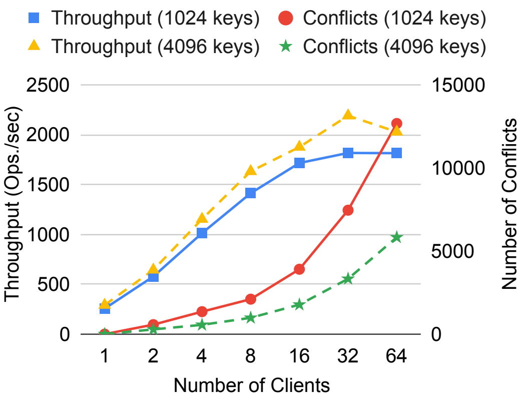


Fig. 9: Performance of counter MRDT.

Blob log Another useful class of MRDTs are *mergeable logs*, where each log message is a string. Such a distributed log is useful for collecting logs in a distributed system, and examining the logs in their global time order. To this end, each log entry is a pair of timestamp and message, and the log itself is a list of such entries in reverse chronological order. The merge function for the mergeable log extracts the newer log entries from both the versions, sorts the newer entries in reverse chronological order and returns the list obtained by appending the sorted newer entries to the front of the log at the LCA.

While this implementation is simple, it does not scale well. In particular, each commit stores the entire log as a single serialized blob. This does not take advantage of the fact that every commit can share the tail of the log with its predecessor. Moreover, every append to the log needs to deserialize the entire log, append the new entry and serialize the log again. Hence, append is an $O(n)$ operation, where n is the size of the log. Merges are also worst case $O(n)$. This is undesirable. We call this implementation a *blob log*.

Linked log We can implement an efficient log by taking advantage of the fact that every commit shares the tail of the log with its predecessor. The value type in this log is:

```
type value =
  | L of float (* timestamp *) * string (* message *)
       * blob (* hash of prev value *)
  | M of blob list (* hashes of the values being merged *)
```

The value is either a log entry `L(t,m,h)` with timestamp `t`, message `m` and a hash of the previous value `h`, or `M hs` where `hs` is the list of hashes of the values being merged. Appending to the log only needs to add a new object that refers to the previous log value. Hence, append is $O(1)$. Figure 10 shows a snapshot of the log assuming a single key `x`. The log at `x` in the public branch `p0` (session `s0`) is `[a;b;c]` (`[a;b;d]`). The merge operation simply adds a new value `M [h1;h2]`, which refers to the hashes of the two log values being merged. This operation is also $O(1)$. The read function for the log does the heavy-lifting of reading the log in reverse chronological order.

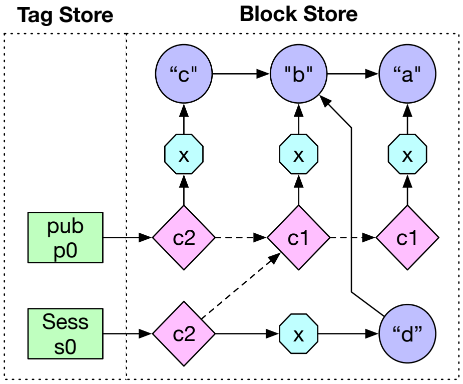


Fig. 10: A snapshot of linked log storage.

