Mergeable Replicated Data Types

DAIMPL Project report - Summer semester 2024

Bennet Bo Fenner **Sebastian Spranger**

1. Motivation

Distributed Systems can be challenging in the best of times. The prevalent means of achieving distribution, Replicated Data Types (RDTs), force the implementation to concern itself with synchronizing single Operations, which can put considerable overhead and strain on converting or designing an application to be distributed.
Kaki et al. [1] envision a different more hands off approach to Distribution in their Paper 'Mergeable Replicated Data Types'. By defining a conversion from ordinary, base line data types to relations that are represented as sets and back from these set representations to the language specific data type, the tasks of synchronization and distribution can be decoupled and abstracted. These relations or set representations have the added benefit of being simple to combine in a way that ensures a consistent and especially convergent combine or merge operation. Using composites of these base mergeable data types virtually any type of data can be made distributed. This results in very minimal change to existing code or mental overhead for implementing new systems.
Their second key concept is a way to handle the replication side of the equation. Other than RDTs that only need to synchronize the operations performed, MRDTs need the complete objects as well as a common ancestor to be able to merge. Under the assumption that in most use cases most of the data will be shared between versions, storage and communication between replicas can be minimized by using an append-only content-addressable storage system that only has to handle the difference between replicas. This framework they call the Quark store, can be easily tailored to a specific application, as it is easily adapted to run on a network, file-systems, etc.

Implementing the conversion into the set representation for a handful of simple data types was, straight forward. The merging on its own wasn't that complicated either, but we ran into performance problems. After altering the merge function as well as tweaking the set representation, the performance of our implementation was reasonable.
Implementing the Quark store however was a bit more involved. We used ScyllaDB[1] as the underlying storage layer, to match what Kaki et al. [2] did in their follow-up paper 'Bolt-On Convergence in Mergeable Replicated Data Types'. However, we could not replicate one of their experiments, a simulation of a distributed text editing application, since our implementation of the Quark store turned out to be way too slow. The most significant factor affecting performance turned out to be the process of retrieving versioned data from the database through the traversal of a linked data structure. Kaki et al. provide insufficient details about this aspect of their implementation, making it difficult to fully understand or replicate their approach.

2. Replicate

2.1. MRDTs

2.1.1. Theory

The key idea of MRDTs is to represent common types, such as lists, maps and trees as specific relations (sets), and then use these representations for performing three-way merges, which converge. In order to perform a three-way merge two divergent states and the lowest common ancestor (LCA) are needed.

[1]https://scylladb.com

To make a type mergeable, Kaki et al. propose the following steps:

1. Compute the relevant relations from the data structure, for the three values (diverged states and their LCA)
2. Perform three-way merges on each of the relations, according to the formal definition
3. Reconstruct the data structure from the merged relations

Kaki et al. define different relations for a lot of common data types, let's take a look at how lists can be made mergeable. They introduce two relations called `mem` (Equation 1) and `ob` (Equation 2). `mem` is used for storing the individual objects in the list, `ob` is used for storing the ordering of elements. E.g. for a list $[c, a, b]$, the computed values of `mem` and `ob` would be $\{a, b, c\}$ and $\{(c, a), (a, b), (c, b)\}$.

$$\begin{aligned} R_{\text{mem}}(v) = {} & R_{\text{mem}}(\text{lca}) \cap R_{\text{mem}}(l) \cap R_{\text{mem}}(r) \\ & \cup R_{\text{mem}}(l) - R_{\text{mem}}(\text{lca}) \\ & \cup R_{\text{mem}}(r) - R_{\text{mem}}(\text{lca}) \end{aligned} \tag{1}$$

$$\begin{aligned} R_{\text{ob}}(v) \supseteq {} & (R_{\text{ob}}(\text{lca}) \cap R_{\text{ob}}(l) \cap R_{\text{ob}}(r) \\ & \cup R_{\text{ob}}(l) - R_{\text{ob}}(\text{lca}) \\ & \cup R_{\text{ob}}(r) - R_{\text{ob}}(\text{lca})) \\ & \cup (R_{\text{mem}}(\text{lca}) \times R_{\text{mem}}(\text{lca})) \end{aligned} \tag{2}$$

Applying these two relations to a three way merge of a list, preserves the intent of the user. Meaning that, items that are removed or added in one version, will also have an effect on the output of the three-way merge, either by being excluded or included in the merged version. See Figure 1 for a visualization of this behavior.

