Date of acceptance Grade

Instructor

Persistent Data Structures for Incremental Join Indices

Antti Karjalainen

Helsinki June 8, 2020

Master's thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Antti Karjalainen

Työn nimi — Arbetets titel — Title

Persistent Data Structures for Incremental Join Indices

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	June 8, 2020	51 pages + 6 appendix pages

Tiivistelmä — Referat — Abstract

Join indices are used in relational databases to make join operations faster. Join indices essentially materialise the results of join operations and so accrue maintenance cost, which makes them more suitable for use cases where modifications are rare and joins are performed frequently. To make the maintenance cost lower incrementally updating existing indices is to be preferred.

The usage of persistent data structures for the join indices were explored. Motivation for this research was the ability of persistent data structures to construct multiple partially different versions of the same data structure memory efficiently. This is useful, because there can exist different versions of join indices simultaneously due to usage of multi-version concurrency control (MVCC) in a database. The techniques used in Relaxed Radix Balanced Trees (RRB-Trees) persistent data structure were found promising, but none of the popular implementations were found directly suitable for the use case.

This exploration was done from the context of a particular proprietary embedded in-memory columnar multidimensional database called FastormDB developed by RELEX Solutions. This focused the research into Java Virtual Machine (JVM) based data structures as the implementation of FastormDB is in Java. Multiple persistent data-structures made for the thesis and ones from Scala, Clojure and Paguro were evaluated with Java Microbenchmark Harness (JMH) and Java Object Layout (JOL) based benchmarks and their results analysed via visualisations.

ACM Computing Classification System (CCS):
Information systems → Data management systems → Data structures
Theory of computation → Design and analysis of algorithms → Streaming, sublinear and near linear time algorithms

Avainsanat — Nyckelord — Keywords
join indices, persistent data structure, in-memory database, OLAP, data warehouse, RDBMS

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Efficient database management systems (DBMS), which are used to access, store and modify information in a database, are a vital part of today's businesses for handling the large amount of data generated by customers and internal systems alike. There exists two main ways of using a database: By doing a huge amount of write-biased queries that touch only a small part of the data, called *on-line transaction processing* (OLTP); and by doing a relatively small amount of analytical queries, which are read focused, that touch a large portion of the stored data called *on-line analytical processing* (OLAP). Both of these methods have important business value. The queries caused by single customer actions are predominately of the OLTP kind, because their actions affect only the data related to their identity or the identity of groups they are in. Queries that calculate aggregates for analytic purposes over the data generated by the customers are of an OLAP nature, because they do not make modifications and just read a subset of the data for large portions of customers [1].

Historically, relational database management systems (RDBMS) have been disk-based. However, due to the slowness of disk versus memory and the recent growth in size and reduction in price of memory, new RDBMSs are mostly in-memory based [2]. This change has also highlighted the benefits of columnar memory layout versus traditional row based memory layout especially but not limited to OLAP usage [1].

To ensure the efficiency of a RDBMS there are architecture-defining design decisions and smaller scale decisions, like what algorithms and data structures are used, to make. This thesis first goes through the larger-scale design decisions, such as different aspects of transaction management and data layout of a particular database, simultaneously giving context to them from the wider database ecosystem. After this, it focuses on the specific sub-problem of join indices and how constructing them could be improved from the current state of the specific database. Given this specific problem, multiple different data structures both existing and new ones specifically created for the thesis are evaluated empirically using benchmarks, to explore how persistent data structures fit into the picture from both a time-efficiency and memory-usage perspective. Conclusions and reflections are then offered.

1.1 Problem statement

This thesis tries to help in answering the question of how to do incremental updating of join indices for a proprietary RDBMS developed by RELEX Solutions, henceforth referred to as FastormDB, an embedded in-memory columnar multidimensional database the development of which began in 2005. FastormDB is the basis of RELEX – the supply chain management software of RELEX Solutions. It is used for storing data received from customers and for doing in-memory analytical queries on it, producing a variety of reports used to improve the efficiency of supply chain management among other business critical uses [3].

In FastormDB there are two kinds of tables: clustered and non-clustered. The data in *clustered tables* is ordered by one of the columns making it fast to index and to do range queries based on that column. For primary keys internal identifiers are used in FastormDB. *Internal identifiers* are based on where in the internal data structure the data is stored, which is in contrast to *external identifiers* which would be generated and would need to be stored explicitly. Because the order of rows inside a clustered table can change when rows are inserted middle of it, they do not have stable internal identifiers. This means that there cannot be foreign key references that point to clustered tables. However, there can be foreign key references from clustered to non-clustered tables and between non-clustered tables. Clustered tables are used for the biggest tables in FastormDB and are, in production use, in the order of 10 billion rows large. Non-clustered tables, which are the majority, are in the order of at most a billion rows. To handle efficient foreign key to primary key joining between tables, join indices are constructed.

In the current implementation these join indices are completely reconstructed on access when the relevant data has changed. The reconstruction is slow, but this scheme works well enough in the common use case where the customer sends their data once or a few times per night for batch processing. However, as the world moves to the direction of more and more online shopping, there is no longer a pronounced daily cycle and so the frequency of transactions grows. This requires rethinking the construction of join indices, to be more incremental, in order to make the process faster.

The aim of the thesis is to understand join indices, analyse existing persistent data structures and their implementation details, and to evaluate the suitability of existing persistent data structures for join indices from performance point of view.

2 Design decisions

This section will explain important design decisions made in FastormDB, how they relate to the wider RDBMS ecosystem, and what implications they have for join indices.

2.1 Transaction management

One of the key design decisions for RDBMSs is how transaction management is done. *Transaction management* is the way in which a database handles concurrent insertions, modifications and removals to the contents of a database correctly and efficiently. When designing the transaction management part of a database there are several key design decisions [4]:

1. concurrency control protocol
2. version storage

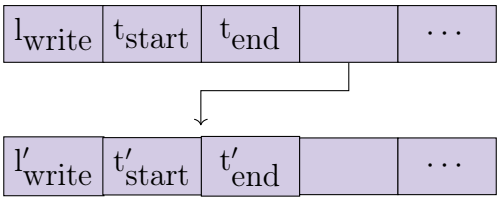


Figure 1: The structure of version chain of tuples in MVCC.

3. garbage collection
4. index management

In the following we will discuss all of these four aspects, as well as other tightly related concepts.

2.1.1 Multi-version concurrency control

FastormDB implements multi-version concurrency control (MVCC), which is the most widely used concurrency control for modern databases [4]. Its selection has wide reaching implications on the rest of the design. The basic idea of MVCC is to allow multiple physical versions of a single logical row (tuple) to exist simultaneously in the database. When an update is performed, the database creates new versions for the affected tuples. When a read is done, the database will fetch the newest version of the tuple that the transaction can see in logical time. This allows read-only queries to happen concurrently with mixed read-write queries, improving throughput. Having old versions of tuples available also provides opportunities for implementing *time travel*, in which a historical state of the database can be inspected and operated on. In MVCC each transaction has an unique timestamp which gives rise to a before/after ordering between the transaction start times. For each version of a tuple there is, at a minimum, the following metadata: A write latch, a start and end timestamp and a pointer to the previous or next version of the tuple depending on the variant of MVCC in use. Figure 1 demonstrates this metadata.

The data of each version of a tuple is valid only for the time between its start and end timestamps. When a transaction wants to modify a tuple it takes that tuple's newest version's write latch, after which it creates a new version that contains the modified data. The start timestamp of that new version is set to the timestamp of the transaction that created it and the end timestamp is set to infinity, marking the new version valid only after the current transaction. The pointer field inside that version is also updated, to make it part of the version chain. The previous versions, whose latch the transaction is still holding, end timestamp is updated to the transaction's timestamp marking it valid until the new version. Finally, when it is time to commit, the transaction gives up the write latch, making its modifications visible to all other transactions.

In most variations of MVCC there is no need for a read latch, though the write

latches have to be respected to avoid reading partially updated data, as the reads always happen to a version that is valid for the current transaction and writes do not modify existing versions data, but instead create new ones. There exists multiple different variants of MVCC that handle conflicting writes and reads differently and have different performance characteristics such as MVTO, MVOCC, MV2PL and serialization certifiers [4].

2.1.2 Isolation

Isolation tells how concurrent transactions are allowed to perceive each other. Isolation is determined by so-called *isolation levels*, each of which guarantee absence of a particular set of phenomena [7]. These phenomenon have been formalized differently in different papers, but in general they are things like *dirty read* in which a transaction reads another running transaction's modifications which is then aborted leading to a transaction relying on non-existent information [7][6]. The strongest isolation level is so-called *strict serializability* [8], which ensures the real-time ordering of non-overlapping transactions and exhibits no phenomena. A system based on long read and write locks supports such a guarantee, but multi-version schemes may not without extra effort [9]. By default MVCC provides *snapshot isolation*, which means that when a transaction starts, it can see a consistent snapshot of the database as it is at that moment: all changes made by previous transactions are seen and no changes of other active transactions will be seen. For it, there is *first-committer-wins* policy, which means that if two active transactions modify the same tuple, then the first of them to commit is successful and the other is rolled back and has to try again. See Figure 2 for relationships between different isolation levels. The database system can also provide different isolation guarantees for running and committed transactions, single tuples and predicates, and the database system can support running different transactions with different isolation levels, giving lots of choice for the database designer [5].

2.1.3 Version storage

The storage of tuples in the version chain can be implemented in different ways. One option is to store all of them in the same physical storage. The problem with this approach is that the physical storage can become fragmented, where old infrequently accessed versions and new versions end up next to each other, slowing database access. Another option is to have a separate physical storage for the newest version and another for all of the old versions. The problem with this is approach is that the performance of accessing an older version is degraded, which can negatively affect long running transactions. It is also possible to, instead of multiple versions of tuples, create deltas based on the modifications and store those separately. This makes modifications that edit only parts of the tuple fast, but degrades read performance [4].

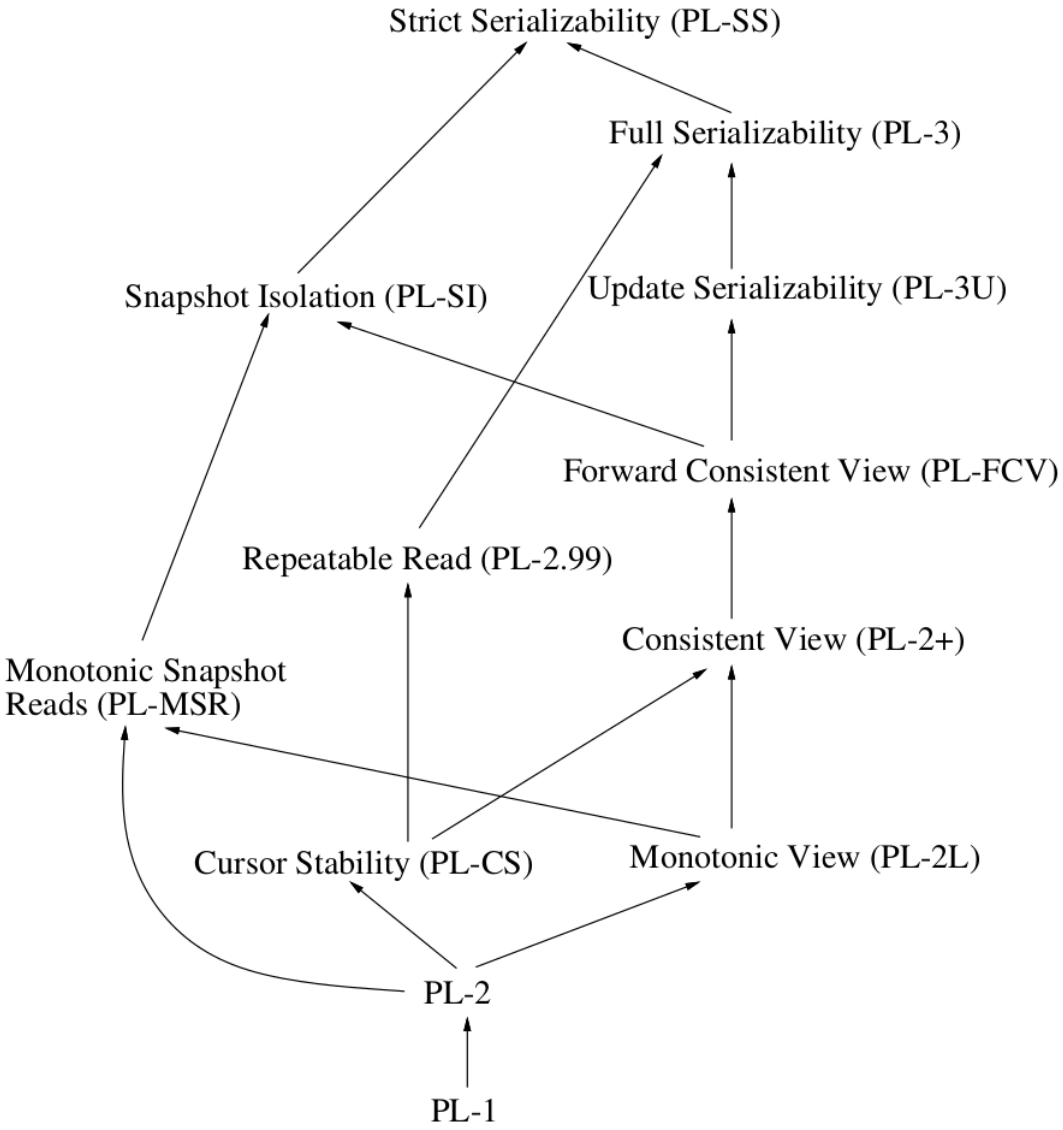


Figure 2: Relative strengths of isolation levels, where arrow points to the stronger one. Codes in parenthesis relates to the formalization in [5], the source of the figure. See [6] for a shorter proceedings version of the formalization.

2.1.4 Garbage collection

If stale versions in the version chain are left on the physical storage, the performance of database operations start to degrade. The strength of degradation effect depends on the chosen version storage, but in general these stale versions have to be garbage collected. To detect the stale versions, their timestamps can be compared with the lowest timestamp of the active transactions to figure out if those versions can still be seen by any of them. With long running transactions this is not efficient as there might be versions in the middle that cannot actually be seen by any transactions, but which are not noticed by this scheme. To fix this, interval-based detection can be used, where interval intersection problem can be solved to find those versions that are not visible to active transactions. If it is also known *a priori* which tables are accessed by transactions, then this information can be used to get even more precise information on the stale versions. Garbage collection can also happen at different granularities: versions can be collected separately or grouped based on the transaction that created them [10].

2.1.5 Time travel

The idea of time travel is to provide a way to inspect and work on a historical version of the database. This is useful for determining what has happened to the database in the past, generating reports on the usage of the database, recovering from mistakes and other operations that need previous versions of data. Supporting time travel increases memory or disk usage as the previous data has to be saved somewhere. Time travel can also degrade the efficiency of all queries and time-travelling queries are usually slower themselves as they are not the usual use case to optimize for. In MVCC, time travel can be implemented by not doing garbage collection and accessing the versions of tuples that are valid for the wanted timestamp. To avoid performance degradation old versions can be persisted to disk.

2.1.6 Index management

An *index* is a data structure used to exchange memory usage for access speed to speed up different operations in a database. For a single table there are two kinds of indices: primary and secondary indices. These can be thought as index in the back of a book, which can be used to quickly locate information inside the book. There can be only one primary index which determines the logical ordering of data inside a table[1]. The actual physical order might be different depending on implementation, but it should still optimize the access of the primary index. There can be multiple secondary indices as they are completely separate data structures to a table. However, they need extra effort to keep them synced up to the state of their

[1]These are also called clustered indices in the literature and secondary indices are called non-clustered ones. [11]

table, slowing modification operations. These primary indices are used for clustered tables in FastormDB.

In MVCC these indices tell about the existence of a version of a tuple, but not which of the versions, if any, the transaction can actually see and so the transaction has to go through the version chain itself to figure this out. This means that indices do not exhibit false negatives, where transaction would miss a data in table, but there can be false positives, where the transaction does extra work to figure out that the version chain does not actually contain anything it can see. [4]

There can also exist so-called join indices. A *join index* is an index between two tables that essentially caches the result of a join operation. This makes future join operations significantly faster in exchange of using more memory and needing maintenance. The maintenance overhead makes them more suited to OLAP situations where modifications are relatively rare [11]. It is also possible construct join indices between two tables that are connected indirectly via a third table [12].

2.2 Columnarity

Whereas in traditional row-oriented databases the physical storage stores rows sequentially after each other, in a columnar database the data of each column is laid out separately in long sequences. This difference between row-oriented and columnar data layout is similar to the difference between an array of structures (AoS) and a structure of arrays (SoA). An *array of structures* is familiar from object oriented programming (OOP): to represent many structures we create an array of objects. In a *structure of arrays* we instead have only single object that contains arrays for each field of the object that we want to represent. For these arrays it holds that the value in nth index corresponds to the value of the nth object's field that the array is representing [13]. Code examples for these can found in Figure 3 and their memory layouts in Figure 4.

SoA may initially seem unintuitive, however it can provide performance benefits in different ways:

1. If one is interested in data contained in only a few of the fields, we do not need to use the CPU cache for the data of other fields, improving our cache utilisation [13].

2. SoA allows us to play representational tricks such as using a (compressed) bit vector for the `alive` field in the example of Figure 3, run-length encoding or using integer codes for the `age` field [14].

3. If the data is stored inline it is possible to perform SIMD operations on it [15].

These benefits can be attained in OLAP workloads: usually analytical queries are interested only in some of the columns. This means that we get the promised cache utilisation boost. Analytical queries are also interested on big portions of values on

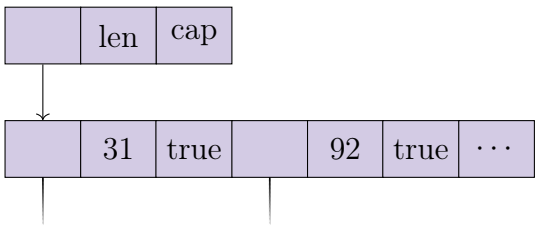
```rust
struct Person {
    name: String,
    age: u32,
    alive: bool,
}

struct AoS {
    people: Vec<Person>,
}
```

(a) An example of AoS

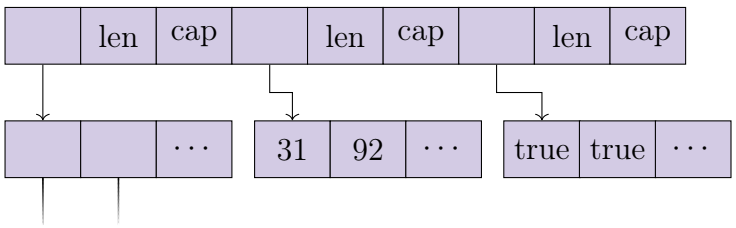
```rust
struct SoA {
    name: Vec<String>,
    age: Vec<u32>,
    alive: Vec<bool>,
}
```

(b) An example of SoA

Figure 3: An example of encoding a list of people as both an array of structures (AoS) and a structure of arrays (SoA)



(a) Memory layout of AoS



(b) Memory layout of SoA

Figure 4: Memory layouts of examples of Figure 3. `len` and `cap` are the length and capacity fields of the vector, respectively.

those columns, negating possible negative effects of compression on random access. Furthermore, operations might be able to be used on the compressed data making them even more efficient [1]. See [16] for analysis on compression techniques used in FastormDB.

