DSaaS
A Cloud Service for Persistent Data Structures

Pierre Bernard le Roux[1], Steve Kroon[1,2], and Willem Bester[1]
[1]*Computer Science, Stellenbosch University, Stellenbosch, South Africa*
[2]*CSIR/SU Centre for Artificial Intelligence Research*
pierrebleroux@gmail.com, kroon@sun.ac.za, whkbester@cs.sun.ac.za

Keywords: DaaS, SaaS, Cloud Computing, Persistent Data Structure, Version Control System, Hash-Array Mapped Trie

Abstract: In an attempt to tackle shortcomings of current approaches to collaborating on the development of structured data sets, we present a prototype platform that allows users to share and collaborate on the development of data structures via a web application, or by using language bindings or an API. Using techniques from the theory of persistent linked data structures, the resulting platform delivers automatically version-controlled map and graph abstract data types as a web service. The core of the system is provided by a Hash Array Mapped Trie (HAMT) which is made confluently persistent by path-copying. The system aims to make efficient use of storage, and to have consistent access and update times regardless of the version being accessed or modified.

1 INTRODUCTION

Collaboration on structured data can be difficult, time-consuming, and frustrating. For structured data sets, this usually involves creating multiple copies of the data and using primitive forms of versioning on the data set, such as keeping copies of various versions on multiple computers with different names in different directories. These approaches often lead to unnecessary duplication, inconsistencies and inaccuracies in the data.

A more sophisticated approach is to use a Version Control System (VCS). While this can alleviate some of the problems sketched above, most VCSs are designed to version control text documents and source code, where the lines in a file constitute the units between which deltas—differences in content—are calculated. Using source code VCSs for other data works relatively well when the data can be represented efficiently in tabular form, such as the comma-separated values (CSV) format understood by many data analysis applications (Pollock, 2015). However, in other cases navigating the trade-off between storing large files and the time impact of resolving long delta chains is non-trivial for large data sets (Bhattacherjee et al., 2015).

1.1 System Overview

Web-based data services allow developers to read and write data from a central, shared data service. The core values of these services are that they allow multiple clients to concurrently access and modify the data from anywhere. A developer can then write a network application—for example, a mobile or a browser-based application—that accesses and modifies these data structures.

The prototype system we introduce here, referred to as DSaaS[1] (from "Data Structures as a Service"), attempts to address the shortcomings of traditional structured data collaboration techniques by providing a platform for users to share data structures, and to collaborate on their development and maintenance. By using techniques from the theory of persistent linked data structures, the platform delivers automatically version-controlled abstract data types in a manner similar to a web-based data service.

In particular, DSaaS currently provides access to cloud-based map (i.e., symbol table) and directed graph data structures. The map structure was chosen because it is the foundational data structure for some languages, such as objects in JavaScript (Kantor, 2015) and dictionaries in Python (Python, 2015), and can be used to implement the array and directed

[1]DSaaS can be accessed at `http://cs.sun.ac.za/~kroon/dsaas`.

graph abstract data types. The directed graph data structure was developed as a proof of concept of adding a new data structure to the system by leveraging the map implementation, and facilitates a number of other linked structures which are special cases of graphs. DSaaS users can create and share data structures, and can view and modify any previous version of any of their data structures via the web interface or language bindings—currently, a prototype Java language binding is provided. Language bindings aim to facilitate interaction with the data structures stored in the system by enabling existing code to make use of the service with only minimal changes relative to previous use of language-specific data structures. Both the web interface and the language bindings access most of the functionality of the DSaaS system via a RESTful API (Fielding, 2000), which developers can also use directly.

1.2 Use cases

DSaaS aims at fine-grained version control for structured data sets. Below, we outline a few potential application areas that could benefit from such a system. Note that some of these applications could be enabled using a VCS, but as discussed earlier, such solutions are not ideal.

First, the approach used in DSaaS enables scientific studies to enhance reproducibility by reporting the exact version of data used for an experiment or computation, by providing intermediate results after various processing or preprocessing steps, and by allowing verification of a computation by investigating the development of data structures used during its execution. The latter can be viewed as an audit trail for a computation.

The same audit trail perspective highlights potential uses in teaching programming and debugging. DSaaS could be used to illustrate how data structures function by replaying the evolution of a data structure as various operations are performed on it. Similarly, it could be used as a debugging tool, since it has the ability to visualize the data structure as it evolves during use. A benefit of this approach is that debugging could be performed offline, without the use of watches and breakpoints to interrupt the code execution.

Since the API and language bindings allow DSaaS data structures to be used in session-based interpreters, such as the IPython interpreter (Pérez and Granger, 2007), these data structures can be conveniently shared and accessed from multiple interpreters in multiple languages, without the usual developer's overhead of serializing and deserializing, or parsing.

1.3 Paper Outline

Section 2 primarily introduces important terminology and concepts necessary to understand the system implementation. The general architecture of the system is discussed in Section 3, after which Section 4 details selected technical aspects of its operation. A major challenge for DSaaS is providing efficient access to and modification of all historical versions of a data structure while making efficient use of storage. We thus present some preliminary experimental results in Section 5, illustrating the system's current performance and comparing it to another system with similar aims. An appendix outlines some system features and capabilities outside the main scope of the paper for the interested reader. Note that a project report (le Roux, 2015) contains expanded discussions of various aspects of DSaaS presented in this paper.

2 BACKGROUND

This section begins by discussing our system's place in the landscape of cloud services. Thereafter the core ideas of persistent data structures are introduced, including an overview of path-copying, a classical technique for obtaining persistence of data structures. Finally, we introduce the Hash Array Mapped Trie (HAMT), the key data structure used in our implementation.