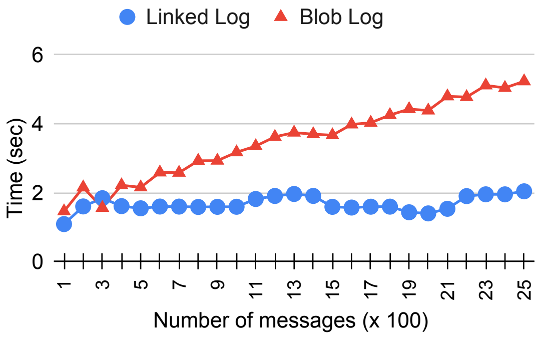


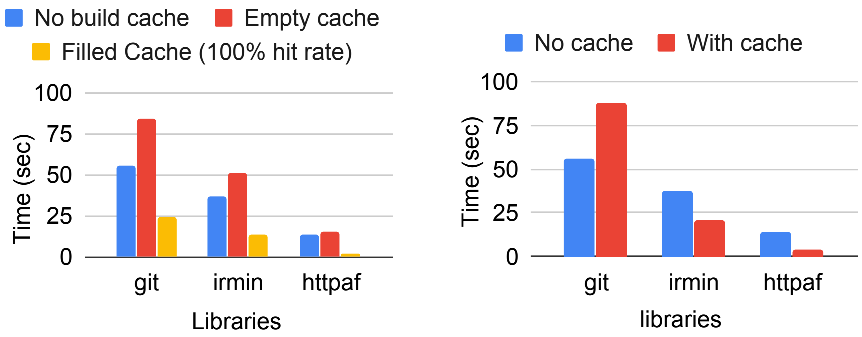
Fig. 11: Performance of mergeable logs.

Observe that unlike the examples seen so far where the values do not refer to other values, this *linked log* implementation refers to other values as heap data structures would do. Figure 11 shows the time taken to add 100 additional messages to the log with 4 clients. Observe that the time stays constant with linked log but increases linearly with blob log. By being able to share objects across different commits (versions), Banyan leads to efficient implementations of useful data structures.

5.4 Distributed build cache

In this section, we evaluate the performance of distributed build cache described in Section 2. We have chosen three OCaml packages: `git`, `irmin` and `httpaf` with common dependent packages. In the first experiment, we measure the benefit of building a package that has already been built in another workspace. Hence, the package artefacts will already be in the build cache.

For each library, we measure the baseline build time (1) without using the build cache, (2) using an empty build cache, and (3) building the same package on a machine with the same package having built earlier on a different machine.



(a) Performance of complete reuse of build artefacts.

(b) Performance of partial reuse of build artefacts.

Fig. 12: Performance of complete and partial reuse of build artefacts.

Figure 12a shows the results. We see that case using an empty build cache is slower than not using the cache since the artefacts are stored in the cache. We also see that building the same package on a different machine is faster due to the build cache when compared to the baseline.

A more realistic scenario is partial sharing of artefacts, where some of the dependencies are in the cache and other need to be build locally, and added to the cache. In this experiment, `git` package is first built on a machine with an empty cache. Subsequently, `irmin` package is built on a second machine (which will now benefit from the common artefacts in the cache). And finally, building `httpaf` on a third machine, which benefits from both of the builds. Figure 12b shows the results. As expected, the `git` package build is slower with a cache than without since the cache is empty and the artefacts need to be written to the cache additionally. Subsequent package builds benefit from partial sharing of build artefacts. The results illustrate that Banyan not only makes it easy to build complex applications like distributed build caches, but the implementation also performs well under realistic workloads.

6 Related Work

Several prior works have addressed the challenge of balancing the programmability and performance under eventual consistency. RedBlue consistency [28] offers causal consistency by default (blue), but operations that require strong consistency (red) are executed in single total order. Quelea [34] and MixT [31] offer automated analysis for classifying and executing operations at different consistency levels embedded in weakly isolated transactions, paying the cost of proportional to the consistency level. Indeed, mixing weaker consistency and transactions has been well-studied [10,25,4].

Banyan only supports causal consistency, but it is known to be the strongest consistency level that remains available [29]. While prior works attempt to reconcile traditional isolation levels with weak consistency, Banyan leaves the choice of reading and writing updates to and from other transactions to the client through the use of `publish` and `refresh`. We believe that traditional database isolation levels are already quite difficult to get right [22], and attempting to provide a fixed set of poorly understood isolation levels under weak consistency will lead to proliferation of bugs.

Banyan is distinguished by the equipping data types with the ability to handle conflicts (three-way merge functions). Banyan builds on top of Irmin [21]. Irmin allows arbitrary branching and merging between different branches at the cost of having to expose the branch name. Banyan refreshes and publishes implicitly to the public branch at a repository, which obviates the need for naming branches explicitly. Irmin does not include a distribution and convergence layer; Banyan uses Cassandra for this purpose. Banyan provides causal consistency and coordination free transactions over weakly consistent Cassandra. Several prior work have similarly obtained stronger guarantees on top of weaker stores [34,6].