2.3 Cube

Furthermore, for certain types of OLAP workloads it is beneficial to organize data into multidimensional tables where two of the dimensions are standard ones: attributes and rows. One good use for the third dimension is as time axis, as data is often temporal. In data warehouse environments the tables are often arranged to so-called *star schema*, where there exists one fact table in the center that is connected to dimension tables around it [17]. This schema allows building cubes efficiently: the cells of a cube contain measures derived from the fact table and its dimensions are derived from the dimension tables.

Multidimensional databases can be formalized in two different ways: relationally and cubically. The relational formalization is the more classical approach, where relational algebra is extended to support multidimensional tables. The cubical view is a more modern approach, where tables are thought of as hypercubes with a set of operations to split and carve cubes into new ones, operations to join cubes together, and for calculating aggregates over cubes [18]. FastormDB takes the cubical approach and defines a set of operations to manipulate its cubes.

3 Implementation details

In this section we will discuss a few implementation details of FastormDB that affect the implementation of join indices. First we explain how FastormDB implements MVCC with columnar layout using fragments, then how the cube operations are implemented using live masks and finally explain how join indices are implemented as so-called mappings.

3.1 Fragments

In MVCC, when a tuple is modified a new version of it is created. However, with columnar layout, tuples are not stored contiguously and so copying tuples cannot be directly done. One could create similar version chain for each column, but as columns are large copying them for each modification would be inefficient. FastormDB addresses this problem by splitting columns into so-called *fragments* that are stored separately. Now, to address a single row we split its identifier into two parts: one which is used to choose the right fragment and a second one to get the value from right row of that fragment. Now each fragment can be thought of similarly as a tuple from MVCCs point of view.

3.2 Live mask

Within a query there exists a live mask for each table in the database. A *live mask* is a set of rows that are active for a given transaction – the rows that the query is interested in. Each live mask starts with all of the rows active. As a query proceeds, different operators can be used to carve pieces from the cube – to kill one or more live rows from the live mask. When joining two tables together, the resulting live mask has to be calculated. When the desired subset of rows are active, aggregation operations can be performed on the active rows in the live mask.

id	name
1	quivira
2	paititi
3	lanka
4	la canela

(a) `location`

id	product-id	location-id
1	1	1
2	1	2
3	2	1
4	2	3
5	3	2
6	3	4

(b) `product_location`

id	name	group-id
1	banana	1
2	raspberry	2
3	bilberry	2

(c) `product`

Figure 5: Derivation of a live mask for a query that selects from the `location` table and joins it with the `product` table via the `product_location` table. Rows highlighted with a color are active.

For example, if we select the rows from the table `location` in Figure 5 where the `location.name` equals either "quivira" or "lanka", the live mask for it is $\{1, 3\}$. Now if we join with the table `product_location` by `product.id = product_location.product-id`, the live mask for the join will be $\{1, 3, 4\}$. Further joining with the `product` table by `product_location.location-id = location.id` will produce a live mask $\{1, 2\}$. To carry out this joining efficiently we will introduce the concept of *mappings*, that works as our join indices, in the next section.

3.3 Mappings

Let us focus on a database that adheres to the schema specified in the Figure 6. Given such a database, let us assume that we have somehow obtained a `product_location.id`. Now it is quite easy to find which `product.id` is associated with it: we can just look at the column `product_location.product-id` at the position that corresponds to the row of our id. The other direction is also easily obtained: to find which `product_location.id`s are associated with `product.id` we can perform a linear scan through the `product_location.product-id` column.

There exists a clear performance asymmetry between these directions. To reduce this difference, we introduce the concept of mappings. We associate each column containing references to another table with two different mappings, one for each direction: a many-to-one mapping for the `product_location.id` $\rightarrow$ `product.id`

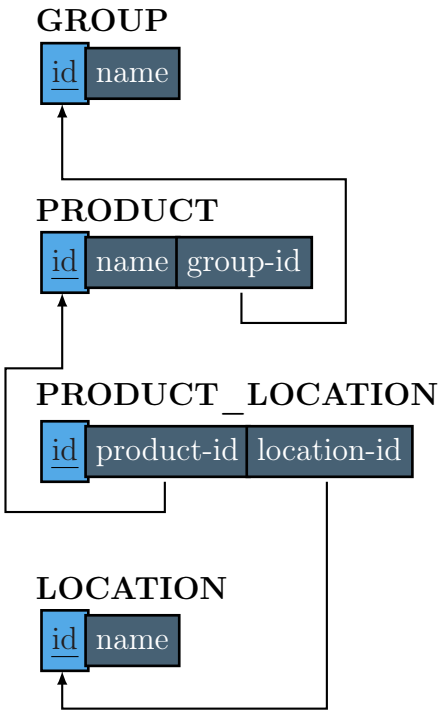


Figure 6: Simple example of database schema

direction and a one-to-many mapping for the `product.id` → `product_location.id` direction. Concretely, a *mapping* is a data structure, a map, that describes how the foreign keys stored in a reference column are connected with the primary keys of another table.

We now know how to connect two tables directly with a mapping. But what if we want to know which `group.id` is associated with our `product_location.id`? There exists no direct mappings between the `product_location` and `group` tables. However, there exists the following two direct mappings: The many-to-one mapping f : `product_location.id` → `product.id` and the many-to-one mapping g : `product.id` → `group.id`. From these two mappings we can now construct the composite mapping $g \circ f$: `product_location.id` → `group.id`. Because both of the constituents of the composition are many-to-one mappings, the resulting composite mapping is also many-to-one as each of the mappings produce only a single identifier for given an input. Concretely, this creates new a map data structure that merges the map data structures of the mappings f and g.

To see that we can compose with one-to-many mappings there are two cases to consider: pre- and post-composition. If we pre-compose with a one-to-many mapping we need to apply the other mapping to all ids in the set produced by it. This results either in a set of ids or in a set of sets of ids. The set of sets can be then flattened into set of ids with a operation called *join* [19]. Post-composing with a one-to-many mapping to a many-to-one mapping is simple: The one-to-many mapping can be just called with the result of the many-to-one mapping given as a parameter. Because a one-to-many mapping produces a set of ids, composing with it also produces

a set of ids and so the composite mapping is also a one-to-many mapping.

For example, if we know a specific `group.id` and we want to find out which of the `product_location.ids` correspond with it, we can first use the one-to-many mapping g^{-1} : `group.id` $\rightarrow$ `set<product.id>` to get a set of `product.ids`, where g^{-1} is the reverse[2] of the mapping g. After that we can use the one-to-many mapping f^{-1} : `product.id` $\rightarrow$ `set<product_location.id>` on every identifier in that set to get a set of sets of `product_location.ids`, which we can then flatten, producing the final composite mapping $f^{-1} \circ g^{-1}$: `group.id` $\rightarrow$ `set<product_location.id>`.

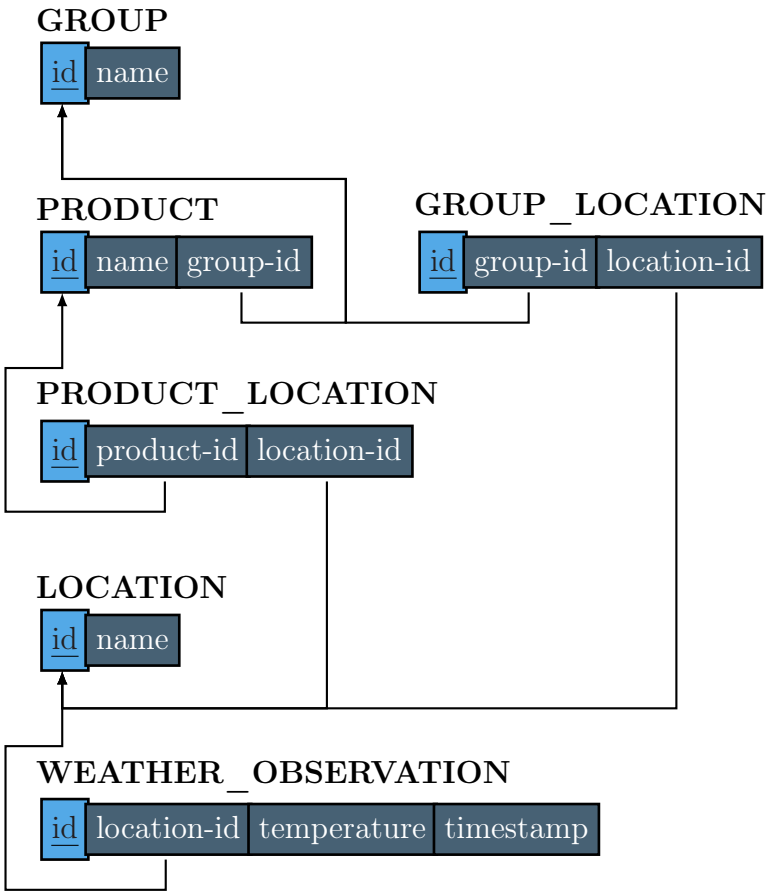


Figure 7: Example of schema with multiple paths

Mappings can be also thought of as the materialization of paths between tables for a certain state of the database in question. A *path* is a way of getting from one table to another via reference columns in the database schema. In a schema there can exist two or more such paths between two tables. To resolve a mapping between two tables with multiple paths between them, first all mappings for available paths are constructed. After doing this the mappings can be combined into single one by either unioning or intersecting the sets that the ids are mapped into, where the results of many-to-one mappings are thought of as singleton sets.

[2]Note that $g^{-1} \circ g \neq id \neq g \circ g^{-1}$ where $id(x) = x$ and so $\bullet^{-1}$ is not an inverse operation.

For example, in the schema described in the Figure 7 there exists two paths from the `group` table to the `weather_observation` table: One that goes via the `product_location` table and another one that goes via the `group_location` table. To calculate a mapping f : `group.id` $\rightarrow$ `set<weather_observation.id>` for a particular state of a database adhering to the schema, we first have to calculate the mappings $f_{product\_location}, f_{group\_location}$: `group.id` $\rightarrow$ `set<weather_observation.id>`. Given these two mappings, we can then define that for all $x \in$ `group.id` either $f(x) = f_{product\_location}(x) \cup f_{group\_location}(x)$ or $f(x) = f_{product\_location}(x) \cap f_{group\_location}(x)$. Notice how the mappings that correspond to each path share the same postfix mapping `location.id` $\rightarrow$ `weather_observation.id`. Besides postfixes, paths can share other portions and the mappings corresponding to those parts can be calculated only once and then memoized.

4 Data structures

In the current implementation of the FastormDB these mappings are calculated from scratch when a transaction modifies a relevant table. This is tolerable if only one or a few transactions are committed in a night. As more transactions are being committed in a day, the calculation of mappings needs to be made faster. One way to do this would be to make the calculation incremental and based on the previous mapping, minimizing the amount of work required to be done.

Currently, the data structure used for mappings is an array of n elements, indexed by an integer key i. The data structure grows dynamically by either doubling in size, if the key is between n and $2n$, to avoid performance degradation or by growing up to i if $i > 2n$. This provides maximal random access performance, for values that exist, assuming no deletes are being done. When the size of the database tables grow, because of the efficient compression used for them, mappings start to account for a relatively large percentage of the memory usage. As the frequency of transactions grows, there might be multiple transactions active simultaneously and so there could exist multiple mappings simultaneously too. Therefore, minimizing the memory usage effect of multiple mappings existing simultaneously should also be pursued.

4.1 Persistent data structures

Persistence in persistent data structures does not mean persisting the data structure to disk or other long-lived storage medium. Instead it means that the data structure is immutable in nature: one cannot directly modify it. Any operations that would traditionally do modifications, such as insertions and deletions, instead create a brand new data structure with the wanted operation applied. This means that, if we have a pointer to such a data structure, we know that nobody is going to modify its structure without our knowledge. This further means that one can easily access the data structure in parallel without causing data races because nobody is

modifying it simultaneously.

There are different levels of persistence, giving rise to partially, fully and confluently persistent data structures. Partial persistence allows read operations to past versions of the data structure, but write operations only for the newest version. Fully persistent data structures allow read and write operations to all past and present versions of the data structure. Confluent persistence additionally allows *melding* or merging different versions together to form a new version of the data structure [20]. Data structures that are not persistent are called *ephemeral*. In this thesis we will concentrate on fully persistent map data structures, though many of them are in fact confluently persistent as they support join operations.

A naive way to make a data structure fully persistent is to copy it each time a modification would be made and then apply the changes to that copy instead. However, just creating copy of the data structure to create persistence is wasteful. Because we know that nobody is going to modify the data structure under us, we can reuse parts of it in the new one, thus potentially making modification operations cheap from both a memory and time perspective. This reuse is called *structural sharing*.

4.2 Prior art

Many modern persistent data structures are based on the ideas of Array Mapped Tries (AMT) [21]. The basic idea in them is of a trie in which the key is partitioned and each part is used to walk down a tree structure, with the actual data is stored in leaves. For Hash Array Mapped Tries (HAMT) [22], the key is hashed and then parts of the hash are used for the descent. HAMT can be used to implement either set or map data structures, which only differ on the data stored in leaves: only the keys or a key value pairs, respectively. The Compressed Hash-Array Mapped Prefix-tree (CHAMP) [23] is a JVM-specific optimized version of HAMT in which data is also stored in the trunk of the trie. The Relaxed Radix Balanced Tree (RRB-Tree) [24] is a persistent vector implemented using AMT that can avoid the overhead of HAMT by assuming that all indices before last index have values. This thesis concentrates on JVM implementations of these data structures. We now describe them in more detail.

4.2.1 Hash Array Mapped Trie

The explanation of HAMT in this section is as a set data structure for ease of exposition. Expanding the idea to a map data structure is just a matter of storing both keys and values in the leaves.

Let h be a hash function that maps keys $k_1, \ldots, k_{10}$ to 8 bit hashes.

$$
\begin{aligned}
h(k_1) &:= 10\,00\,00\,00_2 & h(k_6) &:= 00\,01\,11\,00_2 \\
h(k_2) &:= 10\,01\,00\,00_2 & h(k_7) &:= 00\,10\,00\,10_2 \\
h(k_3) &:= 00\,11\,00\,00_2 & h(k_8) &:= 11\,00\,01\,10_2 \\
h(k_4) &:= 01\,00\,01\,00_2 & h(k_9) &:= 00\,00\,10\,10_2 \\
h(k_5) &:= 00\,11\,01\,00_2 & h(k_{10}) &:= 00\,00\,11\,10_2
\end{aligned}
$$

Now HAMT with branching factor 4 looks like:

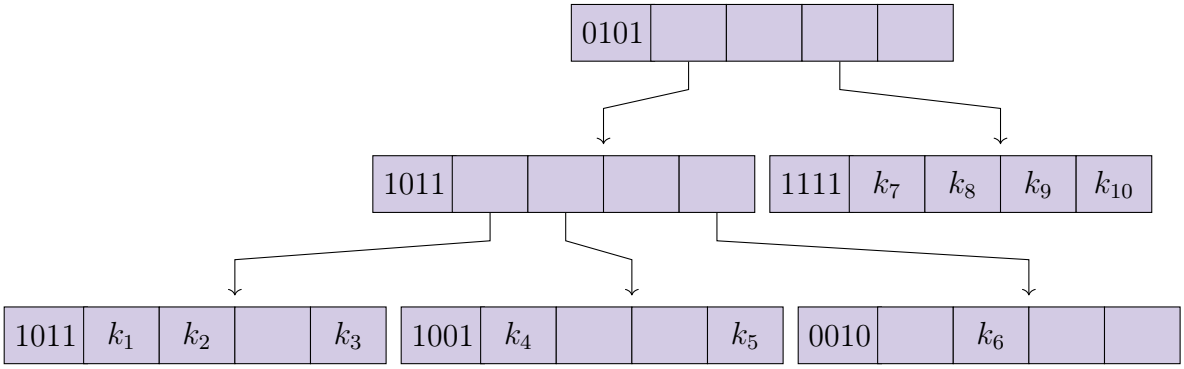


Figure 8: An example of HAMT data structure.

As can be seen from Figure 8, each time the postfixes of the hash code match, a new node is created for them, which disambiguates them. To find a key from the data structure we walk down from the root by taking the two least significant bits of the hash of the key and using them to select which of the four children to descend to until eventually a leaf node is reached. For example, if we want to find k_5 we first look at the $00_2 = 0$ index of the root and from there we find a pointer to a new trunk node. We now look, based on the next two bits, at the $01_2 = 1$ index of the array of that trunk node. We have now encountered a pointer to a leaf node and if the key were looking for exists, it should reside in $11_2 = 3$ index of the array and in this case it indeed does. As we can see from the example, there has to be a way to distinguish whether the node is a trunk or a leaf node. This can be either done by adding an extra bit of metadata, by using sum types [25] which are also called tagged unions or polymorphism [26] if the used programming language supports such features. In practical implementations, the branching factor of the trie is kept as a power of two to allow doing fast modulus using bit mask operation, as the following equation holds:

$$a \bmod 2^m = a \;\&\; (2^m - 1)$$

for all $a, m \in \mathbb{N}$, where & is the bitwise and-operation.

In our example, on each node a bit mask of 4 bits is stored which indicates which of the children are present, one bit for each children where bits from the lowest to highest order corresponds to children from the first to last child respectively. Using

this mask there is no need to actually allocate four slots worth of memory for each node and the previous trie can actually be represented in memory as follows:

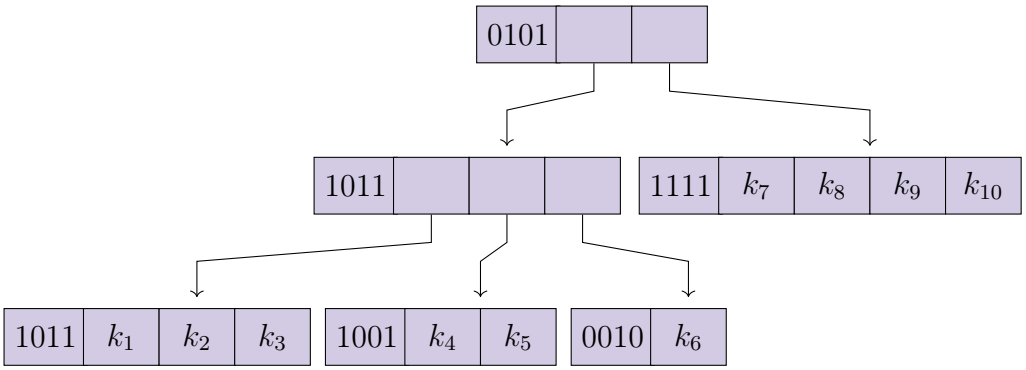


Figure 9: Memory representation of the example HAMT data structure.

The index of each child can then be efficiently recovered using the *population count* instruction `popcnt`, which calculates the number of 1 bits in an integer, by the following formula.

Definition 4.1. Given a bit mask $b \in [0, 2^n)$ where $n \in \mathbb{N}$ and an index $i \in [0, n]$, the corresponding index in an array packed using the bit set is equal to

$$\texttt{popcnt}(b \;\&\; \texttt{mask}) - 1$$

where $\texttt{mask} \coloneqq 2^i - 1$.