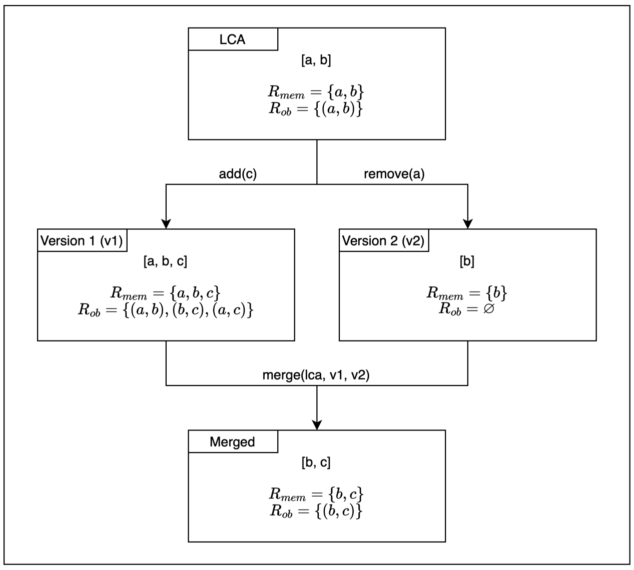

Figure 1: Three-way merge performed on list using `mem` and `ob` sets to preserve intent

2.1.2. Implementation details

For our evaluation we only implemented MRDTs for lists in both Scala3 and Rust. Let's take a brief look at how we implement an MRDT list in Rust. First, we abstract the `merge` function behind a trait.

```rust
pub trait Mergeable {
  fn merge(lca: &Self, left: &Self, right: &Self) -> Self;
}
```

This makes it easy to add three-way merge support to any type. Here's a simplified example of implementing the `Mergeable` trait for `Vec`, utilizing the proposed `mem` and `ob` set variants.

```rust
impl<T: Clone + Ord> Mergeable for Vec<T> {
    fn merge(lca: &Self, left: &Self, right: &Self) -> Self {
        let mem: HashSet<&T> = merge_mem(mem(lca), mem(left), mem(right));
        let ob: Vec<(&T, &T)> = merge_ob(ob(lca), ob(left), ob(right), &mem);
        ob.into_iter().map(|(n, _)| n.clone()).collect()
    }
}
```

Listing 2: Mergeable implementation for the `Vec` type

On each call to merge (Listing 2), we first construct the `mem` and `ob` sets from the three states (`lca`, `left`, `right`), then we apply the three way merge logic defined in Equation 1 and Equation 2. Finally, we have to reconstruct the `Vec` from the `mem` and `ob` sets.

2.2. Quark Store

In order to share MRDTs across replicas, Kaki et al. propose a new storage layer called the Quark store. The Quark store is a content-addressable distributed storage abstraction, which exposes a Git like API with operations such as cloning a remote repository, commiting changes and performing three-way merges.

For our evaluation we implemented a version of the Quark store in Rust, we use ScyllaDB as the underlying storage engine which takes care of replication.

```sql
CREATE TABLE replica
    (id TEXT, commit_id TEXT, PRIMARY KEY (id));
CREATE TABLE commit
    (id TEXT, version BLOB, root_ref BIGINT, prev_commit_id TEXT,
      PRIMARY KEY (id));
```

Listing 3: Database schema for storing commits

The `replica` table (Listing 3) stores the id of the latest commit of each replica. Each commit contains a version (`Vector clock`) and a way to retrieve the state associated with the commit (`root_ref`). We store the id of the previous commit (`prev_commit_id`), which allows us to traverse the commit graph. In order to perform a three-way merge, we can utilize the version `Vector clock` to retrieve the lowest common ancestor (LCA) between two commits. We retrieve the LCA by finding the intersection of these two maps, when a key is present in both vector clocks, we use the minimum value. After determining the LCA version, we can use the storage abstraction to locate the relevant commit, retrieve the objects and perform a three-way merge.