2.1 Cloud Service Models

DSaaS combines the Data as a Service (DaaS) and Software as a Service (SaaS) cloud service models. It provides DaaS since the data structures it manages are stored and maintained by the service, while it provides SaaS because of the service's web-based mechanisms for manipulating the data structures. In this sense, it is somewhat similar to the GitHub (GitHub, 2015) and Bitbucket (Bitbucket, 2015) services. However, a notable difference is that while these services allow centralised editing of the source code, it is not the dominant approach for doing so. In contrast, DSaaS maintains the data structures centrally, with updates performed via the web interface and the API.

Other services combining the DaaS and SaaS models are collaborative document editing services, such as the office suite component of Google Drive (Google, 2015), where users can share, view, and collaborate on documents from multiple devices. This is almost exactly the type of service DSaaS aims to provide, except that the objects stored are data structures rather than specific document types. Besides that, most collaborative document editing services primarily offer partial persistence of documents, while DSaaS provides confluent persistence of data structures; see Section 2.2.

The Dat project (Ogden, 2015b) provides version control for data sets, but with an architectural approach different to the client–server model we employ. The similarities are the fine-grained access control and a dedicated API for managing the data.

2.2 Persistent Data Structures

We next discuss the idea of persistence for data structures, its various forms, and a technique of persisting pointer-based data. Persistent data structures have found use in many applications such as functional programming languages, computational geometry, pattern matching, etc. (Straka, 2013). The terminology we use below is based on (Driscoll et al., 1986).

A data structure that does not provide access to its history is called an *ephemeral* data structure. Ephemeral data structures are changed in-place as updates occur, and only the most recent version is ever available. Local library data structure implementations, such as those in the Java Collections Framework (Watt and Brown, 2001), are typically ephemeral.

In contrast, a *persistent data structure* is a data structure that provides access to different versions of a similar (in the sense of expected operations) ephemeral data structure. For example, a persistent map data structure provides access to different versions of a map data structure. When the data structure is created, it stores only an empty initial version. Subsequent updates lead to new versions of the underlying ephemeral data structure, all of which are stored implicitly in the persistent data structure. These persistent data structures play a central role in functional programming languages because they provide an efficient approach to implementing immutability.

2.2.1 Forms of Persistence

There are various forms of persistence:

Partial persistence permits queries on all the versions, but only allows modifications to the most recent version. The version history then forms a sequence of versions, ordered with respect to temporal evolution.

Full persistence allows queries of and modifications to all previous versions of the data structure. With full persistence the version history forms a tree where any path from the root to a leaf is ordered by temporal evolution.

Confluently persistent data structures are fully persistent and also support a *merge* operation that allows two previous versions of the data structure to be combined to create a new version of the data structure. Here the version history forms a directed acyclic graph (DAG).

Our approach in DSaaS was to develop a confluently persistent map data structure, and use it to implement a confluently persistent directed graph data structure. This means that the entire development history of multiple versions of each data structure is always available, and these versions can be merged to create new versions. This allows multiple users to work on the same data structure concurrently.

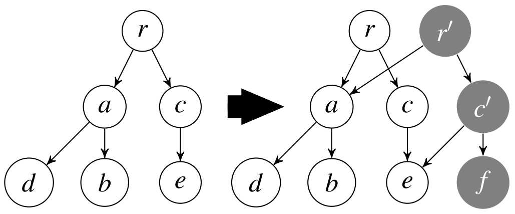

Figure 1: An example of path-copying in a pointer-based tree structure. To add a new node f as a child of node c, a new version is created that can access the new nodes while still referencing elements from the old version which remain unchanged.

2.2.2 Path-copying

Path-copying (Driscoll et al., 1986) is a technique for persisting pointer-based data structures based on the insight that an update of such a data structure will typically only affect a small subset of the nodes in the structure. The path-copying method is thus able to maintain both the original and new versions of the data structure by duplicating only the affected nodes. When a node is duplicated in a new version, all nodes referencing the previous node must also be modified to reference the new version. Typically, this leads to the creation of paths of duplicated nodes, hence the name of the technique. An example where a new node is added to a tree is illustrated in Figure 1. As can be seen in the figure, a new root r' and another new node c' are created when the node f is added. These allow the developer to access the original version of the data structure from root r and the new version from root r'.

2.3 Hash Array Mapped Trie (HAMT)

An HAMT is an implementation of the map data type that allows fast retrieval of key–value pairs while using memory efficiently (Bagwell, 2001). Given a

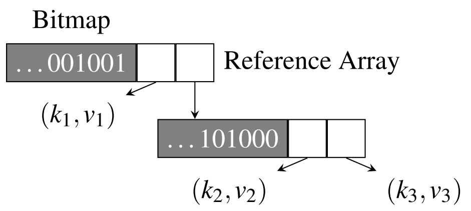

Figure 2: An example of an HAMT. The grey blocks represent the bitmaps, and the white cells represent the array of references to key–value pairs stored in this trie.

key–value pair (k, v), the HAMT applies a hash function h to k, and stores the resulting hash $h(k)$ as an entry in a compressed trie structure, with the value associated with that $h(k)$ being the original key–value pair (k, v). To enable use of a trie structure, the hash $h(k)$ must be represented as a sequence of characters from some alphabet. This is done in a straightforward manner by chunking $h(k)$ into groups of 5 bits. The HAMT then uses a bitmap in each node to indicate which children are non-null, while the non-null references are stored in a resizable array. These array elements either refer to other HAMT nodes or directly to key-value pairs. We illustrate some of these aspects in the example below—for more information on the HAMT, see (Bagwell, 2001).

Example 1. Figure 2 gives an example of a two-node HAMT storing three key–value pairs. For each node, the bitmap shows the lower-order bits and the elements of the reference array corresponding to the set bits. For accessing the HAMT, we chunk the output of the hash function h from left to right, but positions in the bitmap are numbered from right to left.