Based on the bit mask of the examples root node we know that there are values for the first and the third index. To determine where in the compacted array these two are, we can do the following for the first index

$$\texttt{mask} \coloneqq 2^1 - 1 = 1_2$$

$$\texttt{popcnt}(0101_2 \;\&\; 1_2) - 1 = \texttt{popcnt}(1_2) - 1 = 0$$

and the following for the third index

$$\texttt{mask} \coloneqq 2^3 - 1 = 111_2$$

$$\texttt{popcnt}(0101_2 \;\&\; 111_2) - 1 = \texttt{popcnt}(101_2) - 1 = 1.$$

When doing modifications to HAMT, only the nodes on the path from the root to the target of the modification need to be modified to update pointers in them. This means that when used as a persistent data structure only that path needs to be copied. This is illustrated in Figure 10. This means that operations on the trie are $O(\log_b n)$ from both a memory and time perspective, where b is the branching factor, assuming that the hashing distributes entries evenly and so ensures that the

trie is balanced. The most commonly used branching factor is 32, which is chosen as a good trade off between cache-line usage and overhead of copying [27].

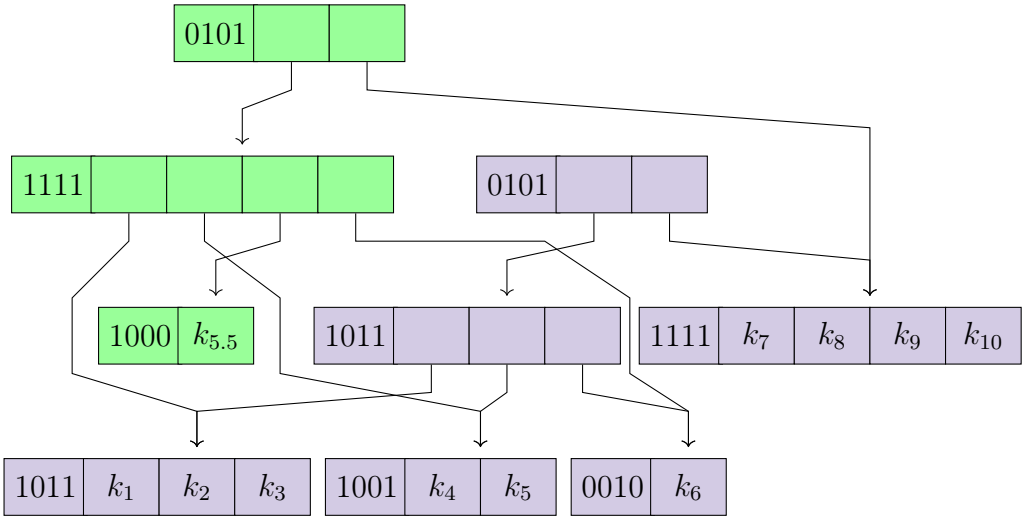


Figure 10: An example of adding $h(k_{5.5}) = 01\,11\,10\,00_2$ to the previous example.

Because the hash code is of fixed length, there exists a maximum depth for the HAMT data structure. In Java and other JVM languages, a hash code is traditionally a 32-bit integer which means that the maximum depth with a branching factor of 32 is $\lceil 32/\log_2 32 \rceil = 7$. Because a hash code has a fixed bit length it is possible to encounter full hash collisions, where two keys hash to exactly the same hash code. In [22] it is claimed that one needs on average $\log_2 n$ bits to distinguish n uniformly distributed hash codes. However from the Birthday Principle [28] we know that for the minimal number of keys n such that the probability of collision is at least 50% it holds that $n \in \{\lceil\sqrt{2 \cdot 2^b \cdot \ln 2}\rceil, \lceil\sqrt{2 \cdot 2^b \cdot \ln 2}\rceil + 1\}$. From this fact we can derive the following bounds:

$$
\begin{aligned}
n &\leq \lceil\sqrt{2 \cdot 2^b \cdot \ln 2}\rceil + 1 \\
n &\leq \sqrt{2^{b+1} \cdot \ln 2} + 2 \\
(n-2)^2 &\leq 2^{b+1} \cdot \ln 2 \\
2^{b+1} &\geq \frac{(n-2)^2}{\ln 2} \\
b &\geq \log_2(\frac{(n-2)^2}{\ln 2}) - 1 \\
b &\geq 2\log_2(n-2) - \log_2(\ln 2) - 1 \\
b &\geq 2\log_2(n-2) - 1
\end{aligned}
$$

$$
\begin{aligned}
\lceil \sqrt{2 \cdot 2^b \cdot \ln 2} \rceil &\leq n \\
\sqrt{2^{b+1} \cdot \ln 2} &\leq n \\
2^{b+1} &\leq \frac{n^2}{\ln 2} \\
b &\leq \log_2 \frac{n^2}{\ln 2} - 1 \\
b &\leq 2 \log_2 n - \log_2(\ln 2) - 1 \\
b &\leq 2 \log_2 n
\end{aligned}
$$

assuming $n > 2$.

Combining the previous bounds we get that

$$2 \log_2(n-2) - 1 \leq b \leq 2 \log_2 n,$$

where b is the length of hash codes in bits and n is the number of hash codes. These bounds do imply you need $O(\log_2 n)$ bits on average for distinguishing n hash keys, but the conclusion drawn from this that the collisions are rare in practice is incorrect. For example, in the case of the largest benchmarks performed in this thesis where $n = 2^{23}$, one needs

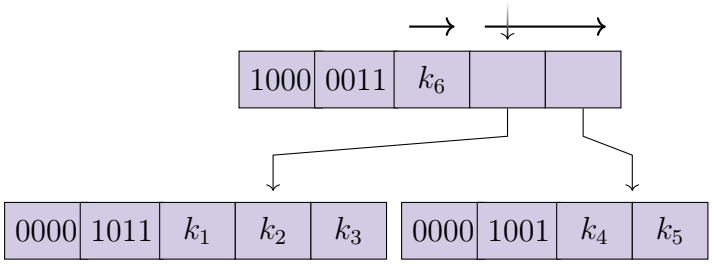
$$
\begin{aligned}
2 \log_2(2^{23} - 2) - 1 \leq b &\leq 2 \log_2(2^{23}) \\
45.999999312 \lessapprox b &\leq 46 \\
b &= 46
\end{aligned}
$$

bits on average for distinguishing all of the keys which is a twice as much compared to $\log_2 2^{23} = 23$ derived from more naive analysis. Furthermore, the bounds tell us that we can expect there to be already at least some collisions when $n \gtrsim 2^{16}$ for a 32-bit hash code. So handling collisions is not only of theoretical interest. One way to handle them is to rehash the key to get more hash bits and use those for continued descending. The more common collision handling strategy is to have a special type of a node that stores a linked list or an array that contains all the collided entries.

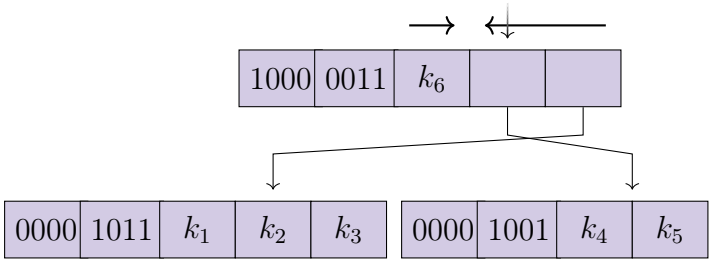
4.2.2 Compressed Hash-Array Mapped Prefix-tree

The Compressed Hash-Array Mapped Prefix-tree (CHAMP) is a JVM optimized version of HAMT [23]. Its main idea is to store keys and values not only in the leaves of the trie, but also in the trunk, reducing memory usage and improving iteration speed. This idea was already present in [22], but CHAMP improves upon it and optimizes it for the JVM. The main difference in CHAMP is to improve memory usage by using two bit maps: one for children and another for the internal key and value storage. If we laid out the internal storage and the children consecutively from front to back, then to be able to index the children we would need to first calculate how many bits are set in the first mask and then use that and the second bit mask to index into it. To sidestep this, CHAMP uses the fact that arrays in Java record

their lengths to index children from back to front by subtracting the number of set bits from the length. These layouts are demonstrated in Figure 11.



(a) Naive layout



(b) Optimized layout

Figure 11: Alternative CHAMP layouts with branching factor 4.

4.2.3 Relaxed Radix Balanced Tree

A Relaxed Radix Balanced Tree (RRB-Tree) uses the idea of AMT in a different way to create a persistent vector data structure [24][27]. Construction of an RRB-Tree starts from a Radix Balanced Tree (RB-Tree) with branching factor 2^b. An RB-Tree is a complete 2^b-ary tree whose non-leaf root has at least two children and whose leaves are the same distance away from the root. For a m-ary tree T we can think of T as an array of its children which can be indexed with T_i, for all $0 \leq i < |T| \leq m$.

Definition 4.2. Full m-ary tree.
Let T be a m-ary tree.

$$\texttt{full}(T) \coloneqq \top \qquad \text{if } |T| = 0, \text{ and}$$

$$\texttt{full}(T) \coloneqq (|T| = m) \wedge \bigwedge_{i=0}^{|T|-1} \texttt{full}(T_i) \qquad \text{if } |T| > 0.$$

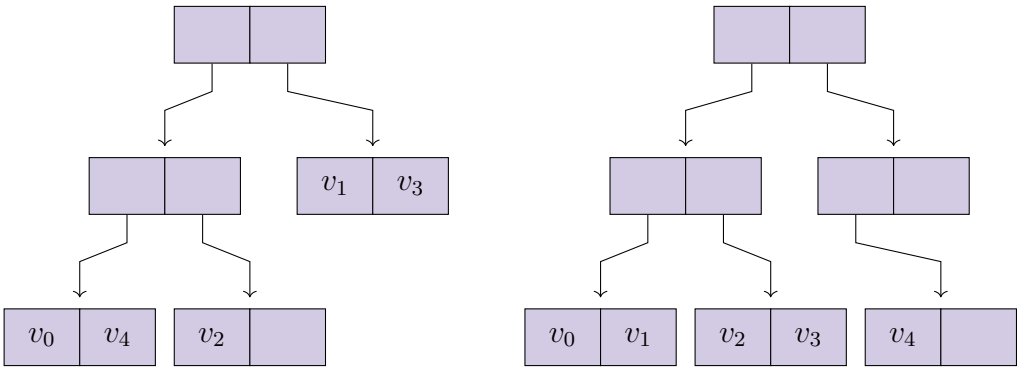
Definition 4.3. Complete m-ary tree.
Let T be a m-ary tree.

$$\texttt{complete}(T) \coloneqq \top \qquad \text{if } |T| = 0, \text{ and}$$

$$\texttt{complete}(T) \coloneqq \texttt{complete}(T_{|T|-1}) \wedge \bigwedge_{i=0}^{|T|-2} \texttt{full}(T_i) \qquad \text{if } |T| > 0.$$

What this means is that all of the nodes of a RB-Tree are filled up to the maximum size of 2^b except for the rightmost nodes starting from the root.

Figure 12 illustrates the difference between a HAMT-like radix tree and an RB-Tree. If one used RB-Tree-like indexing for HAMT, the trie would always need to have maximum depth, because one cannot know *a priori* the depth needed to disambiguate all n hashes.



(a) A HAMT like radix tree (b) A RB-Tree

Figure 12: Radix trees with branching factor 2 for the array $v_0v_1...v_4$ where the subscript is used as a key.

Because a full 2^b-ary tree with n leaves all hanging the same depth is perfect, its height is exactly $\log_{2^b} n$. Consider an RB-Tree T with n leaves. Let h' be the height of one of the perfect children of the root node of T. Now because the leaves of T are of the same depth, the height of T has to be $h' + 1$. The perfect subtree has always less than n children, because the root node has at least two children. In the other extreme when the tree is perfect itself and so has 2^b children, the perfect subtree has $\frac{n}{2^b}$ children. These give bounds for the height of the tree:

$$\log_{2^b} \frac{n}{2^b} + 1 \leq h < \log_{2^b} n + 1,$$
$$\log_{2^b} n \leq h < \log_{2^b} n + 1.$$

And so using the definition of ceiling we get that $h = \lceil \log_{2^b} n \rceil$. Now that we know the height of the tree we can use h of the b-bit sized parts of the key for descending the tree. When we descended to the ith level we use the $(h - i - 1)$th part to decide which children to continue descending to.

RRB-Tree relaxes the previously described RB-Tree to support efficient concatenation while preserving the time complexities of other operations. This relaxation adds a special node which can have less than 2^b children. When such node is encountered radix approach cannot directly be used to descend forwards. Instead a separate list of cumulative sizes of subtrees is maintained, which can be then binary searched

if large enough or linearly scanned if small enough to find the correct children to descend to. When two RRB-Trees are concatenated the nodes of the rightmost branches of the first tree will be merged to the nodes of the leftmost branches of the second tree. This merging is the only way these smaller nodes can be introduced into the data structure. So if no concatenations are made, there are no performance differences between RB and RRB-Trees.

4.3 Implementations

In the following sections we will describe the implementation details of data structures[3] that were implemented for this thesis. They are implementations of previously described data structures specialized to support 32-bit integers. Only insertion, access and iteration operations were implemented.

4.3.1 Int HAMT

Implementations of HAMT were first programmed in Kotlin and then ported to Java. Because the keys are 32-bit signed integers and the hash code is a 32-bit integer, hashing was skipped in a similar way to how some implementations of C++ `std::hash`[4] operate. This does mean that an attacker can easily find hash collisions, but in our use case this should not be a problem as the access to the system is restricted and the user cannot directly affect which keys are used. Because no hashing is done it also means that there will not be any hash collisions and handling of them can be ignored in the implementations. To distinguish trunk and leaf nodes polymorphism via sub-classing was used in both languages. However, in Kotlin this was done using `sealed class`[5], which is Kotlin's version of union type, that restricts the number of sub-classes to an amount known at compile time. A leaf node contains, in addition to the bit mask, two arrays: one integer array for the keys and an object array of the same length for the values. This split was made to avoid having to box the integer keys.

4.3.2 Int CHAMP

Implementations of CHAMP were first programmed in Kotlin and then ported to Java. Similarly to Int HAMT, hashing of keys was skipped. Because every node can carry key-value payload only a single type of node was used. However, this means that the integer keys have to be boxed which means more memory is used for the keys.

[3] The implementations can be found at https://github.com/WaDelma/thrive. The repository contains also raw benchmark results and script used to generate graphs in the thesis.

[4] https://en.cppreference.com/w/cpp/utility/hash#Notes

[5] https://kotlinlang.org/docs/reference/sealed-classes.html

4.3.3 Int Implicit Key HAMT

The idea of Int Implicit Key HAMT was to avoid storing the key inside the data structure and instead save it implicitly in the descend path. The problem then is that this causes there to always be maximum number of pointer chases making it both slow and more memory consuming.

4.3.4 Int RB-Tree

Implementation of the RB-Tree was programmed in Java. This implementation supports only non-negative integers and so essentially only supports 31-bit integers. Because we know the height of the tree, the data structure does not need different types of nodes and instead the leaves can be distinguished by how deep we have descended. The RRB-Tree was not implemented as it mainly affects the data structure if concatenations are done.

4.3.5 Int RB-Tree Redux

A variant of the Int RB-Tree was implemented using Java. The idea was for it to use a branching factor of 32 for trunk nodes and for leaf nodes to use branching factor 64. Because leaves can now hold twice as many values an RB-Tree Redux of height h can now hold at most $2 \cdot 32^h$ values. Thus, for the height h for an RB-Tree Redux with n elements the following holds:

$$
\begin{aligned}
2 \cdot 32^{h-1} \leq n &\leq 2 \cdot 32^h \\
h - 1 \leq \log_{32} \frac{n}{2} &\leq h \\
h \leq \log_{32} \frac{n}{2} + 1 \text{ and } \log_{32} \frac{n}{2} &\leq h \\
\log_{32} \frac{n}{2} \leq h &\leq \log_{32} \frac{n}{2} + 1 \\
\log_{32} n - 0.2 \leq h &\leq \log_{32} n + 0.8
\end{aligned}
$$

So the height of the tree is reduced by 0.2 meaning that 20% of the time the height is reduced by 1. To always reduce the height by 1 a leaf node would need to hold $32^2 = 1024$ values.

5 Evaluation

The evaluation of the different implementations of the data structures described above was performed via microbenchmarks using Java Microbenchmark Harness (JMH)[6]. Evaluation of memory usage was done using Java Object Layout (JOL)[7]

[6] https://openjdk.java.net/projects/code-tools/jmh/
[7] https://openjdk.java.net/projects/code-tools/jol/

library. The benchmark suite consists of 7 JMH and 4 JOL benchmarks, which are described in Table 1. The list of benchmarked data structures can be found from Table 2. In the suite, JMH is configured to run benchmarks with 3 JVM forks with exactly 4GB of RAM using the G1 garbage collector, to do 10 warmup iterations and 20 sample iterations 1 second each with forced garbage collection in between.

The benchmark results of this thesis were run on a Dell Inc. PowerEdge R920 server with 755 GB of DDR3 RAM with clock speed 1066 MT/s with transfer width of 64 bits and 4 of Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz, which each have 15 cores with 32KiB of L1 instruction and data cache, 256 KiB of L2 cache and the whole CPU has 37.5 MiB of L3 cache. The JVM used for benchmarks was OpenJDK 11.0.5+10 (1.8.0_192). The Clojure version used was 1.10.1. The Paguro version used was 3.5.6. For Scala both versions 2.11.12 and 2.13.1 were used.

First we will compare the performance differences of the two different Scala versions, shown in Figure 13. From it we can see that there is only little difference between the Scala versions for tree and int maps. For the RRB map there are substantial improvements in the newer version for the access and the `insert` benchmarks and a slight regression for the `iterateSequential` benchmark. As the implementation of hash map was changed between the two versions, performance differences between them are larger. From the results we can see that the `ScalaV2HashMap` – the successor of the `ScalaHashMap` – seems to have worse or equal performance in all but the iteration benchmarks, where it has considerably better performance after approximately thousand elements. Although it is hard to tell how the trend will continue with larger sizes than those included in the benchmark, it looks like the iteration performance advantage is lost after approximately 10 million elements.

From Figure 13 we can also get a general picture on the performance differences between different types of data structures. As can be seen from it, RRB-Tree based data structures are considerably faster than the others in all of the access and sequential benchmarks. Because RRB-Tree is meant to be used as a persistent vector, it is not surprising that in benchmarks where random entries are inserted or iterated over, the performance drops: Inserting a random element has to grow the vector to a large enough size by filling it with empty values and when filled sparsely empty gaps has to be skipped slowing the iteration down. We can also see that while tree maps have otherwise the worst get performance among the visualised data structures they have good iteration performance.

The int maps have interesting performance characteristics. For the access benchmarks their performance graphs are closest to the search tree based ones, but on average are at least 30% better[8]. The similarity extends to `insert` benchmark where int maps are on average approximately 88% better. In `insertSequential` benchmark int maps have different performance characteristic compared to search tree based ones and are on average approximately 213% better. In the iteration benchmarks the roles are reversed and int maps have on average approximately 32% worse performance than search tree based ones.

[8]See Appendix A.3 for comparison between them.