In order to store objects (Listing 4) we serialize them to a binary format and store their hash as the key. This allows us to store objects in a content-addressable way, where the key is the hash of the object and the value is the object itself.

```sql
CREATE TABLE object
    (id BIGINT, object BLOB, PRIMARY KEY (id));
```

Listing 4: Database schema to store objects in a content addressable way

To address the scalability issue of storing entire application states on each commit, Kaki et al. propose storing only the diffs between versions. This is achieved by using a linked data structure, where individual nodes are inserted into the database. While the exact structure isn't described in detail in their paper, our implementation uses a binary tree to store nodes. On each commit, we traverse a structure (e.g. a list), serializing it into individual nodes that store references (hashes) to the actual objects. These nodes are then inserted into the `ref` table.

```sql
CREATE TABLE ref
    (id BIGINT, left BIGINT, right BIGINT, object_ref BIGINT, PRIMARY KEY (id));
```

Listing 5: Database schema to store references to objects

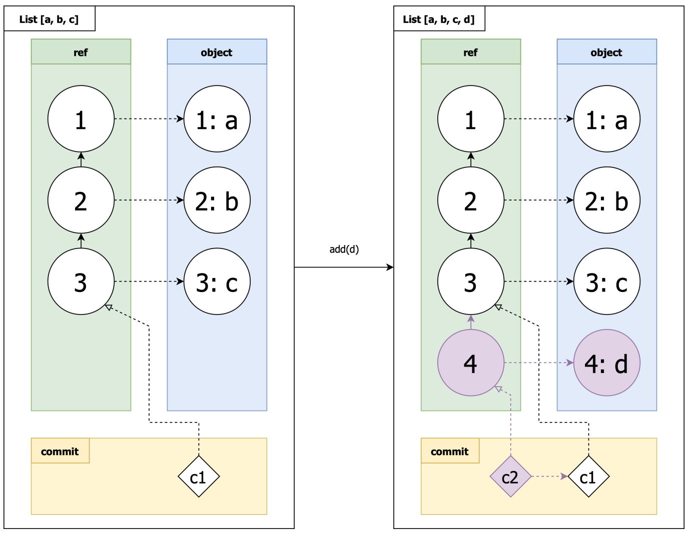

Figure 2: State of the quark store after adding an element to a list. Nodes with a purple background need to be replicated

To store list's in the database our `serialize` function generates `ref` nodes [Listing 5] for each element in the list, the `left` column value is used to built up a linked-list-like structure. As seen in Figure 2, instead of replicating the whole state, storing only the diffs of node's when adding an element to the end of the list, can be handled efficiently. However, to insert a new object in the middle of the list, all the references on the path from the root to the new object need to be inserted and replicated. See Figure 3 for an visualization of this behavior. So in combination with the quark store representation, lists are not ideal for state sharing and therefore minimizing the amount of data that needs to be replicated, however Kaki et al. claims that using tree structures can mitigate this effect. Now that we serialized the data to the database, we need a way to re-compute the whole state by looking at the `ref` table. For a given commit, we start by resolving the `root_ref` from the `ref` table. We then traverse the `left` and `right` fields to recursively gather all the relevant nodes that build up the binary tree in the quark store, and resolve objects from the `object` table given it's hash (stored in the `ref.object_ref` field), see Listing 6 for a simplified example of our implementation.

```rust
impl<T> Deserialize for Vec<T> {
    fn deserialize(root: Ref, cx: DeserializeCx<'_>) -> Result<Self> {
        let mut items = vec![cx.resolve_object(root.object_ref)];
        let mut node = root;
        while let Some(new_node) = cx.resolve_ref(node.left)? {
            items.insert(0, cx.resolve_object(new_node.object_ref));
            node = new_node
        }
        Ok(items)
    }
}
```

Listing 6: Reference implementation of resolving a list of items from the Quark content abstraction