Assume $h(k_1)$ has `00000` as its first five bits, and therefore, the numeric value of the first five bits is 0. Therefore, to retrieve v_1, one must check if position 0 in the bitmap (i.e., the rightmost position) is set, and calculate the number of lower-order bits (i.e., bits to the right of the position) set in the bitmap to identify which element of the reference array to access. In the figure, position 0 is set, and the number of lower-order bits set is 0, so that the key–value pair (k_1, v_1) is stored at position 0 in the reference array.

Now, assume $h(k_2)$ starts with `00011 00011`, and $h(k_3)$ with `00011 00101`. To retrieve v_3, one first checks whether position 3—the binary value of the first five bits of $h(k_3)$—in the root node's bitmap is set, and then one again calculates the number of lower-order bits set to find the position of (k_3, v_3) in the reference array. In the figure there is only one lower-order bit set; therefore the search for v_3 continues to the node referenced in the root's reference array at position 1. This node is queried in a similar way, but using the next five bits of $h(k_3)$ to discover that the bit for position 5 is set, and only one lower-order bit is set, so that (k_3, v_3) can be found at the second element of the reference array.

Since the HAMT is essentially a tree structure, it can be converted into a persistent data structure by path-copying. This is an essential component of the current DSaaS system.

3 ARCHITECTURE

DSaaS incorporates both client- and server-side software. The server provides both static content (mainly client-side software) and dynamic content to the user. Client-side software includes a client-side web application, as well as language bindings that connect to the server via the API service. All static content is available over HTTP, and the dynamic content is available through both HTTP and web-sockets. The dynamic content is handled by the API service, which provides access to all the system features.

When a browser visits the base DSaaS URL, the web application is downloaded. The web application then uses the API service to communicate further with the server. Third-party systems can also communicate with the API service by either using the language bindings, making their own HTTP calls to the API service, or by implementing their own integration layer to connect with the provided socket.

The next subsection discusses the client tools available for accessing the service, focusing on the Java language binding. Thereafter, the components of the API service are considered.

3.1 Client Tools

The client tools currently included in the system are the browser-based client application and the Java language bindings. The browser-based client application allows users to share, inspect and manage the contents and properties of data structures to which they have access. The application, which follows a design inspired by the Flux architecture (Facebook, 2015), is served from a static HTTP server and consists of a combination of JavaScript, HTML, and CSS files.

While many operations can be performed via the client application in principle, it is impractical to use the browser for many updates to large-scale data structures. Thus it is desirable to provide a more automated mechanism for interacting with the data structures in DSaaS. As a proof of concept, a prototype language binding has been implemented for Java. This language binding is essentially a wrapper around the HTTP and web-socket protocols that allows the

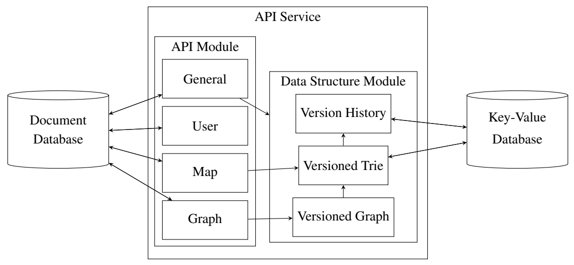

Figure 3: Data communication between the components of the API service and data stores.

data structure updates and query calls to be processed as a Java application runs.

The aim of language bindings for the service is to provide access to the service with the least possible effort from a developer's perspective. With this approach, the developers need only initially specify which data structures and versions they wish to use—this process is similar to checking out a version in a Git repository in that it establishes the base version to which future operations are applied. Once this has been done, the binding encapsulates the data structure in a class exposing an API consistent with that language's traditional libraries for manipulating and querying the data structure; in this way, the checked-out version appears to the developer to be an instance of the regular (ephemeral) data type.

For example, the Java DSaaS language binding exposes the map data structure in a class implementing the `java.util.Map` interface. For the directed graph, there is no standard Java interface, and therefore, we defined a suitable interface.

The language binding library automatically fetches data from the service as the local system needs it. Lazy local caching of portions of the data structure is performed to improve performance.

3.2 API Service

We now consider in more detail the architecture of the API service used by the browser-based client application and library bindings. Figure 3 shows the components of the API service, which are grouped into an API module and a data structure module, each of which is discussed below.

3.2.1 API Module

All operations and features supported by the data structures are accessed through API requests. Thus all HTTP requests that interact with the data structures, as well as user management requests, are handled by the API module. This module is also responsible for maintaining a consistent representation of the data on the client by using web-sockets (Fette and Melnikov, 2011).

The API module consists of multiple handlers. There are general, graph, map, and user handlers for the different request types and socket subscriptions. Each handler is responsible for validating and responding to requests, logging API calls, and keeping track of analytics. The user handler interacts directly with the document database (discussed in Section 3.2.3), whereas most other handlers interact with both the document database and the data structure module. The general handler also has the responsibility of pushing real-time version history updates to subscribed clients via web-sockets.

3.2.2 Data Structure Module

The data structure module consists of versioned trie, versioned graph, and version history submodules.

The versioned trie, based on a persistent HAMT obtained using path-copying, is the underlying structure for exposing a confluently persistent map data structure to the API. This map implementation provides the usual operations of inserting, replacing, or removing a key–value pair, retrieving the value associated with a given key, and retrieving a collection of values, keys, or key–value pairs. The operations for retrieving collections have additional functionality for retrieving a range of elements, as opposed to retrieving all elements in one request. All these operations operate on a specific version of the data structure, and return either the requested information, or in the case of an update, an identifier for a new version of the data structure.