Benchmark	Parameters	Explanation
`insert`	`size`, `density`	A data structure is filled with `size` number of entries, with keys that are distinct and uniformly chosen from the interval $[0, \lceil \frac{\texttt{size}}{\texttt{density}} \rceil]$. The idea of the `density` parameter is to control how much space is between consecutive keys. This should not in theory affect hash based data structures.
`hittingAccess`	`size`, `density`	A data structure is prefilled as in the `insert` benchmark and then benchmarking the access of the data structure is performed with 1000 random keys that are present in it.
`missingAccess`	`size`, `density`	A data structure is prefilled as in the `insert` benchmark and then benchmarking access of the data structure is performed with 1000 random keys that are **missing** from it.
`iterate`	`size`, `density`	A data structure is prefilled as in the `insert` benchmark and then benchmarking iteration through the data structure is performed.
`insertSequential`	`size`	A data structure is filled with entries with keys $[0, \texttt{size})$. Again this should not affect benchmark results for hash based data structures.
`hittingAccessSequential`	`size`	A data structure is prefilled as in the `insertSequential` benchmark and then benchmarking access of the data structure is performed with 1000 random keys that are present in it.
`iterateSequential`	`size`	A data structure is prefilled as in the `insertSequential` benchmark and then benchmarking iteration through the data structure is performed.

(a) JMH based microbenchmarks. The parameter `size` takes values $\{2^n : n \in [0, 24]\}$ and the parameter `density` takes values $\{0.25, 0.5, 0.75\}$.

Benchmark	Parameters	Explanation
`sequential`	`size`	A data structure is filled with entries where keys are $[0, \texttt{size})$ and all values are the same
`sequentialCumulative`	`size`	The same as `sequential`, but after inserting an entry the resulting data structure itself is recorded into a list and at the end the size of that list is reported. This tests the effect of persistence on memory usage.
`random`	`size`	A data structure is filled with entries where keys are distinct and uniformly chosen from $[0, \lfloor \frac{size}{0.5} \rfloor]$ and all values are the same.
`randomCumulative`	`size`	The same as `random`, but after inserting an entry the resulting data structure itself is recorded into a list and at the end the size of that list is reported.

(b) JOL based memory benchmarks. The `size` parameter takes values $\{2^n : n \in [0, 23]\}$.

Table 1: The benchmark suite.

Data structure	Description
`ArrayMap`	Data structure following the current implementation used in FastormDB. An ephemeral data structure made persistent that is used as a baseline.
`JDKHashMap`	Wrapper for Java standard library's hash map that is used as a baseline.
`ClojureHashMap`	Implementation of HAMT that uses linear search and array for hash collisions, polymorphism for different types of nodes, has branch factor of 32 and, if a node has more than 16 elements, switches to uses plain array of size 32 instead of using a bitmask.
`ClojureTreeMap`	Implementation of Persistent Red-Black Tree [29]
`ClojureVectorMap`	RB-Tree like data structure inspired by [22]
`PaguroHashMap`	Derivative of `ClojureHashMap`
`PaguroTreeMap`	Derivative of `ClojureTreeMap`
`PaguroVectorMap`	Derivative of `ClojureVectorMap`
`PaguroRrbMap`	Implementation of RRB based on [24], but in it the relaxed nodes can have between $\frac{m}{3}$ and $\frac{2m}{3}$ nodes and which, when joining, just sticks shorter tree into larger one unaltered[2].
`ScalaHashMap`	Before version 2.13, Scalas hash map was an implementation of HAMT that used linked list for hash collisions and had a special node for the single entry case.
`ScalaV2HashMap`	After version 2.13 Scalas hash map is an implementation of CHAMP.
`ScalaTreeMap`	Implementation of Persistent Red-Black Tree
`ScalaIntMap`	Implementation of Fast Mergeable Integer Map which is a modified version of Patricia trie – variant of binary radix tree [30]. According to its documentation its largely superseded by `ScalaHashMap`[1].
`ScalaRrbMap`	Implementation of Persistent Red-Black Tree
`IntChamp32Kotlin`	Kotlin implementation (Section 4.3.2)
`IntChamp32Java`	Java implementation (Section 4.3.2)
`IntHamt32Kotlin`	Kotlin implementation (Section 4.3.1)
`IntHamt32Java`	Java implementation (Section 4.3.1)
`IntImplicitKeyHamtKotlin`	Kotlin implementation (Section 4.3.3)
`RadixTree`	Java implementation (Section 4.3.4)
`RadixTreeRedux`	Java implementation (Section 4.3.5)

[1] https://github.com/scala/scala/blob/v2.13.1/src/library/scala/collection/immutable/IntMap.scala#L169

[2] https://github.com/GlenKPeterson/Paguro/blob/3.1.2/src/main/java/org/organicdesign/fp/collections/RrbTree.java#L35

Table 2: Benchmarked data structures.

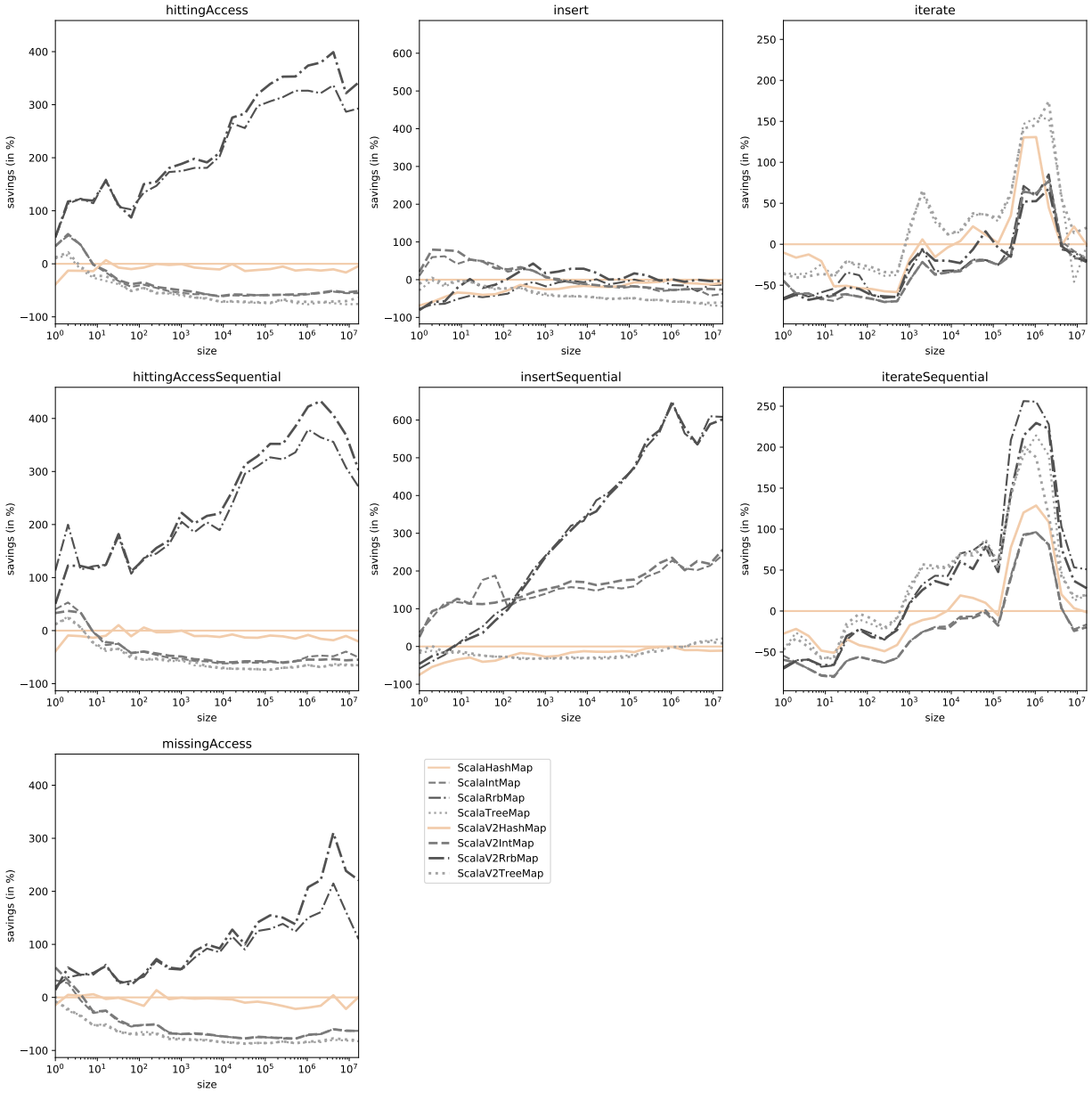


Figure 13: Visualisation of JMH benchmark results for persistent map implementations in Scala. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `ScalaHashMap`. On the horizontal axis is the size parameter in logarithmic scale.

To analyse cumulative benchmarks we first need to understand what they are actually measuring. For a persistent data structure based on a m-ary tree, adding a new leaf amounts to copying the path from the root to a leaf node, which takes $O(\log_m n)$ memory. Measuring memory usage of all n data structures created when the data

structure is constructed one element at a time, we get the following equation

$$\sum_{i=1}^{n} \log_m i = \log_m n! \subseteq n \log_m n - n \log_m e + O(\log_m n) \subseteq O(n \log_m n),$$

where the equality is derived from the fact that $\log_c a + \log_c b = \log_c ab$ and the first approximation is done using Stirling's approximation [31]. On top of this we also measure the memory usage of the headers of the data structure, increasing memory usage by $O(n)$, which does not asymptotically affect the overall memory usage. If we only look at the latest version of the data structure we notice that it also has $O(n \log_m n)$ memory usage. So the overhead of persistence can be at most $O(n \log_m n)$[9].

Figure 14 shows the JOL benchmark results for the data structures from Scala. From it we can see that there is no significant difference between different versions of tree and int maps, which corroborates with the results of JMH benchmarks to the fact nothing changed between the versions for them. For the RRB-Trees we can see that the memory usage of small sizes, under 32 elements, was improved. In the `sequentialCumulative` benchmark we can see that this improvement is at least 20% until approximately 600 elements and in the `randomCumulative` benchmark the improvement is at least 10% until approximately 150 elements. In the cumulative benchmarks the effect is extended to larger sizes, because now each size also contains every version of data structure before it, so the memory improvement accumulates. The improvement does not affect bigger sizes as strongly as the proportion of smaller structures shrinks: In the `sequentialCumulative` benchmark at size 16 the memory usage difference is 3007 bytes and at size 32 and beyond it stays at 3968 bytes. Interestingly, in the `randomCumulative` benchmark, `ScalaV2RrbMap` becomes worse after 200 elements crossing negative 5% threshold at thousand elements slowly creeping downwards. For the hash maps we can see that, while with lower numbers of elements memory usage of `ScalaHashMap` is better, `ScalaV2HashMap` has otherwise approximately 30% better memory usage. However, looking at the cumulative benchmarks we can see the roles are flipped and `ScalaHashMap` is superior. It therefore seems that `ScalaV2HashMap` cannot save as much memory using structural sharing as `ScalaHashMap`, even if it does use less memory as a data structure.

If we use Figure 14 to compare different data structures, we see that except when there are just a few elements in RRB-Tree, it uses noticeably less memory than any of the other data structures. Because we have visualised the `random` benchmark with density parameter 0.5, of possible indices approximately half have an actual value and the other half is empty. This causes RRB-Tree's memory usage advantage to halve when comparing the `sequential` and `random` benchmarks. In the `randomCumulative` benchmark the advantage of RRB-Trees is lost and they use the largest amount of memory among the visualised data structures. In the `sequentialCumulative` benchmark the memory advantage of RRB-Trees is present, though it is

[9]See Appendix A.5 for comparison of persistence overheads of benchmarked data structures.

in second place between approximately hundred and 0.8 million elements beaten by the int maps, after which RRB-Trees take the lead. Int maps have the best memory usage in `randomCumulative` benchmark too.

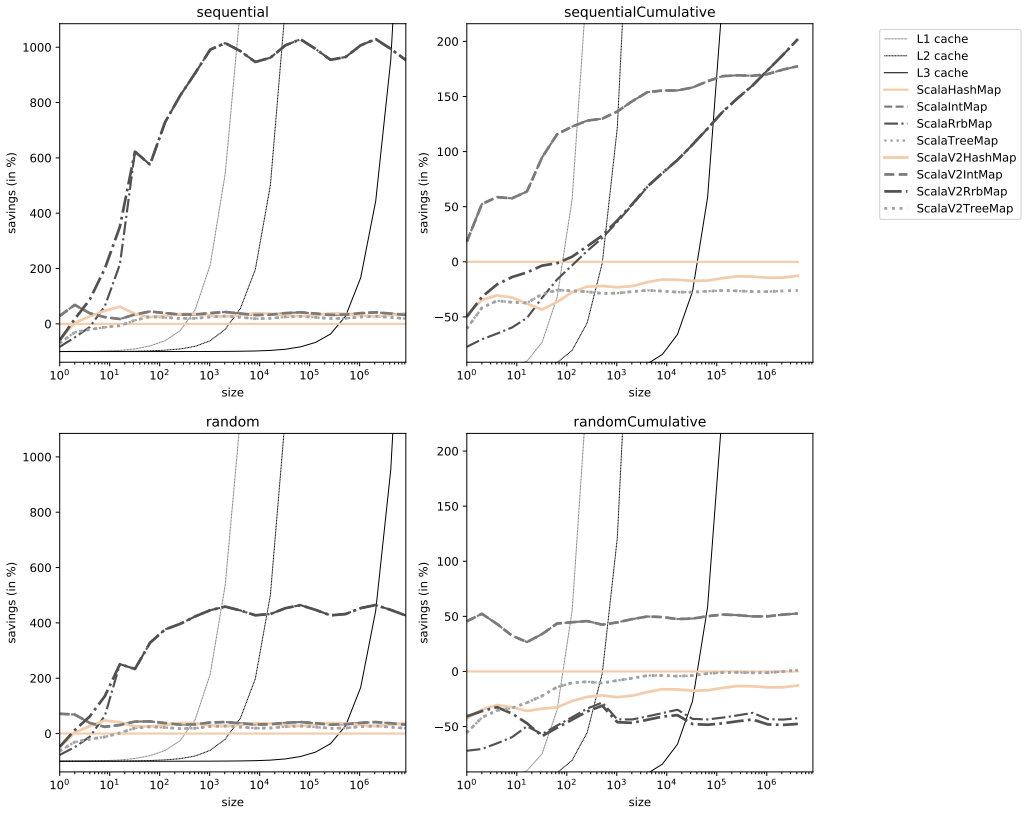


Figure 14: Visualisation of JOL benchmark results for persistent map implementations in Scala. On the vertical axis is the memory usage for each data structure normalized against `ScalaHashMap`. On the the horizontal axis is the size parameter in logarithmic scale. Normalized L1, L2 and L3 cache memory sizes are also visualised to give an idea of absolute memory usage.

In the following analysis we prefer newer versions of Scala's data structures, though we will include both versions of the hash map as their performances were significantly different.

In the next section we discuss those data structures that can be used generically: with any key and value type. We then go through data structures that are specialized to the special case of mappings, which most importantly means that the data structures need to only support integer keys.

5.1 Generic data structures

Of the persistent data structures mentioned above, the following are usable with any keys that are either hashable or comparable depending on if they are hash or search

tree based: `ClojureHashMap`, `ClojureTreeMap`, `PaguroHashMap`, `PaguroTreeMap`, `ScalaHashMap`, `ScalaV2HashMap` and `ScalaV2TreeMap`. Figure 15 shows the results of JMH benchmarks of these data structures normalized against `ScalaHashMap`.

The search tree based data structures perform worse than others in both insertion and access benchmarks, except when they contain only a few elements. This is because their tree structures have a smaller branching factor and so they are deeper and require more pointer chases to reach leaves. In the insertion benchmarks the `ScalaV2TreeMap` performs significantly better than other tree maps. This can be seen especially in the `insertSequential` benchmark where it is the second best generic data structure when it has more than approximately 2.5 million elements. In the iteration benchmarks, search tree based data structures do not perform comparatively as badly and in particular the `ScalaV2TreeMap` has the best iteration performance among all generic persistent data structures when it has more than approximately 600 elements. The superior iteration performance of `ScalaV2TreeMap` versus both `PaguroTreeMap` and `ClojureTreeMap` is most likely due to it preallocating a $2\log_2(n+2)-2$ sized stack – the maximum height for a red-black tree [32] – to hold the nodes on path from root to leaves. Additionally both `PaguroTreeMap` and `ClojureTreeMap` use `java.util.Stack` as their stack which is based on `java.util.Vector` which uses `synchronized` on its methods to make it thread safe. This might impose extra overhead if the JVM cannot elide the locks.

Because Paguro's implementations are derivative from Clojure's, one would expect them to perform similarly and this indeed is true for their search tree based data structures. However, in the case of hash based data structures, in all of the get benchmarks Clojure's hash map performs badly and has completely different performance characteristics compared to Paguro's. In insertion benchmarks their performance graphs do have a similar shape, but Clojure's data structure is still considerably worse. In iteration benchmarks this gap is smaller, but it nonetheless remains, especially in the case of the `iterateSequential` benchmark. This discrepancy needs further investigation.

In the get benchmarks `PaguroHashMap` intermittently has the best performance and in the `missingAccess` it has clearly the best performance among all of the compared generic data structures. In the `insertSequential` benchmark `PaguroHashMap` also has the best performance approximately after 150 elements. In iteration benchmarks the performance of `PaguroHashMap` is clearly not as good: it surpasses `ScalaHashMap` in the `iterateSequential` benchmark only approximately between 0.2-0.6 million elements and in the `iterate` benchmark only at approximately around 0.4 million elements.

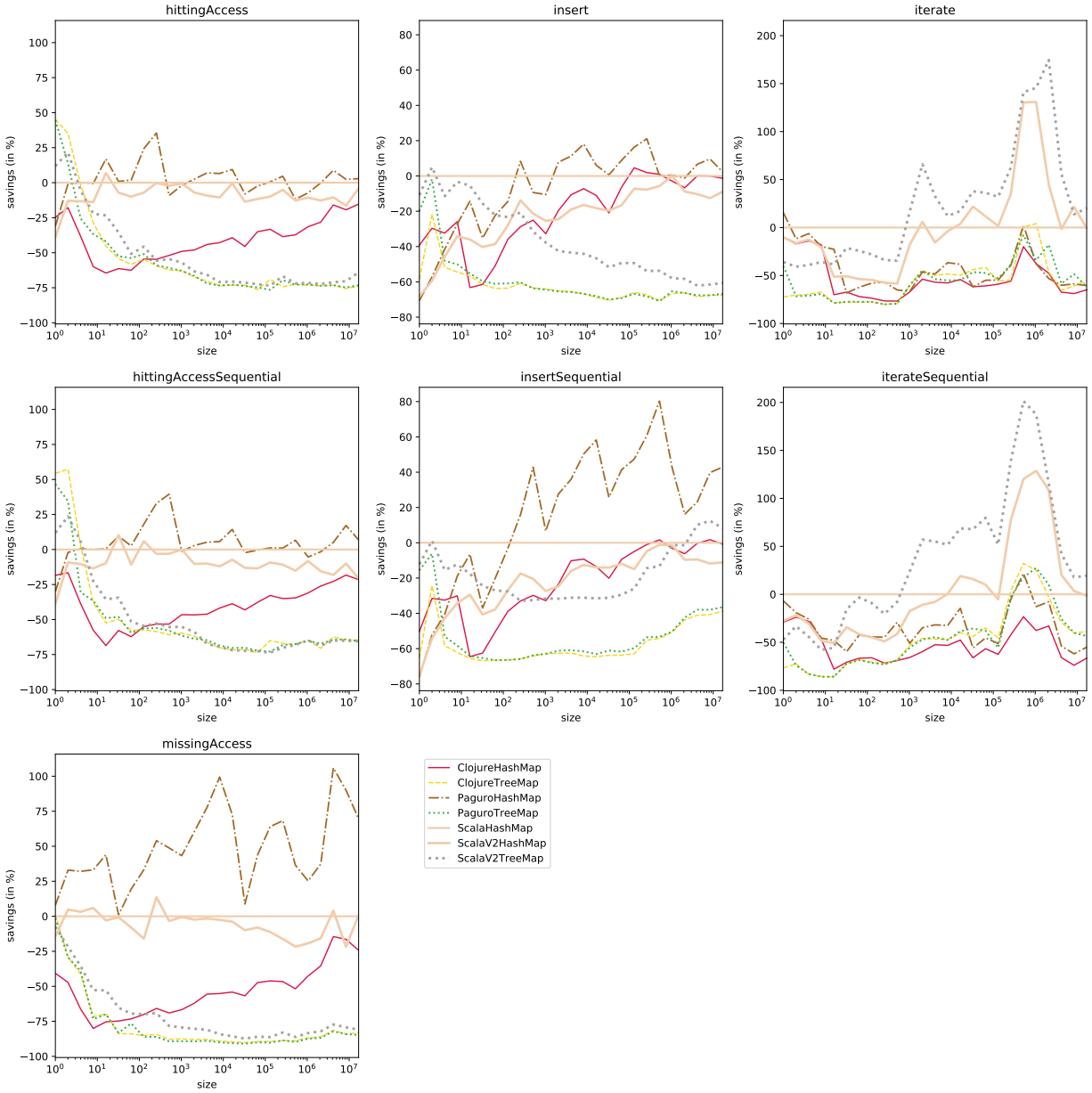


Figure 15: Visualisation of JMH benchmark results for persistent maps that support generic keys that are either comparable or hashable depending on the data structure. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `ScalaHashMap`. On the horizontal axis is the size parameter in logarithmic scale.