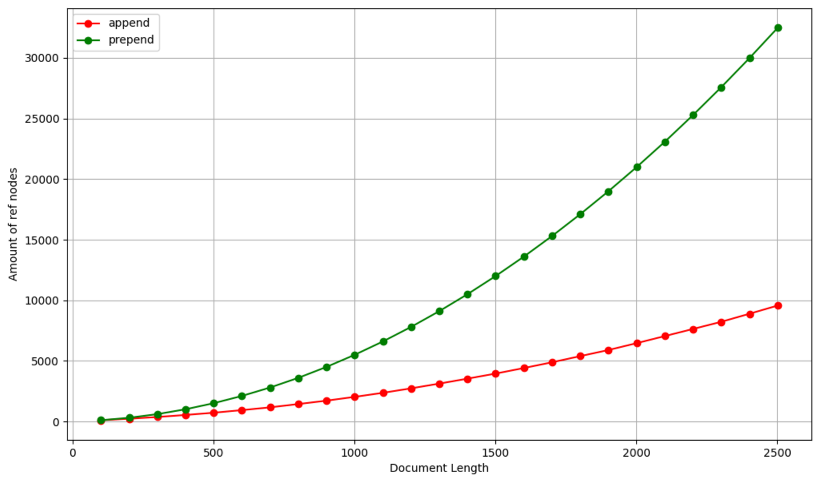

Figure 3: Amount of `ref` nodes inserted by the quark store when appending characters near the end (25%) vs inserting near the start (25%) of a list

3. Evaluate

3.1. Benchmarking our mergeable list implementation

During testing, we discovered that creating `ob` sets from our intermediate representation was inefficient due to quadratic growth in set size.

For example, given a list $[a, b, c, d]$, the `ob` set as defined by Kaki et al. would be $\{(a, b), (b, c), (c, d), (a, c), (a, d), (b, d)\}$. The set size grows by $\frac{n(n-1)}{2}$, approximately doubling with each additional list element.

To optimize this, we implemented a strategy to store only direct neighbors in the `ob` set. For the above example, we only store $\{(a, b), (b, c), (c, d)\}$, reducing the set size to $N - 1$ elements for a list of N items.

While this approach works for appending elements, it fails when elements are removed in one version, as we lose the link between nodes that are not direct neighbors. To address this, we added a second step to our optimization strategy: Instead of only using set operations like unions and intersections, we use the `ob` sets to generate a graph using `topological sorting` and then remove any node that is not contained in the `mem` set from that graph. By removing the elements at the end, we can compute the transitive edges easily because of our graph representation. By using `depth-first-search` to generate our `topological sorting`, we can guarantee a linear time complexity.

$$O(|\text{mem}| + |\text{ob}|) \rightarrow O(|\text{mem}| + (|\text{mem}| - 1)) \rightarrow O(|\text{mem}|)$$

Another added benefit of this approach is the ease of handling cases were there is no clear order like in Figure 4. In the paper, they have to add extra edges to resolve unclear orderings. Where as with our implementation by simply sorting the worklist i.e. the `ob` set, the `depth-first-search` generates the consistent sorted graph, with no extra steps.

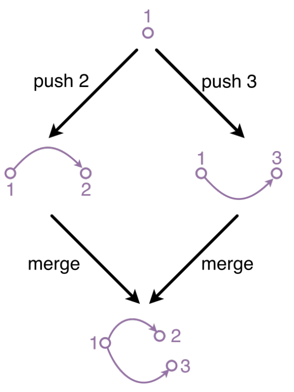

Figure 4: To resolve all possible merges you have to have an ordering for the objects you store. This ordering can be arbitrary, but must be consistent between replicas

After optimizing the `ob` set implementation, we observed that the performance of merges was reasonable, e.g. merging lists with 10k / 100k elements takes around 10 ms / 150 ms respectively[2].

3.2. Benchmarking our Quark store implementation

In our attempt to benchmark our implementation by simulating a collaborative editing application, we encountered significant challenges in replicating the experiment conducted by Kaki et al. in [3]. The primary obstacle in replicating the experiments was that the performance of our implementation was simply way too slow.

As mentioned above, we use a binary tree where each node points to two children to store nodes in the Quark store. We found that over 95% of merge time was consumed in resolving nodes from the `ref` table. For our list representation, this process requires N database queries to resolve all items, followed by additional queries for each node to resolve its referenced object (`object_ref`).

We made efforts to optimize by utilizing batch statements, which allowed us to resolve multiple objects simultaneously, thereby reducing the number of database queries needed for object resolution from given `ref` nodes. However, we were unable to apply this optimization to the process of resolving references. This limitation is enforced by the sequential nature of the process: we must query each reference to determine which nodes it points to before proceeding to the next.

It is unclear how Kaki et al. managed to achieve their results, unfortunately there is no sufficient information included in the paper [3] on how this particular part of their implementation works.