Similarly the versioned graph—which is implemented using the versioned trie as a symbol table to represent the nodes and the edges of the graph—is the underlying structure for exposing a confluently persistent directed graph structure to the API. This directed graph data type provides a decorated graph by allowing the storage of additional data as attributes of nodes and edges. Typical graph operations, such as adding and removing a node or an edge, updating a node's or an edge's attributes, and retrieving a collection of nodes or edges, are available, and again, operate on a specific version of the data structure.

Finally, the version history is used for recording the relationships between the various versions of each data structure and is implemented as a directed acyclic graph. The version history also serves as a visual aid in the web application, helping the user to navigate the potentially complex connections between versions that may arise during use of the data structures.

3.2.3 Databases

The primary database used by DSaaS is a NoSQL key–value data store. For the prototype, LevelDB (Google, 2015) is used because it is a fast database, and stores keys and values as strings. This makes it easy to store the JSON representation of the data structures.[2]

The reason a key–value store is used is that the database should be fast, highly scalable, and as close as possible to having a distributed array. This makes translating the data structures from in-memory data structures to stored data structures less complex. Traditional relational databases are unnecessary, providing superfluous functionality.

For the metadata and user data, greater querying functionality is required to simplify the design; here a traditional relational database works well. MongoDB (Membrey et al., 2010) was used as a document store for storing data related to the user management, data structure metadata, and logs.

4 TECHNICAL DETAILS

The two most essential components of DSaaS are the versioned trie and the version history modules. These are vital to the system's performance, because they provide the underlying data structure and operations necessary for the versioning and storage of the data.

Our versioned trie implementation is inspired by the Clojure implementation of a persistent HAMT data structure which is used as its core functional map data structure (Clojure, 2015). As in Clojure, we use path-copying for obtaining persistence of the HAMT. Our implementation differs from a straightforward Javascript port of the implementation used by Clojure in a number of ways: (a) it is implemented to work on storage instead of directly in memory; (b) it adds a merge operation that allows combining two versions to provide confluent persistence; (c) it identifies transpositions of the same data structure (instead of viewing the same trie obtained through a different sequence of operations as different); and (d) it keeps references to the various versions in a navigable graph structure explicitly by means of the version history module instead of doing so implicitly. These differences are discussed in more detail below. We end the section with a discussion of how the versioned graph is built using the versioned trie, and considerations for adding new data structures to DSaaS.

[2]Note that while other NoSQL databases might be more suitable for large-scale deployment, our current interest is purely in the feasibility of the approach.

4.1 Moving to Storage

Modifying the original HAMT implementation to a storage-based implementation was done by writing the data to storage as they are created and only fetching the necessary values from storage when they are needed. By employing a key–value store, this process was fairly straightforward, although it required giving unique identifiers for all objects stored in the key–value store. The identifiers used were either hashes of key–value pairs for leaf nodes or a uniquely generated identifier for internal nodes.

4.2 Merging

The system also provides a three-way merge operation (Santos, 2013) for automatically combining two different versions, avoiding tedious manual application of the primitive data structure operations to one of the versions to create a combination of the two versions.[3]

The result of the merge operation can be described as follows. Given two versions L and L', sharing a common ancestor K, a set of updates $K \rightsquigarrow L$ was applied to K to create L; similarly, $K \rightsquigarrow L'$ was applied to K to create L'. To merge L into L', the set of updates $K \rightsquigarrow L$ must be applied to L'. However, this may lead to merge conflicts, for example, if L and L' both contain the same key, not originally in K, with different values.

DSaas deals with conflicts arising from the operation merge(L, L'), merging L into L', by giving precedence to the data in the first argument L, which is called the *principal version*—that is, the data in L overrides the data in L' (and K). Note that this means the merge operation is not symmetric in its arguments in the presence of merge conflicts.

Example 2. An illustration of the merge process is given in Figure 4, where the map records different versions of a food order for a restaurant customer: Map versions B and C are both derived from a common ancestor version A. The restaurant software now requests a merge of these two versions. Relative to the common ancestor A of versions B and C, it is evident that, to create version B, the value for the key *Food* was changed from "Burger" to "Pizza", whereas to obtain version C, the key–value pair (*Drink*, "Soda") was removed and the key–value pair (*Extra*, "Cheese") was added (i.e., these are the

[3]This merge operation differs considerably from that proposed in the original literature on confluent persistence (Kaplan, 1995), however: in particular, while their proposed operation allowed generating data structures with size exponential in the number of operations, ours does not.

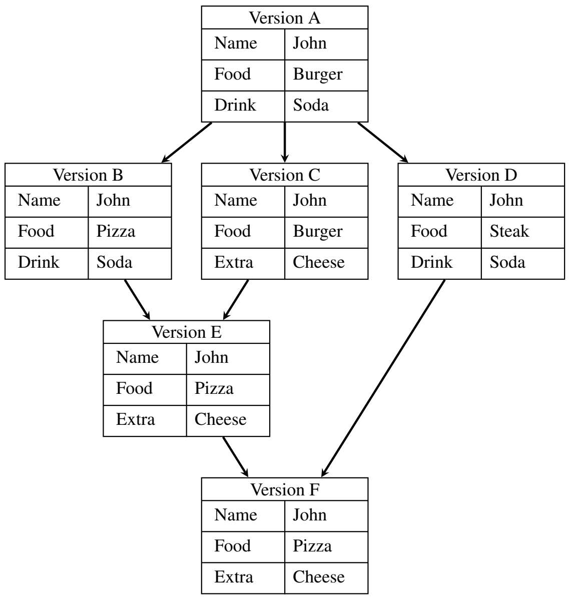

Figure 4: An example of the three-way merge used in DSaaS. To merge versions B and C, the changes made since the common ancestor version A are taken into account. A merge involving versions D and E involves a conflict, and the result depends on which version is considered the principal version.

changes in $A \rightsquigarrow C$). The final merged version E takes all these changes into account. Since there are no conflicts between the edits, the result will be the same regardless of whether B is merged into C, or vice versa.