TARDiS [14] supports user-defined data types, and a transaction model similar to Banyan. TARDiS is however a machine model that exposes the details of explicit branches and merges to the developer, whereas Banyan is a programming model that can be instantiated on any eventually consistent key-value store. For instance, in TARDiS programmers need to invoke a separate merge transaction that does an n-way merge. Banyan transaction model is more flexible than TARDiS. For example, Banyan can support monotonic atomic view, which TARDiS cannot – TARDiS transactions do not have a way of allowing more recent updates since the transaction began. TARDiS does not discuss merges without LCAs or the issue with recursive merges. We found recursive merges to be a very common occurrence in practice.

Concurrent revisions [12] describe a programming model with branch and merge workflow with explicit branches and restrictions on the shape of history graphs. Banyan makes the choice of branches to publish and refresh implicit leading to a simpler model. Concurrent revisions does not include an implementation. Antidote SQL [30] is a database system for geo-distributed applications that provides the user the ability to relax SQL consistency when possible, but remain strict when necessary. Similar to Banyan, Antidote SQL transactions are executed over replicated data types. While Antidote SQL only permits parallel snapshot isolation level [35], by making `refresh` and `publish` explicit, Banyan permits weaker isolation levels such as monotonic atomic view [5].

7 Limitations and Future Work

Many eventually consistent databases such as CosmosDB [4], DynamoDB [2] and Cassandra provide tunable consistency levels for operations ranging from eventual consistency to strong consistency. Banyan only provides causal consistency, which is known to be the strongest available consistency level, but does not provide weaker or strong consistency levels. As such applications that require strong consistency, such as bank accounts with a minimum balance requirement, cannot be expressed in Banyan. We believe that we can extend Banyan with strongly consistent operations. However, operations with weaker consistency (and presumably better performance) cannot be incorporated in Banyan due to the underlying expectation about the *causal history* for each operation.

We have yet to implement the garbage collector for Banyan based on the design sketched in Section 4.4. In the absence of a garbage collector, the storage requirements are quite significant compared to traditional databases which only store the most recent version of the data (Section 5.2). We leave the implementation of the garbage collector for future work.

8 Conclusions

We present Banyan, a novel programming model for developing loosely connected distributed applications based on the principles of Git. We illustrate the practicality of this approach by instantiating Banyan on Cassandra, an off-the-shelf eventually consistent distributed store. Our experimental results suggests that Banyan makes it easy to build complex distributed applications without compromising performance.

Acknowledgements

Parts of this research were funded by grants from the Tezos Foundation.

References

1. Abadi, D.: Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. Computer **45**(2), 37–42 (Feb 2012). https://doi.org/10.1109/MC.2012.33
2. Amazon DynamoDB: Fast and flexible NoSQL database service for any scale (2020), https://aws.amazon.com/dynamodb/
3. Apache Cassandra: The right choice when you need scalability and high availability without compromising performance (2020), https://cassandra.apache.org/