As can be seen from Figure 16 the memory usage of the data structures does not change drastically between inserting entries with random or consecutive keys. However, when we compare `sequentialCumulative` and `randomCumulative` benchmarks, we can see that search tree based data structures can save more memory due to structural sharing when the keys are random. For `PaguroHashMap` the spikiness in the memory usage improvement is stronger in the `sequential` benchmark compared

to the `random` benchmark.

If we examine the memory usage differences between the search tree based data structures, we can see that their memory usages follow each other and also that there is a clear ordering between them. The exception is when there are only a few elements, in which case `ClojureTreeMap` takes the edge. The `PaguroTreeMap` uses the least amount of memory among them, followed by `ClojureTreeMap` and then `ScalaV2TreeMap`. However, when we look at the benchmark results for the cumulative cases, while `PaguroTreeMap` still uses the least amount of memory among search trees, the difference between memory usages of `ClojureTreeMap` and `ScalaV2TreeMap` is collapsed when there is more than a few elements.

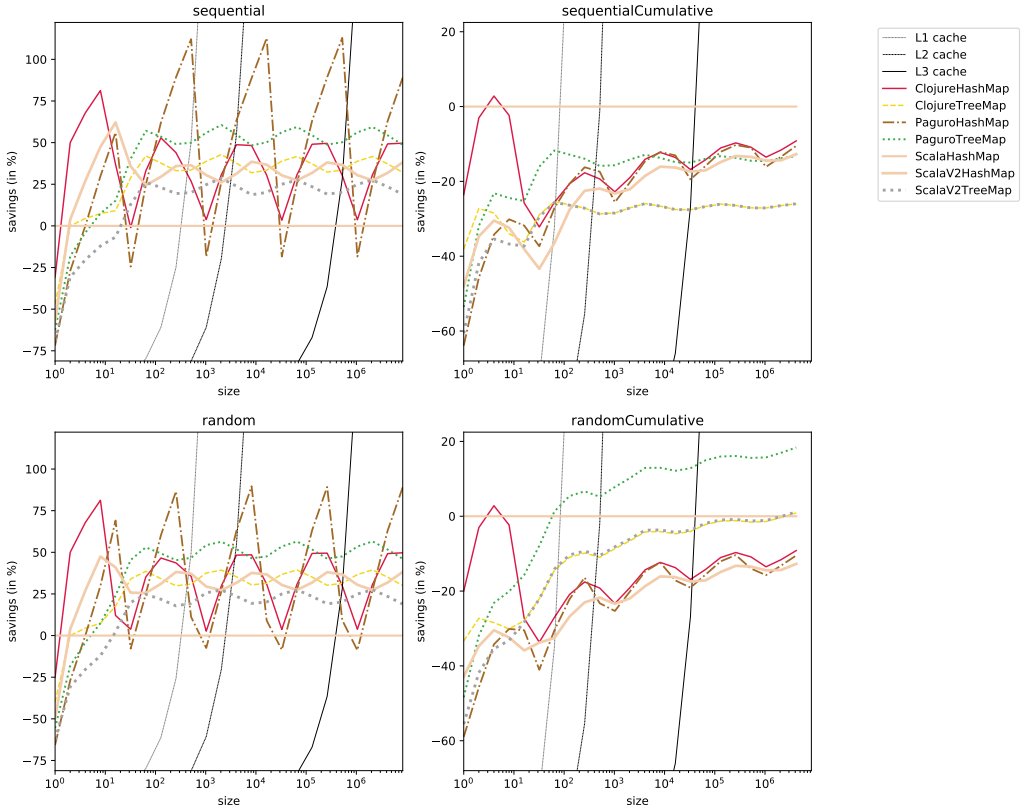


Figure 16: Visualisation of JOL benchmark results for persistent maps that support generic keys that are either comparable or hashable depending on the data structure. On the vertical axis is the memory usage for each data structure normalized against `ScalaHashMap`. On the horizontal axis is the size parameter in logarithmic scale. Normalized L1, L2 and L3 cache memory sizes are also visualised to give an idea of absolute memory usage.

Given that `ScalaHashMap`, `ScalaV2HashMap` and `PaguroHashMap` have the best access benchmark results, but otherwise have different strengths, we will include them in future comparisons. We will also include `ScalaTreeMap` due to good iteration performance.

5.2 Specialized data structures

Now we will compare the data structures that only support integer keys. First we will compare those of them that were implemented specifically for this thesis. Afterwards we will compare the best of the custom data structures with the rest of the specialized ones.

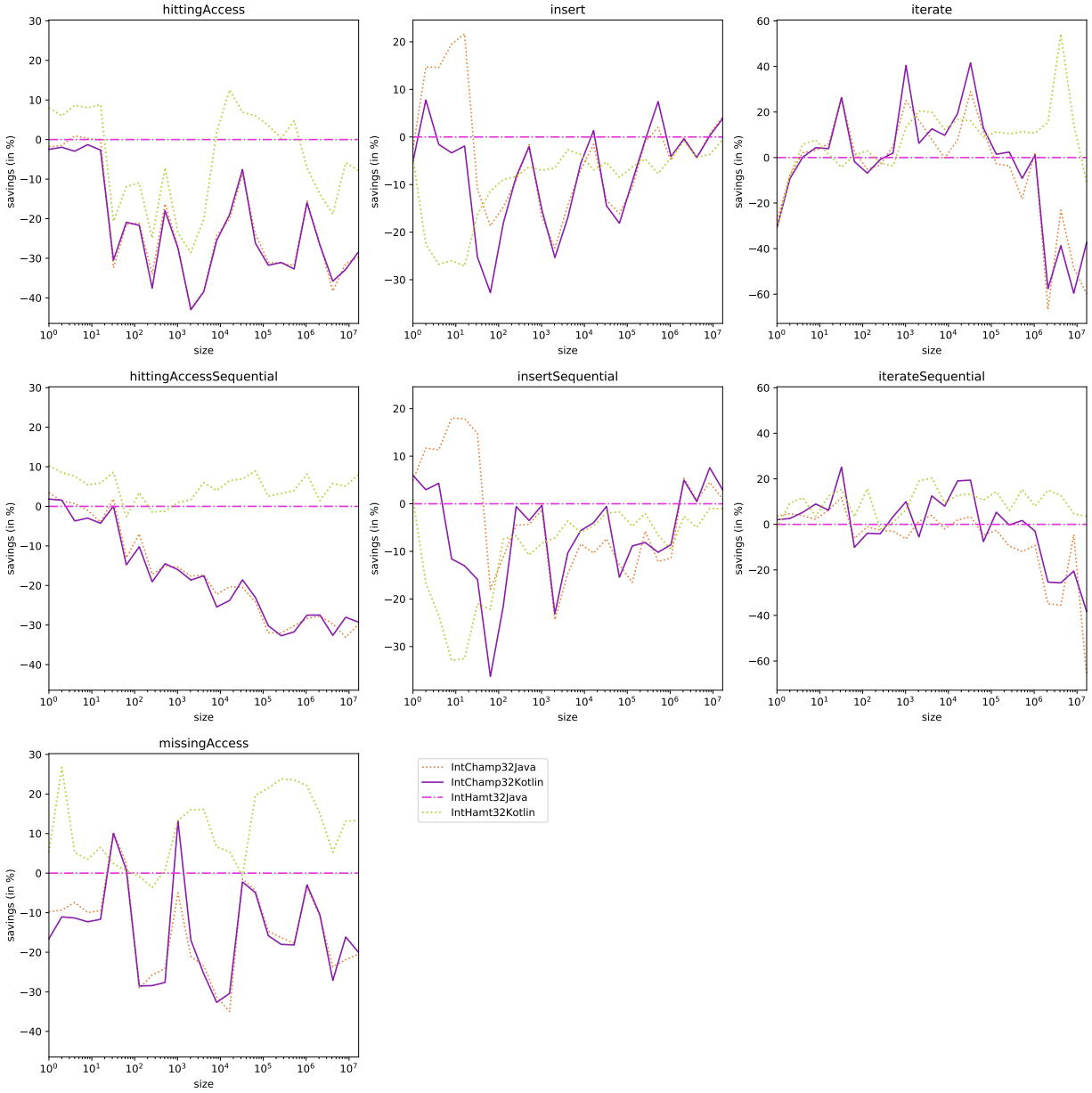


Figure 17: Visualisation of JMH benchmark results for HAMT and CHAMP implementations with branching factor 32 made as part of this thesis that support integer keys. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `IntHamt32Java`. On the horizontal axis is the size parameter in logarithmic scale.

In Figure 17 we see JMH benchmark results of HAMT and CHAMP data structures made for this thesis with branching factor 32 written in both Java and Kotlin normalized against `IntHamt32Java`. From it we can see that there is not much difference between performance of `IntChamp32Java` and `IntChamp32Kotlin` except in the insert benchmarks where `IntChamp32Java` has the edge when the number of elements is less than approximately 200 elements. However, between `IntHamt32Java` and `IntHamt32Kotlin` there are larger differences. In the `hittingAccess` benchmark, `IntHamt32Kotlin` has worse performance between 20 and 7 thousand elements and after 0.7 million elements and is otherwise superior. In the `hittingAccessSequential` benchmark, `IntHamt32Kotlin` has better performance except between 50 and thousand elements, where the performance of the structures are equal. In the `missingAccess` benchmark, `IntHamt32Kotlin` has either equal or better performance. In insert benchmarks `IntHamt32Kotlin` has consistently worse performance, but the difference quickly goes down from around -30% to around -5% as the size grows.

We can also see from Figure 17, when comparing the better of the implementations, that HAMT has significantly better performance compared to the CHAMP implementation in the access benchmarks after 200 elements. In iteration benchmarks, `IntHamt32Kotlin` has the best performance in `iterate` benchmark approximately after $70,000$ elements and in `iterateSequential` benchmark approximately after $40,000$ elements.

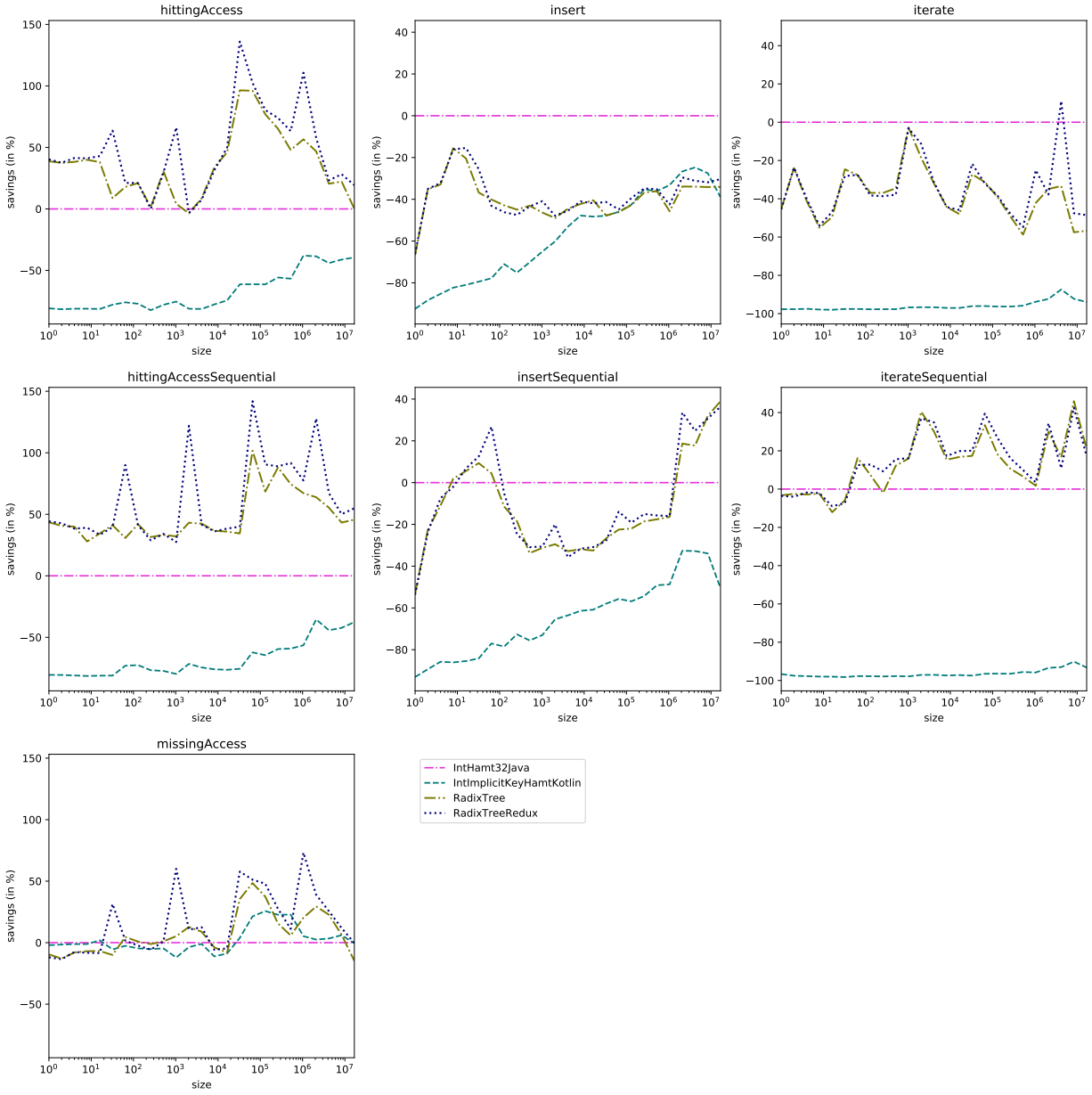


Figure 18: Visualisation of JMH benchmark results for persistent maps made as part of this thesis that support integer keys. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `IntHamt32Java`. On the horizontal axis is the size parameter in logarithmic scale.

Figure 18 shows comparisons of the JMH benchmark results of `IntHamt32Java`, `IntImplicitKeyHamtKotlin`, `RadixTree` and `RadixTreeRedux`. From it we can see that `RadixTreeRedux` has similar or better performance in get benchmarks compared to `RadixTree`. This improvement seems to be more consistent for larger sizes. For the insertion benchmarks the differences are smaller. However, in `insert` benchmark

there seems to be slight improvement for larger sizes[10]. We can also see how the performance of `IntImplicitKeyHamtKotlin` is significantly worse in all but `insert` and `missingAccess` benchmarks.

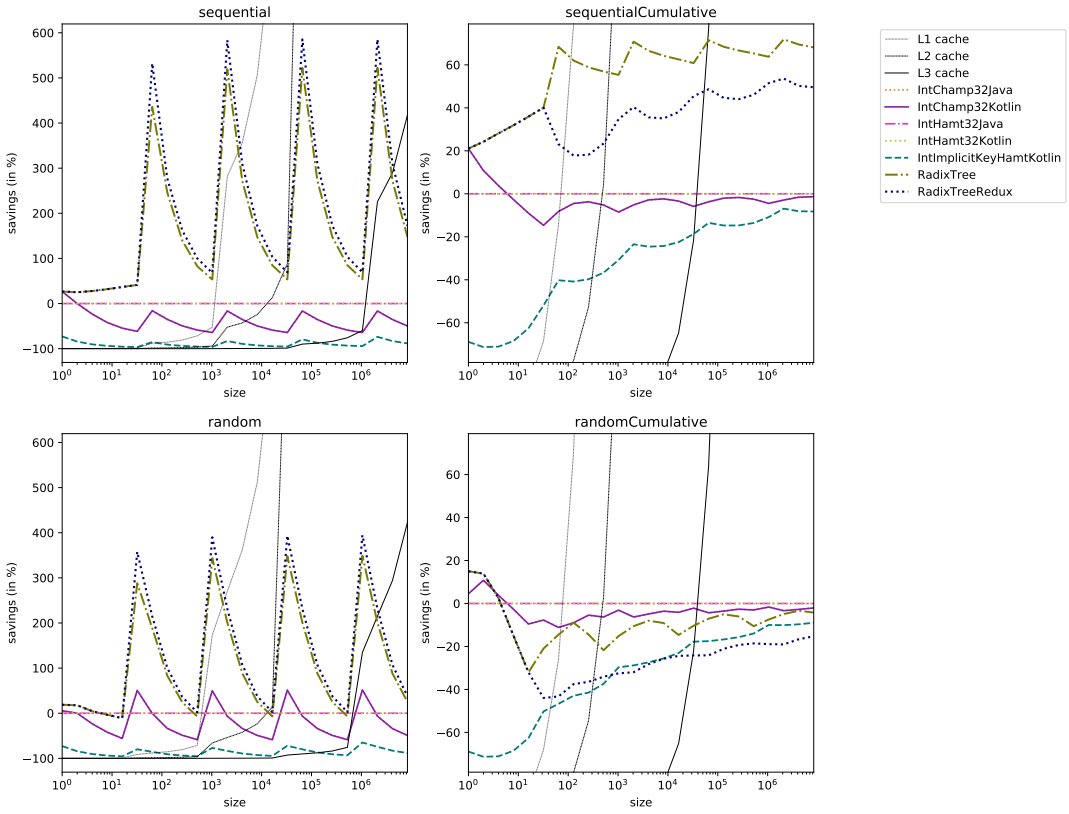


Figure 19: Visualisation of JOL benchmark results for persistent maps made as part of this thesis that support integer keys. On the vertical axis is the memory usage for each data structure normalized against `IntHamt32Java`. On the horizontal axis is the size parameter in logarithmic scale. Normalized L1, L2 and L3 cache memory sizes are also visualised to give an idea of absolute memory usage.