Example 3. Map version D is derived from version A by changing the key–value pair (*Food*, “Burger”) to (*Food*, “Steak”). A merge involving versions D and version E features a conflict, as the value of *Food* is different in the two maps, and both differ from the value of *Food* for their common ancestor version A. This conflict is automatically resolved by using the value from the principal version: merging D into E via $\mathrm{merge}(D,E)$ will result in the key–value pair (*Food*, “Steak”) being stored in the merge result, while merging E into D (as illustrated in Figure 4) will maintain the key–value pair (*Food*, “Pizza”) in the result.

The implementation of $\mathrm{merge}(L,L')$ involves first finding a nearest common ancestor K of L and L', and then following a recursive merge process on the three tries representing these three map versions. A nearest common ancestor K of L and L' is a common ancestor that minimizes the sum of the shortest path lengths from K to L and from K to L'.[4] The process starts by making a duplicate of the root node at version L'—this is the entry point to the trie version after the merge is completed. The system then applies the following rules for each of the 32 possible bit positions p in the bitmap of the HAMT nodes for the three versions.

1. (BASE) If p is set only in the bitmap of the node of L, then the subtrie corresponding to the position is added to the newly created trie version and p is set.
2. (BASE) If p is not set in the bitmap of the node of L, but is set in the bitmaps of the corresponding nodes of both K and L', and these both reference the same child node for p, then the subtrie corresponding to that child node's subtree is removed from the newly created trie version, and p is unset.
3. When p is set for two or more of K, L, and L', there is the potential for a conflict. A conflict arises when the child corresponding to p in the different trie versions differ. If the child references are the same, no conflict arises, and the second base case resolves any issues for position p. Otherwise:
   (a) (RECURSE) If all the non-null child references refer to HAMT nodes, a recursive merge call is performed on the subtries corresponding to p in each trie version to construct a new subtrie for use in the new trie version, and to set (or unset if the new subtrie is null) p.[5]
   (b) (BASE) Various intricate cases must be dealt with carefully to resolve conflicts when one or more of the non-null child references refers to a key–value pair. Depending on the case, this may involve adding (or removing) the key-value pair to (or from) a corresponding subtrie, merging key–value pairs into a new HAMT node, or overwriting the value associated with the key in the new trie version.

4.2.1 Differencing

When dealing with multiple versions of a data structure, it is also natural to ask what the difference between two versions is. Therefore, DSaaS provides a query operation to obtain the difference between versions of a data structure in a human-readable format: As with classical text file differencing utilities, the query output shows which key–value pairs must be added to or removed from one version to obtain the other.

The implementation of the difference operation between two versions follows a procedure similar to

[4]In general L and L' may have multiple nearest common ancestors, and which one is chosen may subtly influence the merge result in some cases.

[5]Note that in this case, one of the three subtries may be null.

the merge operation, except that the operations required to transform one version into the other are recorded and combined into an output string.

4.3 Transpositions

The original HAMT will give different version identifiers and use space for multiple versions with the exact same contents if they were created in different ways. We therefore adapted Zobrist hashing (Zobrist, 1970) for identifying transpositions. Zobrist hashing is a probabilistic hashing technique which is designed to incrementally build hashes and is commonly used in game tree search (Marsland, 1986). This is done by generating a random bitstring for each new key-value pair added to the trie. This random bitstring is then bitwise XOR'ed with the current version identifier to produce the new version identifier. If a key–value pair is subsequently removed, its bitstring is once again bitwise XOR'ed with the current version identifier to produce the new version identifier. Since the bitwise XOR is its own inverse, adding a key–value pair and removing it directly afterwards from an initial version leads to the same version identifier as the initial version. This simple example illustrates the general principle: the version identifier will depend on the content of the version, and not on the path followed to obtain it. There is a risk of hash collisions with this scheme where two tries with different contents are assigned the same version identifier. However, this risk can be made vanishingly small by using sufficiently long random bit-strings (albeit at the cost of increased computation requirements)—our implementation currently uses bit-strings of length 64.

4.4 Version History

In order to allow access to previous versions of the trie, access pointers to previous versions are stored in nodes of a separate directed graph structure. The graph itself represents the relationships between the various versions, where the nodes represent versions, and edges represent updates to the data structure. For the map, these updates could be insertion, replacement, or removal of a key–value pair, or a merge operation. This graph structure is explicitly stored in LevelDB, where every key is a version identifier for a node and the value is a list of the version identifiers for the node's edge destinations. DSaaS can thus provide an access map of the various versions of the data structure to the developer by presenting the version history graph in a navigable format. By having each user operate in their own namespace provided by the system, each data structure of each user can be given its own individual version history.

For classical confluent persistence, the graph representing the relationship between various versions is a directed acyclic graph; however, due to our detection of transpositions, the possibility of cycles in the graph arises. This is an interesting technical challenge for future work; however our current approach is to discard any edge whose addition to the version history would create a cycle.

4.5 Versioned Graph

DSaaS provides a versioned graph which employs the versioned trie to provide a single symbol table for representing both the nodes and edges of the graph. In the case of the versioned graph, typical operations include adding or removing a node or an edge, or merging two graph versions. These operations generally require multiple map operations to complete. While the intermediate versions of the symbol table during such an operation are perfectly valid maps, they are not generally consistent with a corresponding graph version. Thus, to achieve a suitable granularity for versioning new core data types, certain lower-level operations should effectively be grouped into transactions.