4. Azure CosmosDB: Build or modernise scalable, high-performance apps (2020), https://azure.microsoft.com/en-in/services/cosmos-db/
5. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly Available Transactions: Virtues and Limitations. Proc. VLDB Endow. **7**(3), 181–192 (Nov 2013). https://doi.org/10.14778/2732232.2732237
6. Bailis, P., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Bolt-on Causal Consistency. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 761–772. SIGMOD '13 (2013). https://doi.org/10.1145/2463676.2465279
7. Bazel: A fast, scalable, multi-language build system (2020), https://bazel.build/
8. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A Critique of ANSI SQL Isolation Levels. SIGMOD Rec. **24**(2), 1–10 (May 1995). https://doi.org/10.1145/568271.223785
9. Bernstein, P.A., Shipman, D.W., Wong, W.S.: Formal Aspects of Serializability in Database Concurrency Control. IEEE Trans. Softw. Eng. **5**(3), 203–216 (May 1979). https://doi.org/10.1109/TSE.1979.234182
10. Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.: Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In: Proceedings of the 44th ACM SIGPLAN Symposium on POPL. pp. 458–472. POPL 2017 (2017). https://doi.org/10.1145/3009837.3009895
11. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated Data Types: Specification, Verification, Optimality. SIGPLAN Not. **49**(1), 271–284 (Jan 2014). https://doi.org/10.1145/2578855.2535848
12. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually Consistent Transactions. In: ESOP 2012. pp. 67–86. ESOP'12 (2012). https://doi.org/10.1007/978-3-642-28869-2
13. Crain, T., Shapiro, M.: Designing a Causally Consistent Protocol for Geo-Distributed Partial Replication. In: Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data. PaPoC '15 (2015). https://doi.org/10.1145/2745947.2745953
14. Crooks, N., Pu, Y., Estrada, N., Gupta, T., Alvisi, L., Clement, A.: TARDiS: A Branch-and-Merge Approach To Weak Consistency. In: Proceedings of the 2016 International Conference on Management of Data. pp. 1615–1628. SIGMOD '16 (2016). https://doi.org/10.1145/2882903.2882951
15. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making Data Structures Persistent. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. pp. 109–121. STOC '86 (1986). https://doi.org/10.1145/12130.12142
16. Farinier, B., Gazagnaire, T., Madhavapeddy, A.: Mergeable Persistent Data Structures. In: Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015) (2015)
17. Gilbert, S., Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. SIGACT News **33**(2), 51–59 (Jun 2002). https://doi.org/10.1145/564585.564601
18. Git: A distributed version control system (2020), https://git-scm.com/
19. Gradle: An open-source build automation tool (2020), https://gradle.org/
20. Herlihy, M.P., Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (Jul 1990). https://doi.org/10.1145/78969.78972
21. Irmin: A distributed database built on the principles of Git (2020), https://irmin.org/

22. Kaki, G., Nagar, K., Najafzadeh, M., Jagannathan, S.: Alone Together: Compositional Reasoning and Inference for Weak Isolation. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158115
23. Kaki, G., Priya, S., Sivaramakrishnan, K., Jagannathan, S.: Mergeable Replicated Data Types. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360580
24. Kermarrec, A.M., van Steen, M.: Gossiping in Distributed Systems. SIGOPS Oper. Syst. Rev. **41**(5), 2–7 (Oct 2007). https://doi.org/10.1145/1317379.1317381
25. Kraska, T., Pang, G., Franklin, M.J., Madden, S., Fekete, A.: MDCC: Multi-Data Center Consistency. In: Proceedings of the 8th ACM European Conference on Computer Systems. pp. 113–126. EuroSys '13 (2013). https://doi.org/10.1145/2465351.2465363
26. Lakshman, A., Malik, P.: Cassandra: A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (Apr 2010). https://doi.org/10.1145/1773912.1773922
27. Lamport, L.: Paxos Made Simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) pp. 51–58 (December 2001), `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`
28. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., Rodrigues, R.: Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. pp. 265–278. OSDI'12 (2012)
29. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 401–416. SOSP '11 (2011). https://doi.org/10.1145/2043556.2043593
30. Lopes, P., a o Sousa, J., Balegas, V., Ferreira, C., 'e rgio Duarte, S., Bieniusa, A., Rodrigues, R., c c a, N.M.P.: Antidote SQL: Relaxed When Possible, Strict When Necessary. CoRR **abs / 1902.03576** (2019), `http://arxiv.org/abs/1902.03576`
31. Milano, M., Myers, A.C.: MixT: A Language for Mixing Consistency in Geodistributed Transactions. In: Proceedings of the 39th ACM SIGPLAN Conference on PLDI. pp. 226–241 (2018). https://doi.org/10.1145/3192366.3192375
32. Riak: Enterprise NoSQL Database (2020), `https://riak.com/`
33. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free Replicated Data Types. In: Symposium on Self-Stabilizing Systems. pp. 386–400. Springer (2011)
34. Sivaramakrishnan, K., Kaki, G., Jagannathan, S.: Declarative Programming over Eventually Consistent Data Stores. In: Proceedings of the 36th ACM SIGPLAN Conference on PLDI. pp. 413–424 (2015). https://doi.org/10.1145/2737924.2737981
35. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional Storage for Geo-Replicated Systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. p. 385400. SOSP '11 (2011). https://doi.org/10.1145/2043556.2043592
36. Viotti, P., Vukolić, M.: Consistency in Non-Transactional Distributed Storage Systems. ACM Comput. Surv. **49**(1) (Jun 2016). https://doi.org/10.1145/2926965