Figure 19 shows a visualisation of the JOL benchmark results for data structures made for this thesis. Firstly, there is no difference in the memory usage between Java and Kotlin versions of the HAMT and CHAMP implementations. We can also see that while `RadixTreeRedux` uses less memory in non-cumulative benchmarks, in cumulative benchmarks this is reversed. Of the data structures, `IntImplicitKeyHamtKotlin` uses most memory, except in `randomCumulative` benchmark where it uses less memory than `RadixTreeRedux` after approximately $6,000$ elements.

[10]Though it seems that the improvement is after approximately 2 thousand elements more precise benchmark reveals that there is a dip in the relative performance approximately after 0.8 million elements which is recovered approximately after 2 million elements. This dip might be artifact of something disturbing the server on which the benchmarks were run. See Appendix A.1 and Appendix A.2 for visualisations for this.

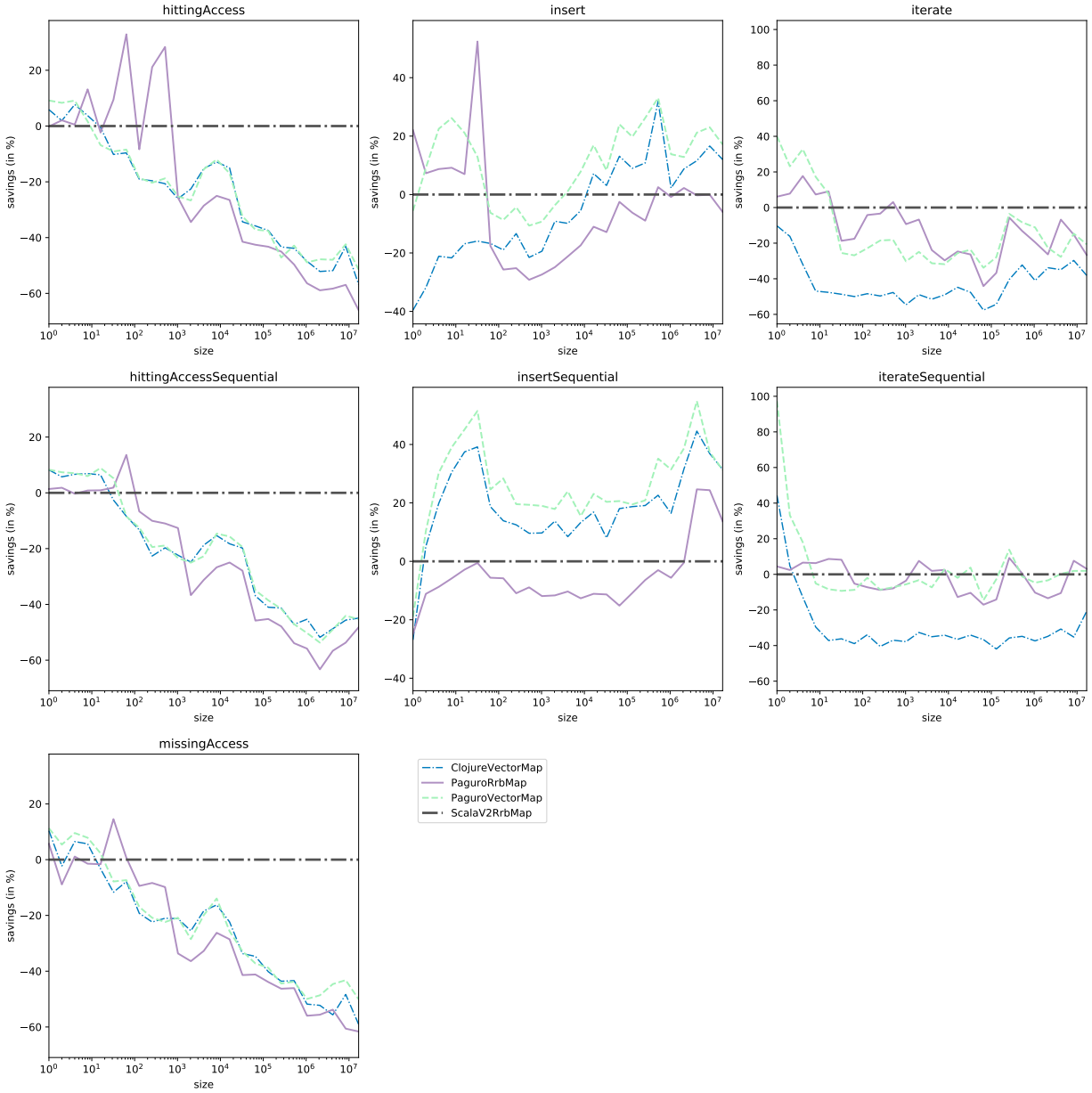


Figure 20: Visualisation of JMH benchmark results for persistent maps made out of persistent vectors from Clojure, Paguro and Scala. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `ScalaV2RrbMap`. On the horizontal axis is the size parameter in logarithmic scale.

In Figure 20 are visualisations of the JMH benchmarks for the map data structures made out of persistent vector implementations from Clojure, Paguro and Scala. From it we can see that `ScalaV2RrbMap` has the best performance in the access benchmarks after approximately $1,000$ elements. We can also notice that the implementation of `PaguroRrbMap` has the slowest access performance, after $1,000$ elements, which is a stark contrast to the Scala's implementation. If we look at the ac-

cess performance of `PaguroVectorMap` and `ClojureVectorMap`, they are quite close to each other, which is to be expect as the implementation in Paguro is a derivative of Clojure's implementation. However, in the insertion and iteration benchmarks there is larger difference between `PaguroVectorMap` and `ClojureVectorMap` implementations with Paguro's being superior.

In the `insert` benchmark, `PaguroRrbMap` has the slowest insertion speed between approximately 60 elements and 0.4 million elements. After that the insertion speed of it is tied with that of `ScalaV2RrbMap`. However, `PaguroRrbMap` is again slower after 7 million elements, however it is hard to say how things will pan out with larger sizes than ones included in the benchmark. In the `insertSequential` benchmark, `PaguroRrbMap` is worse until 2 million elements and after catching `ScalaV2RrbMap` its performance is clearly better. Similarly to the `insert` benchmark, the relative performance of `PaguroRrbMap` starts to become worse after 9 million elements. In the iteration benchmarks, the performance of `ClojureVectorMap` is worst after just a few elements. In the `iterate` benchmark, `ScalaV2RrbMap` has the best performance after approximately 20 elements, with `PaguroRrbMap` having briefly similar performance between approximately 400 and 600 elements. In the `iterateSequential` benchmark, the relative iteration performances of `PaguroRrbMap`, `PaguroVectorMap` and `PaguroRrbMap`, after few elements, fluctuate within 20% of each other.

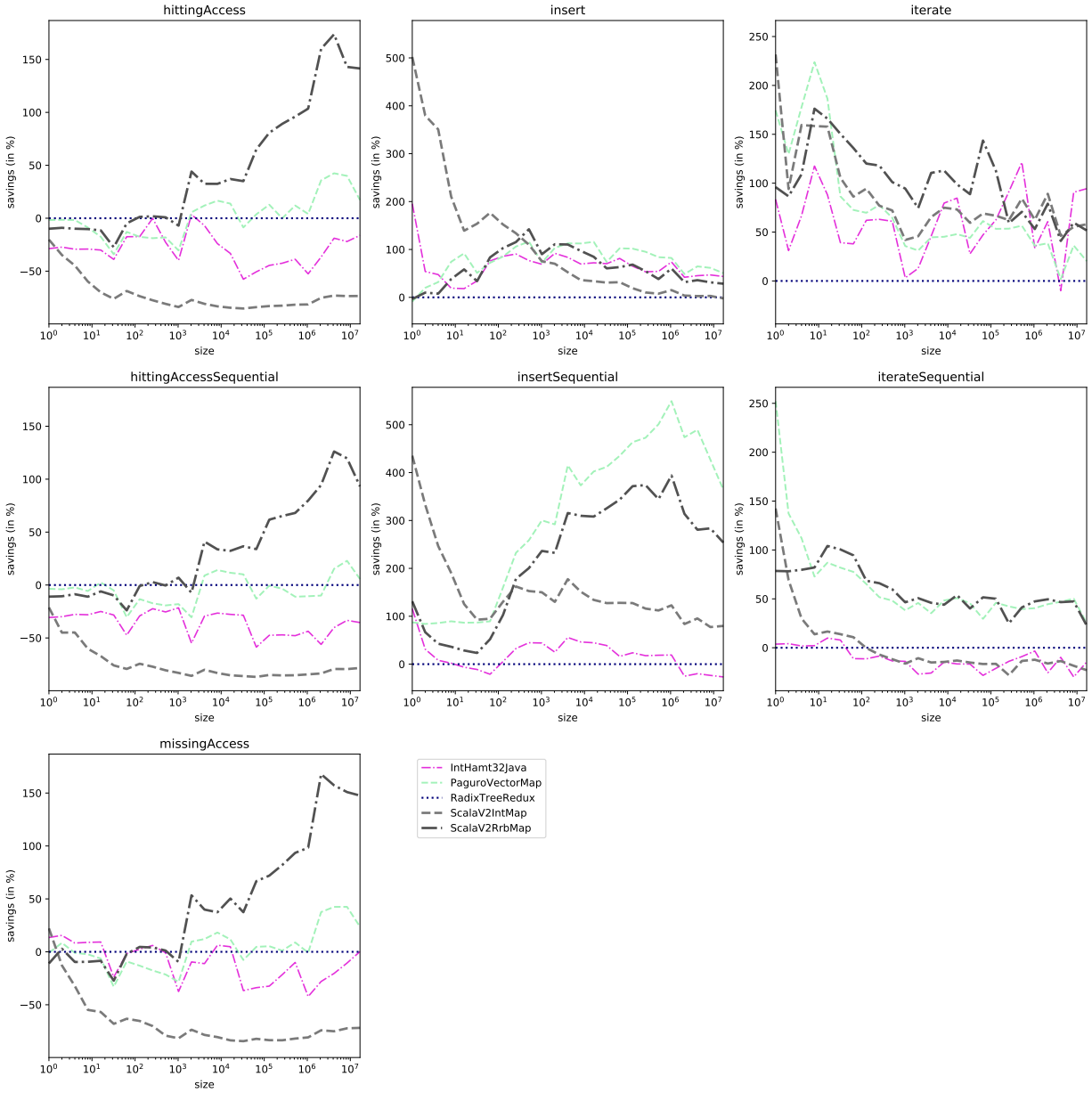


Figure 21: Visualisation of JMH benchmark results for persistent maps that support integer keys. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `RadixTreeRedux`. On the horizontal axis is the size parameter in logarithmic scale.

In Figure 21 we see the JMH benchmark results for the data structures `PaguroVectorMap`, `ScalaV2RrbMap`, `RadixTreeRedux` and `IntHamt32Java`. We can see that, while `RadixTreeRedux` has performance comparable or better access performance to `PaguroVectorMap`, its insertion and iteration performances are clearly worse. Similarly all of the other compared data structures are overall worse than `PaguroVectorMap` and `ScalaV2RrbMap`, even if there are some areas they might be better at some points.

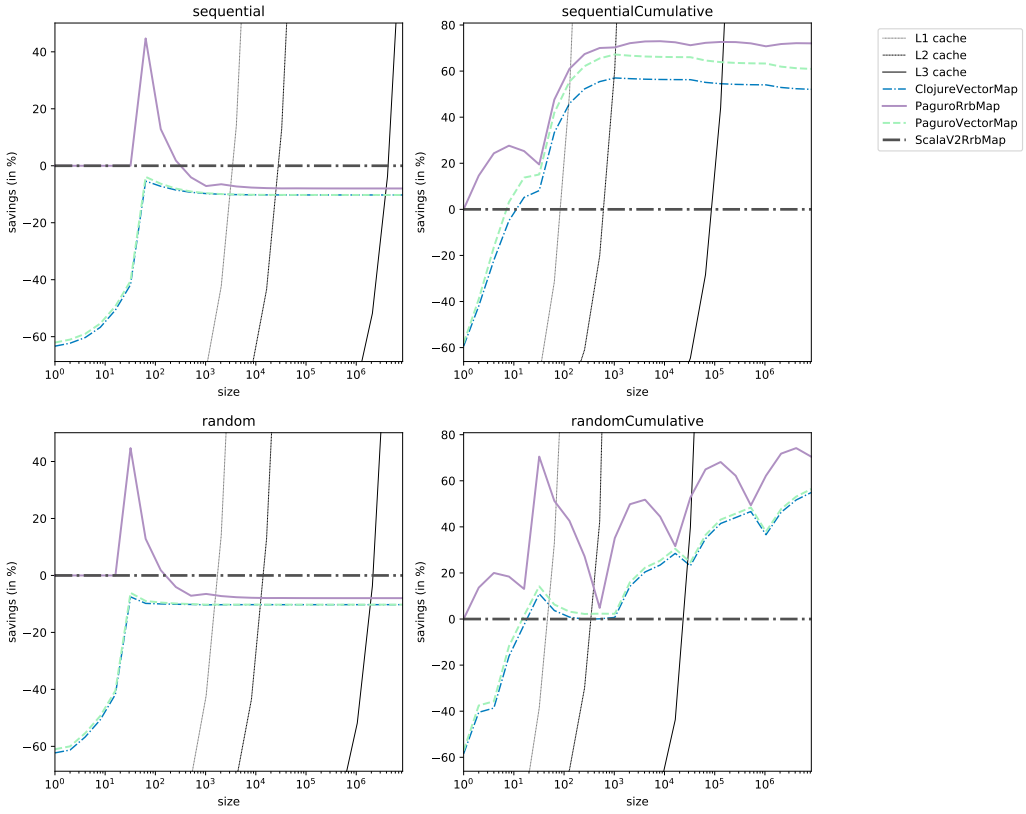


Figure 22: Visualisation of JOL benchmark results for persistent maps made out of persistent vectors from Clojure, Paguro and Scala. On the vertical axis is the memory usage for each data structure normalized against `ScalaV2RrbMap`. On the horizontal axis is the size parameter in logarithmic scale. Normalized L1, L2 and L3 cache memory sizes are also visualised to give an idea of absolute memory usage.

In Figure 22 are the JOL benchmark results visualised for the map data structures made out of persistent vector implementations from Clojure, Paguro and Scala. From it we can see that the memory usage of `PaguroVectorMap` and `ClojureVectorMap` is close to each other, which makes sense as Paguro's implementation is a derivative of Clojure's. However, in the `sequentialCumulative` benchmark, there is a larger difference with `PaguroVectorMap` being superior. For `ScalaV2RrbMap` we can see an inversion in its performance: in both non-cumulative benchmarks it uses the least amount of memory and then in both cumulative benchmarks, after approximately 20 elements, it uses the most. `PaguroRrbMap` uses second most memory in the non-cumulative benchmarks after approximately 200 elements. For it there is no inversion in its performance: it uses the least amount of memory in the cumulative benchmarks.

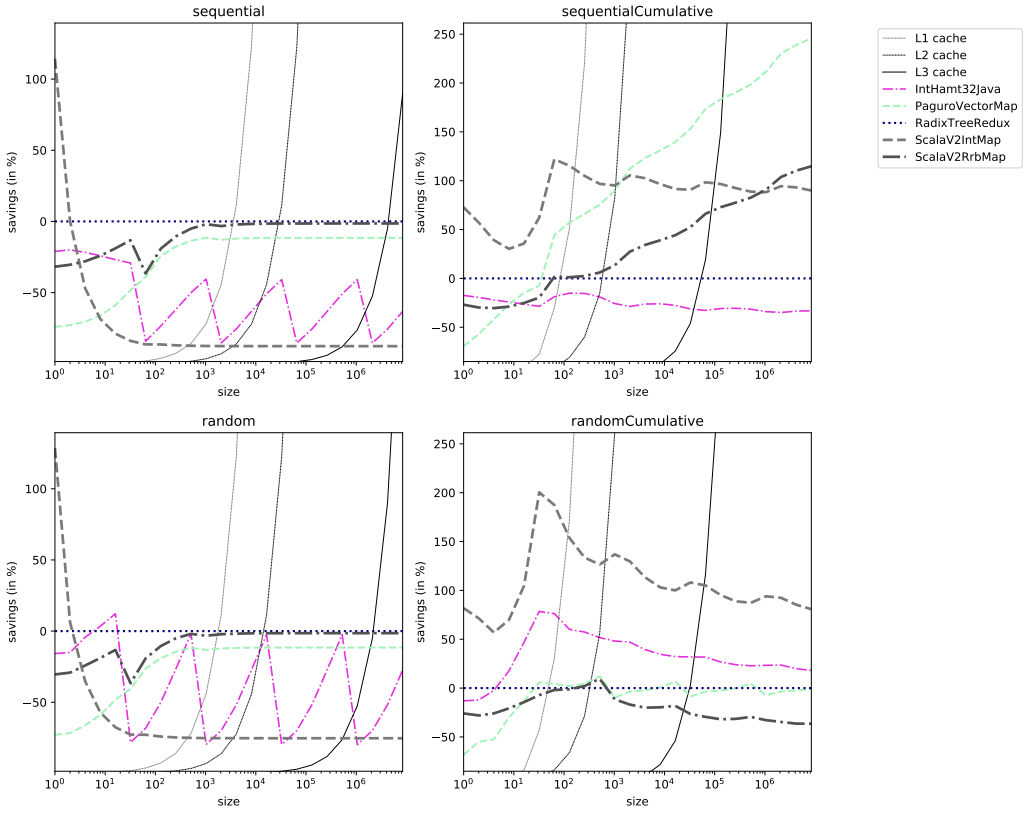


Figure 23: Visualisation of JOL benchmark results for persistent maps that support integer keys. On the vertical axis is the memory usage for each data structure normalized against `RadixTreeRedux`. On the horizontal axis is the size parameter in logarithmic scale. Normalized L1, L2 and L3 cache memory sizes are also visualised to give an idea of absolute memory usage.

In Figure 23 are the JOL benchmark results visualised for the data structures `PaguroVectorMap`, `ScalaV2RrbMap`, `RadixTreeRedux` and `IntHamt32Java` normalized against `RadixTreeRedux`. In non-cumulative benchmarks we can see that `RadixTreeRedux` uses the least amount of memory when it contains more than a few elements. We can also see that `ScalaV2RrbMap` catches `RadixTreeRedux`'s memory usage at thousand elements, but after which the advantage of `ScalaV2RrbMap` is only few percentages. In the `sequential` benchmark, after a few elements, `ScalaV2IntMap` uses the most memory. In the `random` benchmark this is also nearly always the case however, because `IntHamt32Java`'s relative memory usage fluctuates, `IntHamt32Java` sometimes uses more memory than it.