The directed graph provides a proof of concept of this approach, grouping the map operations used when performing a graph operation into a single transaction, which then generates a single new directed graph version corresponding to the single graph API call. Merging and differencing of versioned graphs is almost identical to the processes for versioned tries, due to the use of the versioned trie as a symbol table for storing the graph nodes and edges.

As a further example, in the context of a binary search tree (BST), deleting a value from the BST should be equivalent to creating one new version. However, if the BST were implemented using a directed graph representation, then the deletion requires a number of changes to the underlying graph structure, which without such transactioning would result in multiple intermediate graph versions being created which do not represent valid BSTs.

5 EVALUATION

To benchmark the core of our prototype, we ran some simple tests comparing execution time of the DSaaS versioned trie and the Dat core library (Ogden, 2015a). We also tested the storage usage of our core operations. Finally, we evaluated the Java language binding by comparing access to a local machine and

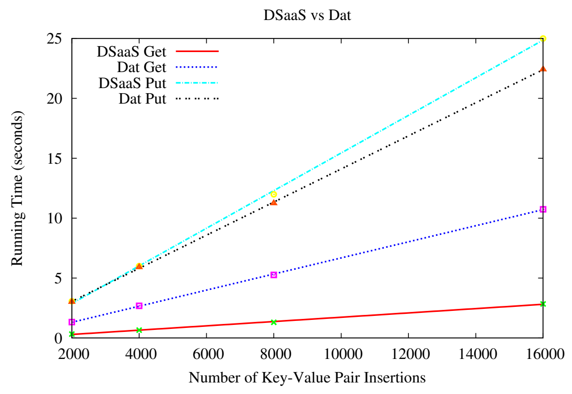

Figure 5: Running times for inserting batches of key-value pairs using the same key with different values and retrieving the values from different versions at the end.

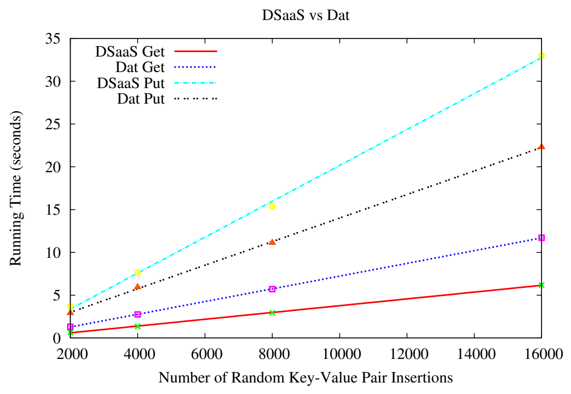

Figure 6: Running times for inserting different batches of random key-value pairs, and then retrieving the values from different versions.

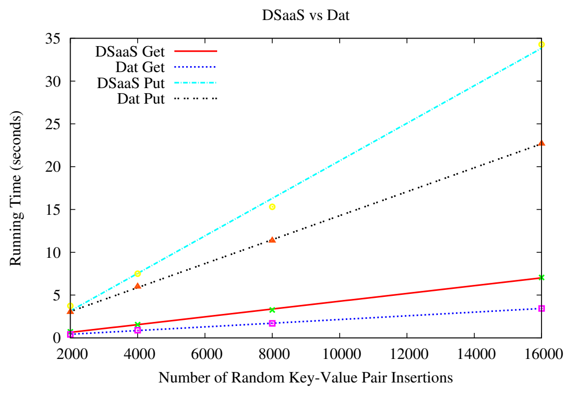

Figure 7: Running times for inserting batches of different random keys and values, and retrieving batches of values from the final version.

to a remote server in terms of latency and throughput. The experiments were run on an Acer Aspire 5750 containing a Core i7 processor, 8GB RAM and 500 GB Seagate Momentus 5400 rpm hard drive.

5.1 Execution Time

The following tests were run on the local machine. The first test used the core libraries to insert 2 000, 4 000, 8 000, and 16 000 unique random strings in turn as values for a fixed key into a versioned map provided by the two systems. Each insertion replaced the existing value, and resulted in a new version; afterwards, the value associated with the key was retrieved for each version.

Figure 5 plots the running times in seconds against the number of insertions: Insertion times on Dat are somewhat better than on DSaaS for a larger number of iterations, whereas retrieval times are faster on the DSaaS platform than on Dat. This is likely due to DSaaS doing a single operation to retrieve a key–value pair from a specified version, as opposed to Dat having to perform two operations: First, Dat retrieves the version (this operation is called a *checkout*) and then it queries the key–value pair for the retrieved version.

The next test was similar, except randomly generated keys and values were used instead of a single fixed key. Retrieval of a key was performed at the version it was inserted. Figure 6 plots the results: DSaaS is slightly slower than in the previous experiment, but is still faster than Dat during retrieval from different versions, presumably for the same reason as above.

The final test differed from the second test in that each key–value pair was retrieved from the final version after all the key–value pairs had been added. From Figure 7, it is evident that the retrieval and insertion times are very similar, and that Dat outperforms DSaaS when it does not have the extra penalty of checking out a new version before each retrieval.

Therefore, DSaaS is better for manipulating and accessing various versions—there is no penalty associated with retrieving multiple versions. This is important for DSaaS, as multiple users may be manipulating or accessing the same data structure simultaneously and requiring each user to first checkout a version would add an unnecessary time penalty.