4. Findings

To conclude, the merging and conversion between different representations of the data seem mathematically sound and were easy to implement for various data types. We also achieved reasonable performance once we applied our optimization. Understanding the Quark store concept proved to be a challenge for us. While the underlying idea is not inherently complex, Kaki et al. approach is not described in enough detail, which required us to piece together information from different papers. In the end our implementation turned out to be too slow for using it in any real world application. The primary bottleneck was the process of resolving a linked data structure from the database (Section 3.2).

What left us wandering was, if there is a way of generalizing the Quark store. Storing objects in a linked data structure, can be as simple as a linked list. But as soon as the data types become more complex, i.e. graphs or mergeable types that itself contain other mergeable types, there might not be such a trivial 1:1 conversion into a linked data structure. The issue we see is not the problem of finding

[2]Benchmarks were run on an Apple M2 Pro chip, with 16 GB of memory

a linked representation, but that for every data type you want to implement you have to implement not only a conversion to the set representation, but also have to fiddle with the Quark Store.
One possible solution would be to use the already existing set representation and store that in the Quark Store. This would be trivial to implement, as one would only ever have to store sets and tuples. But as you can see in the example (Figure 5) of a simple queue or list, there would be a nonnegligent overhead.

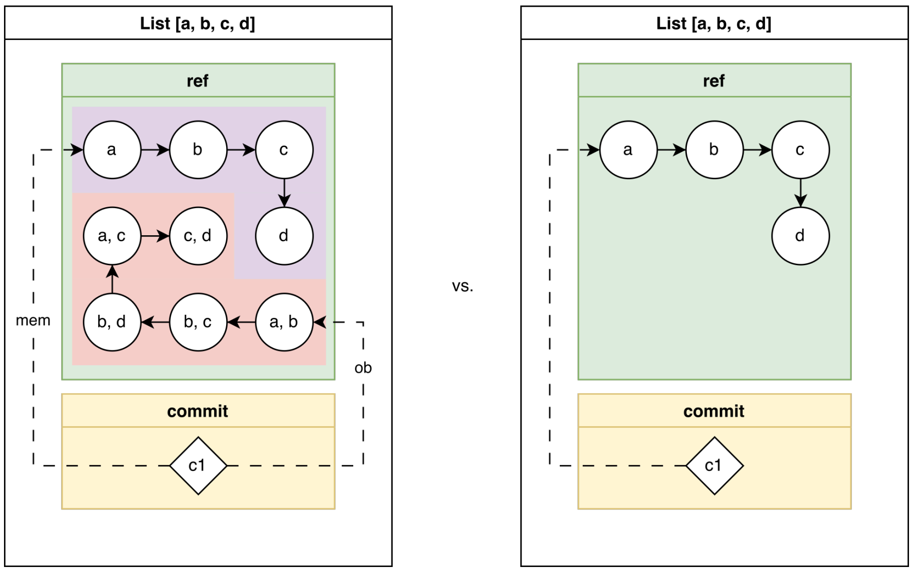

Figure 5: *left* storing both *mem* and *ob* set; *right* using the already present links to store structure. Note that the *ob* set grows by a factor of $O(n^2)$, $n :=$ number of entries in the list

It makes much more sense to use the structure we necessarily have by using links, to represent the structure and relation of the data. This however depends on what data you are storing, bringing us back to the problem of not having a generalized way to store data in the Quark Store.

Another thing we wondered about was the problem of an ever increasing amount of data you have to store. Since the Quark store is *append only* and you have to be able to access every old version to be able to merge every possible commit, the space requirements could only ever increase with the runtime. So the Quark Store necessarily has to implement a way of cleaning up and removing old information. This however is not discussed in a meaning full way in the paper.

We were also wondering, since the merging happens abstracted away by using the set representation, if this approach looses some fine control one would get by using conventional RDTs, but we didn't investigate that further and it is totally possible this is not an issue in practical use or it may not be the case in general.

Bibliography

[1] G. Kaki, S. Priya, K. Sivaramakrishnan, and S. Jagannathan, "Mergeable replicated data types," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019, doi: 10.1145/3360580.

[2] G. Kaki, P. Prahladan, and N. V. Lewchenko, "Bolt-On Convergence in Mergeable Replicated Data Types," [Online]. Available: https://api.semanticscholar.org/CorpusID:265223380

[3] G. Kaki, P. Prahladan, and N. V. Lewchenko, "RunTime-assisted convergence in replicated data types," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, in PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 364–378. doi: 10.1145/3519939.3523724.