For the cumulative benchmarks there are stronger differences between `sequentialCumulative` and `randomCumulative` benchmarks. In the `sequentialCumulative` benchmark, `PaguroVectorMap` uses the least amount of memory after approximately thousand elements. In the `randomCumulative` benchmark, `PaguroVectorMap`'s memory usage is around that of `RadixTreeRedux`, after few elements, fluctuating between using slightly more or less memory. In the `randomCumulative`

benchmark, `ScalaV2IntMap` uses the least amount of memory. In the `sequential Cumulative` benchmark, it starts at the first place, but is then surpassed by first `PaguroVectorMap` at approximately thousand elements and then by `ScalaV2Rrb Map` at 0.6 million elements. Before attaining second place `ScalaV2RrbMap` is at the third place starting from around 50 elements. In the `randomCumulative` benchmark there is a reversal in performance for it and so `ScalaV2RrbMap` uses the most memory after approximately 500 elements. Interestingly, even though `IntHamt32Java` still performs poorly in the `sequentialCumulative` benchmark, using the most memory after 20 or so elements, it also uses the second least amount of memory in the `random Cumulative` benchmark. The worse performance of vector based maps in `random Cumulative` is explainable by the fact they need to be filled with empty entries until the index where the value is being inserted. The effect of this filling can also be seen in non-cumulative benchmarks where the lead of vector based maps is smaller in the `random` benchmark compared to the `sequential` benchmark.

Based on the previous analysis `ScalaV2RrbMap` has the edge in access benchmarks and `PaguroVectorMap` in the insert benchmarks and so we include both of them in future comparisons.

5.3 Results in context

We now compare, in Figure 24, JMH benchmark results for some of the best data structures from the previous analysis: `PaguroHashMap`, `PaguroVectorMap`, `Scala HashMap`, `ScalaV2HashMap`, `ScalaV2RrbMap` and `ScalaV2TreeMap`. These are compared against `ArrayMap` and `SdkMap`, which are used as baseline ephemeral data structures.

Immediately noticeable from the figure is how well `ArrayMap` performs in both insertion and access benchmarks. This was to be expected as `ArrayMap` operates over a contiguous piece of memory. Examining the `insert` benchmark, the speed of random insertions to the `ArrayMap` starts to suffer when its backing array is too large for the L3 cache. Even after that it still stays faster than most of the other data structures and is only beaten by the `SdkMap` between 2 and 10 million elements. However, what is unexpected is how relatively badly `ArrayMap` performs in the iteration benchmarks. In the `iterate` benchmark, this slowness can be explained by the fact that the visualised density is 0.5. This means that there is on average a gap of size 1 between each element, which the iteration has to skip slowing it down. However, in the `iterateSequential` benchmark, where all of indices from 0 to n are filled, beating its iteration performance should be hard. Also, at the largest sizes, `ArrayMap` starts to falter in both `hittingAccess` and `missingAccess` benchmarks, but based on this data it is hard to say what will be the trend with even larger sizes.

`SdkMap` also has advantage in random insertions before going out of L3 cache. The advantage compared to persistent data structures is because it only has to do traversals in the linked list, which it uses for separate chaining, when there are collisions. This is in contrast with persistent structures which are all essentially trees and so

always need to traverse to leaves. The focus optimizations [27], that optimize operations that target the end or the indices next to previous modifications, do not help persistent data structures as the insertion target is random and focus is rarely hit. `SdkMap` performs similarly in the `insertSequential` benchmark as it does in the `insert` benchmark, which is due to hashing the keys. Because of their good append performance, RRB-Trees catch sequential insert performance of `SdkMap` once `SdkMap` goes out of L3 cache.

From Figure 24 we can clearly distinguish, in both `hittingAccess` and `hittingAccessSequential` benchmarks, the three different types of data structures: search tree based, hash based and integer keyed ones from slowest to fastest. Interestingly, in `missingAccess` this is almost the case again and only the `PaguroHashMap` breaks it by having performance close to `PaguroVectorMap` in the benchmark. `ScalaV2RrbMap` takes all of the access benchmarks as the fastest persistent data structure. It even surpasses `SdkMap`, when the element exists in the data structure, only losing to `SdkMap` after approximately 7 million elements. In `insertSequential` we can clearly see that both `ScalaV2RrbMap` and `PaguroVectorMap`, which are persistent vectors, are highly optimized for appending to the end of the data structure. In all of the insertion benchmarks `PaguroVectorMap` is the fastest persistent data structure. As we have seen from the previous analysis, iteration benchmarks are the most complex ones to analyse. In the `iterate` benchmark, among compared data structures, we can see that `ScalaHashMap` has the best performance for small sizes and `ScalaV2TreeMap` has the best performance for large sizes[11]. In the `iterateSequential` benchmark, `ScalaHashMap` still has the best performance for small sizes. However, `ScalaV2TreeMap`'s lead is first surpassed by `ArrayMap` and then eventually by `PaguroVectorMap` and `ScalaV2RrbMap` for large sizes.

[11] Around 8 million elements `ScalaHashMap` and `ArrayMap` have better performance than `ScalaV2TreeMap`, but from the precision of benchmarks it is hard to say exactly what is happening there and the `ScalaV2TreeMap` surpasses them afterwards. The situation is most likely explainable by important parts of the data structures going out of L3 cache at different sizes.

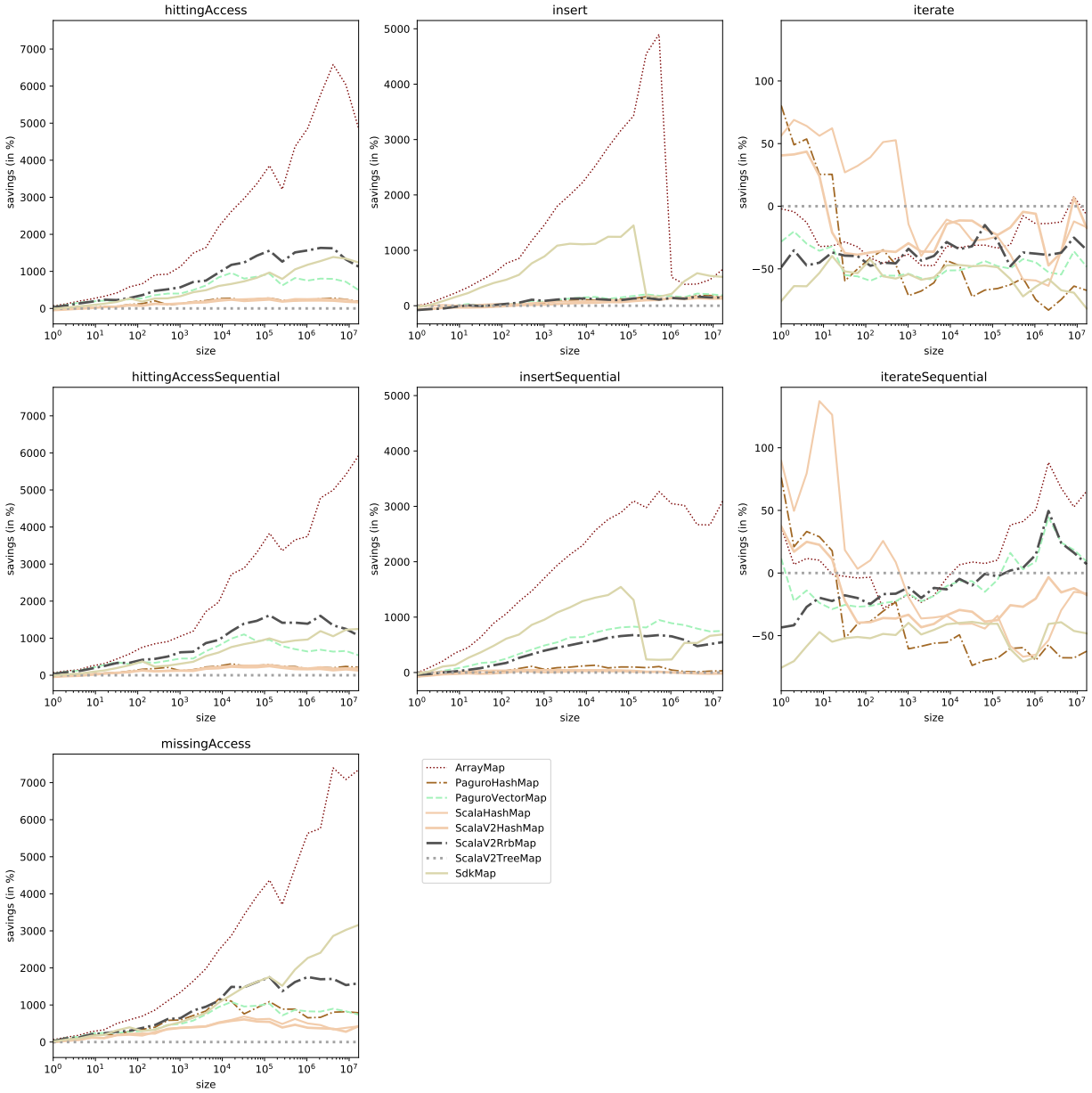


Figure 24: Visualisation of JMH benchmark results of best data structures based on previous analysis compared to `ArrayMap` and `SdkMap`. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `ScalaV2TreeMap`. On the horizontal axis is the size parameter in logarithmic scale.

We now compare, in Figure 25, JOL benchmark results for some of the best data structures from the previous analysis: `PaguroHashMap`, `PaguroVectorMap`, `ScalaHashMap`, `ScalaV2HashMap`, `ScalaV2RrbMap` and `ScalaV2TreeMap`. These are compared against `ArrayMap` and `SdkMap`, which are used as baseline ephemeral data structures. As can be immediately seen in the cumulative benchmarks, the memory usage of both `ArrayMap` and `SdkMap` skyrockets as they are ephemeral data struc-

tures and the naive persistence applied to them quickly uses all available memory. In the `sequential` benchmark we can see that the `ArrayMap` uses the least memory, however in the `random` benchmark both `ScalaV2RrbMap` and `PaguroVectorMap` perform better. We can also see that the memory usage of `SdkMap` is between that of `ScalaHashMap` and `ScalaV2HashMap`.

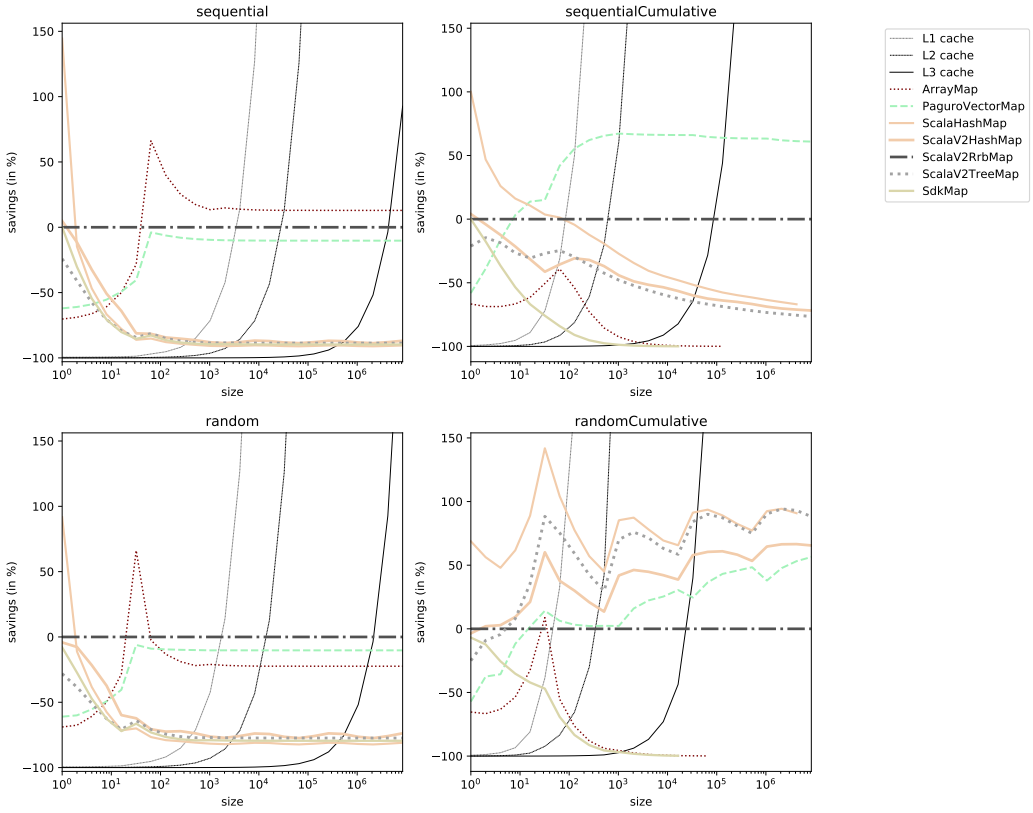


Figure 25: Visualisation of JOL benchmark results of best data structures based on previous analysis compared to `ArrayMap` and `SdkMap`. The density parameter is set to 0.5. On the vertical axis is the memory usage for each data structure normalized against `ScalaV2TreeMap`. On the horizontal axis is the size parameter in logarithmic scale. Normalized L1, L2 and L3 cache memory sizes are also visualised to give an idea of absolute memory usage.

If persistence is required from a data structure, then the previous analysis suggest the following. From the benchmark results it can be concluded that `ScalaV2RrbMap` is most likely the best data structure choice, as access operations are often the most prevalent operations and it is the leader in their benchmarks. If we know that all values from 0 to n are present and that insertion is strongly bottleneck, then `PaguroVectorMap` might be better choice. If we know that we have lots of small data structures and iteration is strongly bottleneck, then using `ScalaHashMap` might be worth to benchmark. If the data is distributed randomly and iteration is strongly bottleneck, then `ScalaV2TreeMap` might be better choice. In Figure 26 and Figure 27 their JMH and JOL benchmark results can be seen visualised.

Based on the analysis, the most suitable of the benchmarked persistent data structures for mappings is `ScalaV2RrbMap`, however its access performance regression, to the currently used data structure, is quite steep. The benchmarks visualised in Appendix A.6 and the benchmarks conducted in [24] for RRB-Trees, suggest that increasing branching factor would improve the access performance. So while existing data structures might not be usable as mappings, the techniques used in their implementations can possibly be used to implement one that is.

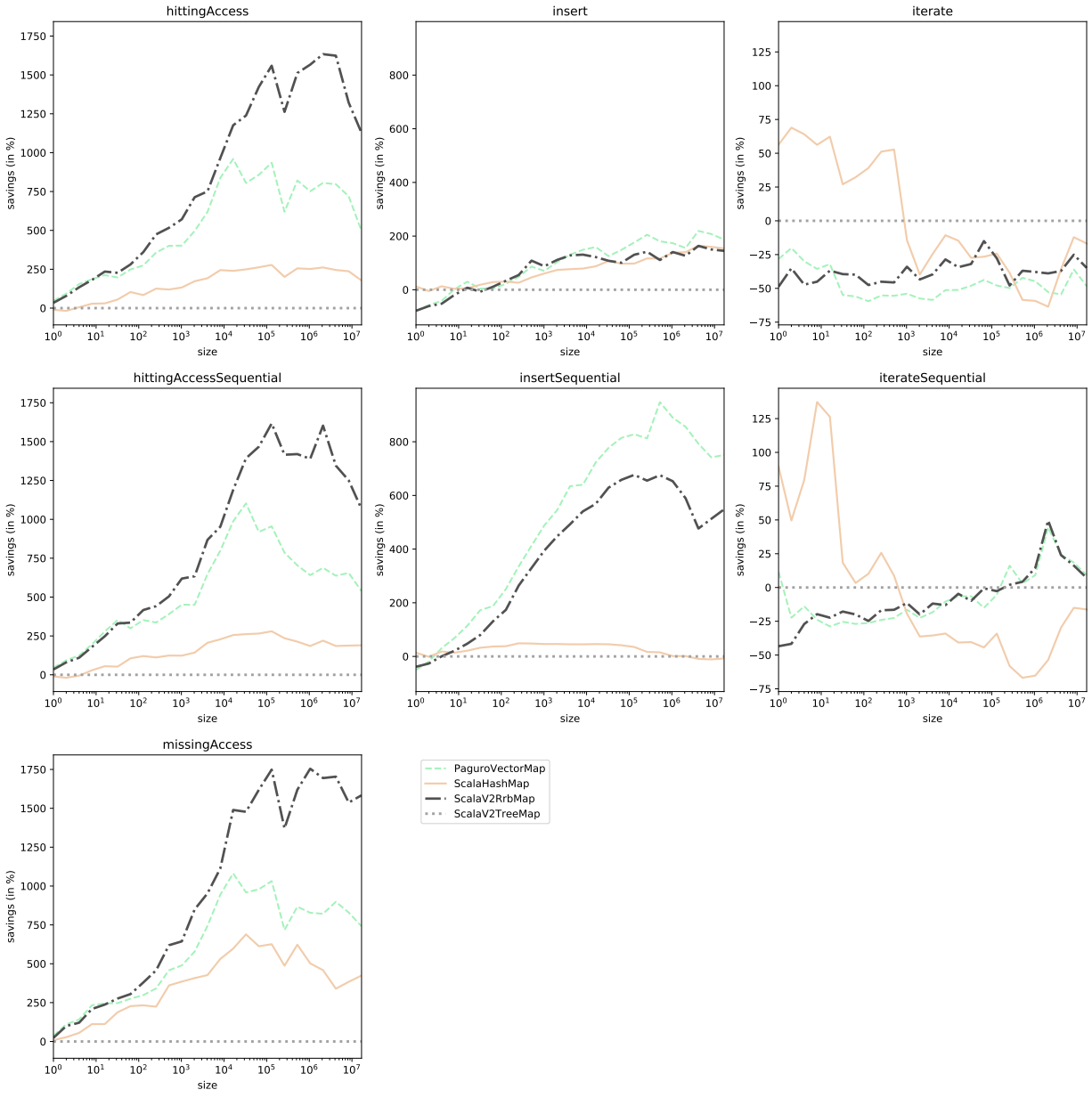


Figure 26: Visualisation of JMH benchmark results of best data structures based on previous analysis. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `ScalaV2TreeMap`. On the horizontal axis is the size parameter in logarithmic scale.

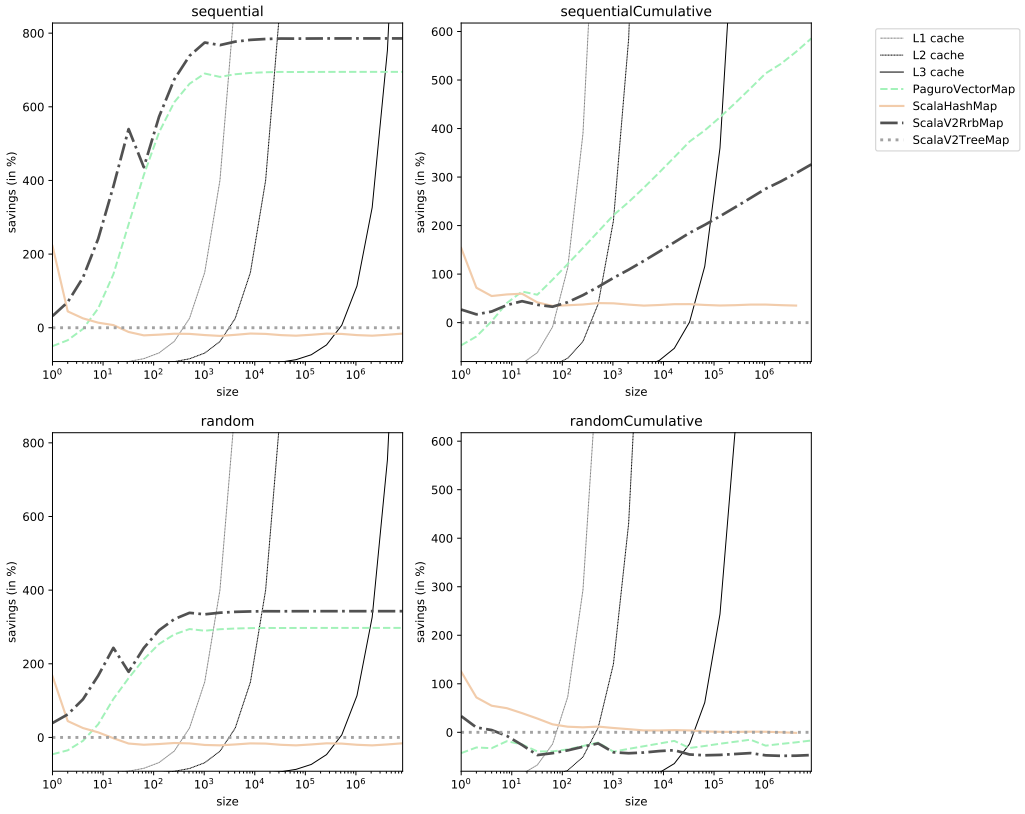


Figure 27: Visualisation of JOL benchmark results of best data structures based on previous analysis. The density parameter is set to 0.5. On the vertical axis is the memory usage for each data structure normalized against `ScalaV2TreeMap`. On the horizontal axis is the size parameter in logarithmic scale. Normalized L1, L2 and L3 cache memory sizes are also visualised to give an idea of absolute memory usage.