5.2 Storage Usage

5.2.1 Insertion

The storage usage of the versioned trie was evaluated by sequentially inserting batches of sizes 500, 1 000, 2 000, 4 000, 8 000 and 16 000 from a data set of 165 995 payment records made by the County of Denver (County of Denver, 2015), in each case result-

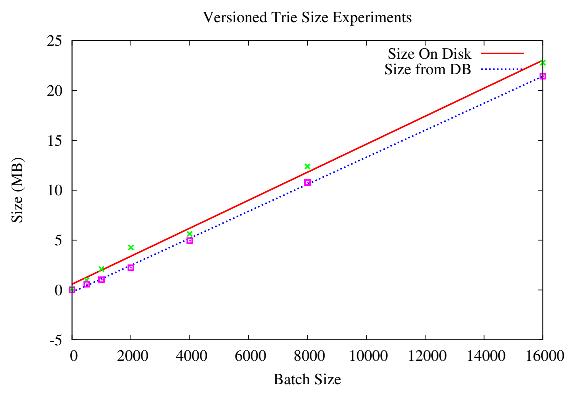

Figure 8: Storage usage when inserting batches of real-life information into the versioned trie.

ing in a linear version history of length corresponding to the number of the elements inserted. For the experiment, the RowID was used as key, and the payee, funding source, city, and amount were serialized into a JSON string for use as the corresponding value.

A small program was built using the Java library binding for sending the experimental data to a local server. Before inserting each sequence, the directory containing the database for all the data structures and version data was cleared. Thereafter, the data structure was created through the web interface and the Java program was executed. For each batch size, the size of the database was found by using LevelDB's `approximateSize()`, and the size of the `db_data` directory was determined by the Linux command-line utility program `du`.

Figure 8 plots the data and a least squares linear fit for each storage metric. As expected, we obtain almost exactly linear growth ($R^2 = 0.99$), with the slope indicating that every key–value pair inserted increases the disk storage used by approximately 1.3 KiB. Since the average size of a key–value pair in this data set is 107 bytes, the system can thus store the full history of the structure with roughly a 12-fold increase in storage over only storing an ephemeral version in this case.

5.2.2 Removal

The effect of removal on storage usage was tested by first inserting elements and then, starting at the last version, removing all the elements again. This was done in batches of sizes: 500, 1 000, 2 000, 4 000, 8 000, and 16 000 from the data set.

Data storage size increases linearly when removing data; again we give a reminder that we are preserving the entire history as it changes. In this case, the least squares linear fit ($R^2 = 0.99$) indicates that a single removal adds approximately 1 KiB of data. Therefore, bearing in mind that the key–value pairs used have an average size of 107 bytes, remembering a removal in the version history costs the storage equivalent of 10 data items in the ephemeral structure.

Table 1: The latency (in ms) for the *put* operation using the library binding to connect to a remote server and the localhost, and using JavaScript to test it on the core system.

Approach	**Latency (ms)**
Remote Server (Library Binding)	206
Localhost (Library Binding)	8
Core (JavaScript)	2

5.2.3 Merging

Next we attempted to evaluate the impact of the merge operation on the storage required by the system. This involved adding k distinct elements from the initial version, where k proceeds over the sequence 500, 1 000, 2 000, 4 000, 8 000, and 16 000. This was done in two batches—both batches have unique elements—creating two final version identifiers. The merge operation was then applied to the two distinct final version identifiers, with storage measured before and after the application of the operation, and the change in data structure size was recorded for each k.

A linear relationship ($R^2 = 0.99$) between the batch size and the increase in storage was found. The increase in storage was, however, smaller than reinserting the items added during the merge: Merging 16000 with 16000 unique elements results in an increase in size of approximately 650 KB, equivalent to approximately 6000 items.

5.3 Latency and Throughput

The Java library binding was used to send requests to a remote server as well as the localhost in an effort to measure the latency and throughput of each request. In addition, the same operations were executed using JavaScript to interface directly with the core versioned trie. The results are summarized in Table 1. We observe a decrease in performance of roughly an order of magnitude by employing the API service, and another order of magnitude by routing the requests over the network. Addressing these performance degradations will be crucial to making this system more attractive for practical use.

6 CONCLUSION

This prototype is a step in the direction of a new service-oriented architecture for enabling collaboration on data structures. We developed a modified implementation of the HAMT data structure to fit our needs for a confluently persistent map, and used the map to implement a confluently persistent directed graph. We then exposed these data structures, with additional operations such as merging, differencing and forking, through an API of a web service than can be accessed by any web-enabled front-end or through a library binding.

This prototype can be improved in many ways; the main focus should be on increasing the system's speed and making it more storage-efficient. One option is experimenting with different implementations for the core versioned map data structures to identify the fastest and most efficient implementation. For example, instead of an HAMT, a fully persistent B-tree (Brodal et al., 2012) or Stratified B-Tree (Twigg et al., 2011) could be considered. Another important task is reducing the latency of the system—this can be tackled by looking into various ways of optimizing the library binding, such as using sockets, better local caching and sending batch requests.

On the front-end, the system can be improved by adding more data structures; adding better security by using HTTP(s) and encryption; and improving front-end performance by maintaining the front-end data in a data structure similar to the back-end.

ACKNOWLEDGEMENTS

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF. The financial and spatial assistance of the Naspers Media Lab is hereby acknowledged.

REFERENCES

Bagwell, P. (2001). Ideal hash trees. Technical report. `http://infoscience.epfl.ch/record/64398/files/idealhashtrees.pdf` [Accessed: 2015-11-07].

Bhattacherjee, S., Chavan, A., Huang, S., Deshpande, A., and Parameswaran, A. (2015). Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *41st International Conference on Very Large Data Bases*, volume 8, pages 1346–1357, Kohala Coast, Hawaii.

Bitbucket (2015). Git and mercurial code management for your team. `https://bitbucket.org/` [Accessed: 2015-08-29].

Brodal, G. S., Sioutas, S., Tsakalidis, K., and Tsichlas, K. (2012). Fully persistent B-trees. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 602–614.

Clojure (2015). Clojure Source Code of the PersistentHashMap.java. `https://github.com/clojure/clojure/blob/master/src/jvm/clojure/lang/PersistentHashMap.java` [Accessed: 2015-11-09].

County of Denver (2015). Denver Open Data Catalog: Checkbook. `http://data.denvergov.org/dataset/city-and-county-of-denver-checkbook` [Accessed: 2015-10-01].

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. (1986). Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121. ACM.

Facebook (2015). Flux. `https://facebook.github.io/flux/docs/overview.html` [Accessed: 2015-05-19].

Fette, I. and Melnikov, A. (2011). The WebSocket Protocol. RFC 6455 (The WebSocket Protocol). `http://tools.ietf.org/html/rfc6455` [Accessed: 2015-08-01].

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

GitHub (2015). Github – where software is built. `https://github.com/` [Accessed: 2015-08-29].

Google (2015). Google drive – cloud storage and file backup for photos, docs and more. `https://www.google.com/drive/` [Accessed: 2015-08-29].

Google (2015). LevelDB. `http://leveldb.org/` [Accessed: 2015-05-20].

Kantor, I. (2015). Objects JavaScript Tutorial. `http://javascript.info/tutorial/objects` [Accessed: 2015-08-30].

Kaplan, H. (1995). Persistent data structures. In Mehta, D. and Sahni, S., editors, *Handbook of Data Structures and Applications*. CRC Press.

le Roux, P. B. (2015). DSaaS: A Cloud Service for Persistent Data Structures. Honours project report, Stellenbosch University Computer Science Division. `http://cs.sun.ac.za/~kroon/dsaas/docs/dsaas_report.pdf` [Accessed: 2016-02-15].

Marsland, T. A. (1986). A review of game-tree pruning. *ICCA Journal*, 9(1):3–19.

Membrey, P., Plugge, E., and Hawkins, D. (2010). *The Definitive Guide to MongoDB: the NoSQL Database for Cloud and Desktop Computing*. Apress.

Ogden, M. (2015a). maxogden/dat-core. `https://github.com/maxogden/dat-core` [Accessed: 2015-08-13].

Ogden, M. (2015b). Versioned Data, Collaborated. `http://dat-data.com/` [Accessed: 2015-08-13].

Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29.

Pollock, R. (2015). Git and GitHub for data. `http://blog.okfn.org/2013/07/02/git-and-github-for-data/` [Accessed: 2015-08-30].

Python (2015). Built-in Types – Python 2.7.10 documentation. `https://docs.python.org/2/library/stdtypes.html#mapping-types-dict` [Accessed: 2015-08-30].

Santos, P. (2013). Three-Way Merge. `http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902` [Accessed: 2016-02-11].

Straka, M. (2013). *Functional Data Structures and Algorithms*. PhD thesis, Computer Science Institute of Charles University, Prague.

Twigg, A., Byde, A., Milos, G., Moreton, T., Wilkes, J., and Wilkie, T. (2011). Stratified B-trees and Versioned Dictionaries. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems, HotStorage*, volume 11, pages 10–10.

Watt, D. A. and Brown, D. (2001). *Java Collections: an Introduction to Abstract Data Types, Data Structures and Algorithms*. John Wiley & Sons, Inc.

Zobrist, A. L. (1970). A new hashing method with application for game playing. *ICCA Journal*, 13(2):69–73.

APPENDIX

This appendix discusses a number of additional aspects of the DSaaS system potentially of interest to the reader, but not central to the main paper.

Access Control

Since DSaaS is a web application providing a service to multiple users, it is imperative to ensure users are only granted access to data at a level corresponding to the permissions allocated to them.

Registration and authentication for the system is currently handled using Google's single sign-on authentication service, with each user selecting a unique *namespace* during registration. Each data structure is allocated to its creator's namespace, and the data structure is also given a unique identifier, as specified by the creator, within the namespace. This makes it possible to identify a data structure unambiguously by its namespace and data structure identifier.

Various access levels—none, read, read/write, and administration—specify the actions any given user can perform on a particular data structure. The creator of a data structure automatically has administrator access, whereas other users have none by default. An administrator of a data structure may modify the access level of other users—the common usage of granting new users read or stronger access is called sharing—and also has access to modify selected properties of the data structure.

To facilitate wider sharing beyond the fine-grained mechanism above, two special users are defined. The special user "registered" represents all registered users of the system. Since the system can also be used without registration, the special user "public", which represents all users of the system—including casual, unregistered visitors—is also defined.

Data Freedom

A forking feature is provided to enable duplication of data structures. Forking creates a virtual copy of a data structure. The virtual copy mechanism used saves space by duplicating the version history of the data structure, without duplicating the underlying persistent data structure. Since each version stored by the versioned trie is immutable, the developer of the forked version can carry on applying updates and the original trie will not be influenced.

Pull requests and changes—which allow the changes made to a forked data structure to be incorporated into the original— could be added as future improvements.

Besides forking, a user may wish to extract a data structure from the system for offline storage or use. Alternatively, a user may wish to import a previously exported data structure, or even import data from other sources. To facilitate this, DSaaS provides an export operation which allows users to download a JSON-formatted version of a data structure and its history. Such a file can also be imported into the system; however, importing data in other formats is not yet supported. Note that exporting and re-importing a data structure is essentially the same as completely duplicating (not forking) the data structure.

Reproducibility

The prototype DSaaS system is open source, and all code is available at `https://bitbucket.org/dsaas/`. The code repositories include code for the client application, the server, and the language bindings, as well as testing code and code used for performing the experiments in this paper—note that in all cases the repository version used for experiments is tagged as "closer-article".