6 Conclusion

In this thesis we have first gone through important design decisions that affect database systems, explored how these have affected an actual implementation of a database focusing on join-indices, seen variety of different persistent data structures and then lastly evaluated them empirically.

The previously used data structure used for join indices, that persistent data structures were compared against, was essentially dynamically growing array where indices were used as keys. From the empirical evaluation we can see that the previously used solution is clearly superior from performance perspective, however persistence does save lots of memory when there exists even a couple versions of the same data structure. It is possible that increasing branching factor would improve access speed to acceptable level while keeping at least some of the memory benefits of persistence and this could be investigated next. The benchmarks also show that there are huge differences in the performance of different implementations of the same data struc-

ture. Among the persistent data structures, persistent vectors in form of RRB-Trees are clearly fastest for access, being able to even overtake the performance of Java's standard hash map. However, the search tree based data structures were able to overtake RRB-Trees in iteration benchmarks, if the data structures were randomly filled.

The benchmark results of this thesis are far from complete. As seen from the Appendix A.1 and Appendix A.2 that, even though the one with less samples does give idea on the relative performance, increasing sampling density does show details missing from it. However these details might be caused by external factors and might not be wholly trustworthy. For this specific data structure, where the improved version should be better for 20% of the elements, illustrates the limitations of the benchmarks. If the taken samples would have hit only those 20% of elements, we might have come to too positive conclusion. Also, in most of the graphs, the margin of error, with confidence level 99.9%, was left out for visual clarity. However, when leaving it out we also lose the sense of uncertainty of benchmarks and might make too decisive conclusions. There is also the matter of using a logarithmic scale for visualisations, which is able to show trends more clearly, but which makes it harder to realize how changes in the graph in larger sizes have exponentially more weight. This is illustrated well by the weighted average and variances drawn to Appendix A.3 which might look "wrong" from casual glance as the average seems not to be middle of the graph, but if looked at without a logarithmic x-axis in Appendix A.4 look "correct". When an average is illustrated with the trend of a benchmark visible it does not mislead, but average should not be used as the value when condensing benchmark results to a single value and instead geometric mean should be used [33]. While writing the thesis it was noticed that the scale used in visualisations is important: The author of the thesis was predisposed to think that the choice of the programming language would not affect the performance and for a while the benchmark results seemed to support this, but after changing how the data was visualized, to get clearer idea on the percentages of change, it was apparent that benchmarks do not actually support this interpretation.

6.1 Future work

The following topics of interest were not investigated in the thesis, but could be fruitful future work. The thesis did not answer why `ClojureHashMap` performs so shockingly poorly in the benchmarks. Finding this out requires further investigation. The reason for the relatively poor performance of `ArrayMap` for sequential iteration was not explained. Even though the benchmarks varied the density parameter, no throughout analysis for effect of it was done. The effect of different branching factor for HAMT data structures was benchmarked, but a bug was found in the implementation. At that point the server used for benchmarking had broken and there was not enough time for rerunning all of the benchmarks. Based on the benchmarks on the new server larger branching factor makes access benchmarks faster and insertion benchmarks slower. See Appendix A.6 for visualisation of those

benchmark runs. No memory benchmarks were run for the fixed version, but the hypothesis is that larger branching factor decreases the memory usage of a single data structure due it being shallower and increases the memory usage of the cumulative benchmarks as more data has to be copied on the paths from root to leaves. The effect of different branching factors was not benchmarked for RRB-Trees, but in [24] time benchmarks were done, from which it was concluded that the branching factor 32 is a good compromise between insertion and access speeds. For the custom HAMT implementations the effect of hashing and the effect of separate key and value arrays was not investigated. It would be interesting to thoroughly investigate how much different collision handling strategies effect the performance. The implementations made for the thesis were not fully optimized and it would be interesting to investigate the effects of different micro-optimizations. The effects of focus optimization, which is the reason inserting to the end of data structures was fast for many of the data structures, was not investigated [27]. Effects of transience, where persistent data structures can implicitly or explicitly modified, to skip unnecessary version creation, was not investigated, though it was used to optimize insertion of large values to RRB-Tree based maps [34]. One future direction of investigation is support of multi-maps using persistent data structures [35].

References

1 H. Plattner, "A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database," in Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD '09, (New York, NY, USA), p. 1–2, Association for Computing Machinery, 2009.

2 H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," IEEE Transactions on Knowledge and Data Engineering, vol. 27, pp. 1920–1948, July 2015.

3 J. Berg, "Query Optimizing for On-line Analytical Processing; Förfrågningsoptimering för uppkopplad analytisk bearbetan," g2 pro gradu, diplomityö, Aalto University, 2017.

4 Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An Empirical Evaluation of In-Memory Multi-Version Concurrency Control," Proc. VLDB Endow., vol. 10, p. 781–792, March 2017.

5 A. Adya, Weak consistency: a generalized theory and optimistic implementations for distributed transactions. PhD thesis, Massachusetts Institute of Technology, 1999.

6 A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073), pp. 67–78, IEEE, 2000.

7 H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, SIGMOD '95, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 1995.

8 R. Sethi, "Useless actions make a difference: Strict serializability of database updates," Journal of the ACM (JACM), vol. 29, no. 2, pp. 394–403, 1982.

9 D. R. K. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," Proceedings of the VLDB Endowment, vol. 5, p. 1850–1861, August 2012.

10 J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han, "Hybrid garbage collection for multi-version concurrency control in sap hana," in Proceedings of the 2016 International Conference on Management of Data, pp. 1307–1318, 2016.

11 P. Valduriez, "Join indices," ACM Trans. Database Syst., vol. 12, p. 218–246, June 1987.

12 Z. Xie and J. Han, "Join index hierarchies for supporting efficient navigations in object-oriented databases," in VLDB, vol. 94, pp. 12–15, 1994.

13 H. Homann and F. Laenen, "SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes," Computer Physics Communications, vol. 224, pp. 325 – 332, 2018.

14 V. N. Anh and A. Moffat, "Inverted Index Compression Using Word-Aligned Binary Codes," Information Retrieval, vol. 8, pp. 151–166, Jan 2005.

15 J. Abel, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot, "Applications tuning for streaming SIMD extensions," Intel Technology Journal Q, vol. 2, pp. 1–12, 1999.

16 A. Karikoski, "Case study on the compression techniques of a column oriented database," Master's thesis, University of Helsinki, 2019.

17 A. Weininger, "Efficient execution of joins in a star schema," in Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, (New York, NY, USA), p. 542–545, Association for Computing Machinery, 2002.

18 P. Vassiliadis and T. Sellis, "A Survey of Logical Models for OLAP Databases," SIGMOD Rec., vol. 28, p. 64–69, December 1999.

19 P. Wadler, "Monads for functional programming," in Advanced Functional Programming (J. Jeuring and E. Meijer, eds.), (Berlin, Heidelberg), pp. 24–52, Springer Berlin Heidelberg, 1995.

20 A. Fiat and H. Kaplan, "Making data structures confluently persistent," Journal of Algorithms, vol. 48, no. 1, pp. 16 – 58, 2003. Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms.

21 P. Bagwell, "Fast And Space Efficient Trie Searches," tech. rep., École Polytechnique Fédérale de Lausanne, 2000.

22 P. Bagwell, "Ideal Hash Trees," tech. rep., École Polytechnique Fédérale de Lausanne, 2001.

23 M. J. Steindorfer and J. J. Vinju, "Optimizing Hash-Array Mapped Tries for Fast and Lean Immutable JVM Collections," in Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, (New York, NY, USA), p. 783–800, Association for Computing Machinery, 2015.

24 P. Bagwell and T. Rompf, "RRB-Trees: Efficient Immutable Vectors," Infoscience, p. 16, 2011.

25 S. Thompson, Type Theory and Functional Programming. Addison Wesley, June 1991.

26 K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech, "PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language," ACM Trans. Program. Lang. Syst., vol. 25, p. 225–290, March 2003.

27 N. Stucki, T. Rompf, V. Ureche, and P. Bagwell, "RRB Vector: A Practical General Purpose Immutable Sequence," in Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, (New York, NY, USA), p. 342–354, Association for Computing Machinery, 2015.

28 D. Brink, "A (probably) exact solution to the Birthday Problem," The Ramanujan Journal, vol. 28, pp. 223–238, June 2012.

29 C. Okasaki, Purely functional data structures. Cambridge University Press, 1999.

30 C. Okasaki and A. Gill, "Fast mergeable integer maps," in Workshop on ML, pp. 77–86, 1998.

31 G. Marsaglia and J. C. W. Marsaglia, "A New Derivation of Stirling's Approximation to n!," The American Mathematical Monthly, vol. 97, no. 9, pp. 826–829, 1990.

32 R. Hinze, "Constructing red-black trees," in Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL, vol. 99, pp. 89–99, 1999.

33 P. J. Fleming and J. J. Wallace, "How not to lie with statistics: the correct way to summarize benchmark results," Communications of the ACM, vol. 29, no. 3, pp. 218–221, 1986.

34 J. P. B. Puente, "Persistence for the Masses: RRB-Vectors in a Systems Language," Proc. ACM Program. Lang., vol. 1, pp. 1–28, August 2017.

35 M. J. Steindorfer and J. J. Vinju, "To-many or to-one? all-in-one! efficient purely functional multi-maps with type-heterogeneous hash-tries," SIGPLAN Not., vol. 53, p. 283–295, April 2018.

A Appendix

A.1 Comparing radix tree implementations

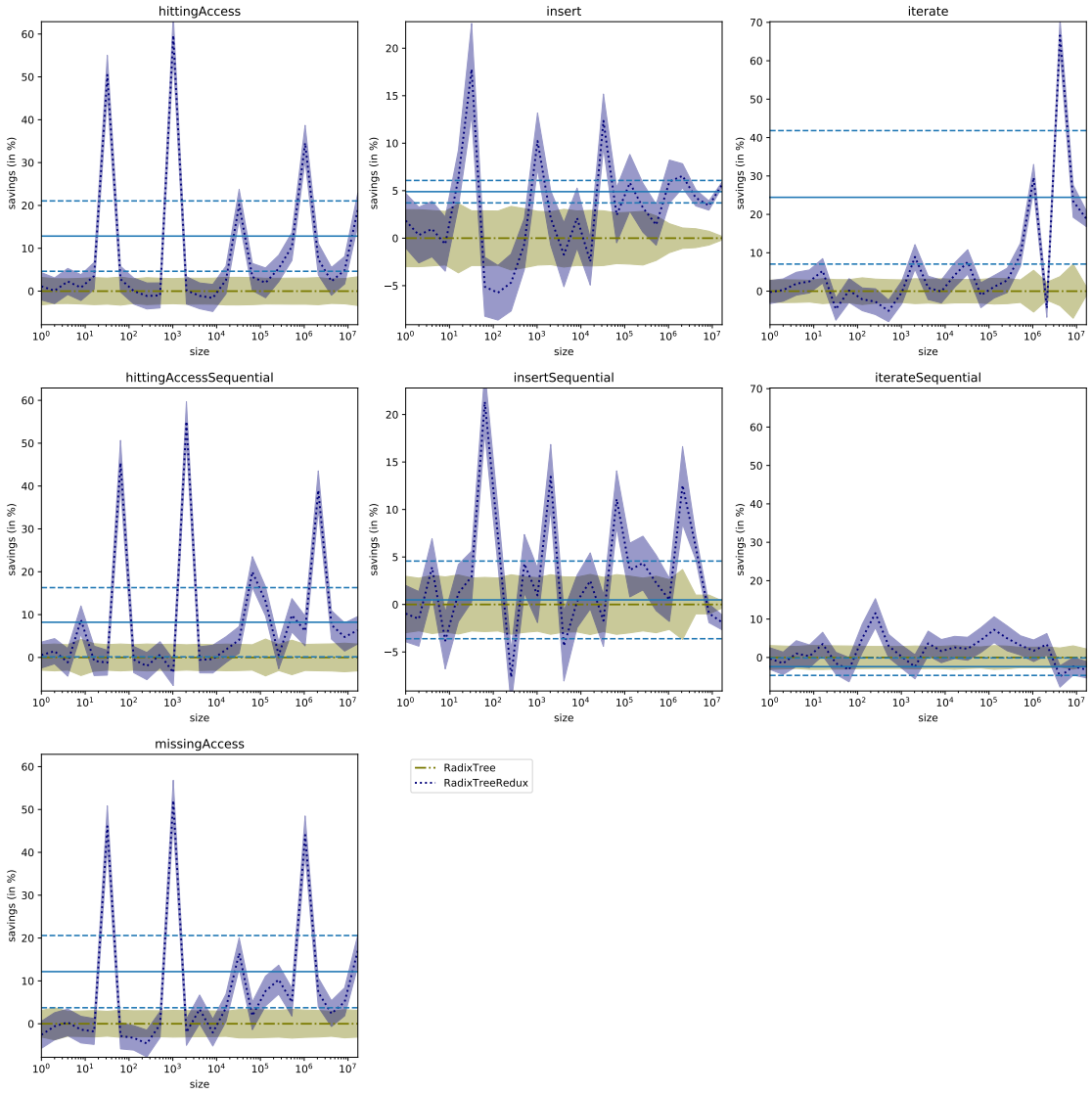


Figure A.1: Visualisation of JMH benchmark results for `RadixTree` and `RadixTreeRedux`. The density parameter is set to 0.5. On the vertical axis is operations per second normalized against `RadixTree`. On the horizontal axis is the size parameter in logarithmic scale. The margin of error with confidence level 99.9% is also visualised. The blue vertical lines are sample means, weighted by how many elements were added between two consecutive sizes, of the normalized operations-per-second values for the `RadixTreeRedux`. The blue dashed vertical lines indicate similarly weighted sample deviation.

A.2 Comparing radix tree implementations with more precise benchmarks

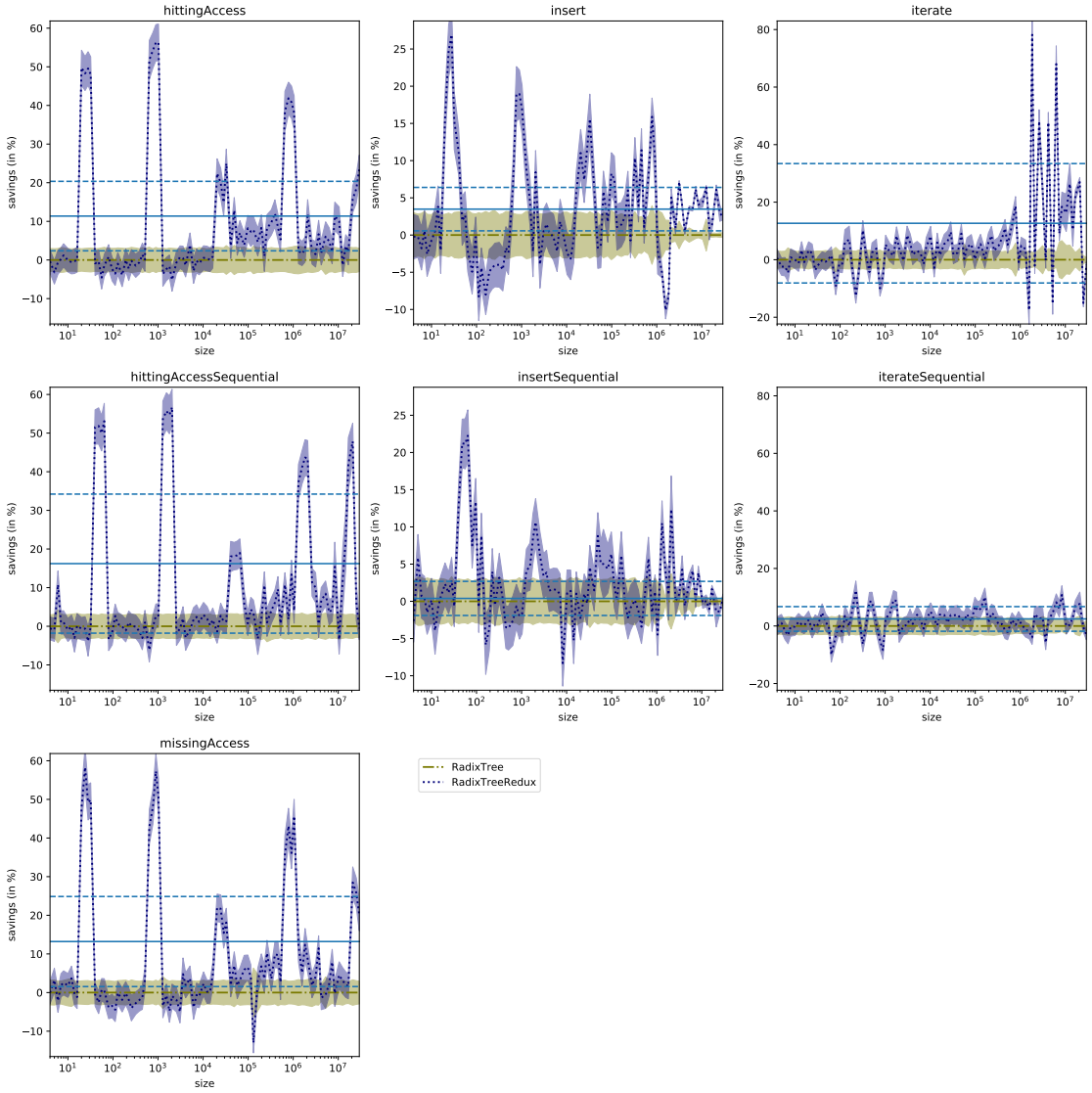


Figure A.2: Visualisation of JMH benchmark results for `RadixTree` and `RadixTreeRedux`. The density parameter is set to 0.5. On the vertical axis is operations per second normalized against `RadixTree`. On the horizontal axis is the size parameter in logarithmic scale. The margin of error with confidence level 99.9% is also visualised. The size parameter is more precise for this graph: $\{2^n + k\frac{2^n}{4} : n \in [1, 23], k \in [0, 3]\}$. The blue vertical lines are sample means, weighted by how many elements were added between two consecutive sizes, of the normalized operations-per-second values for the `RadixTreeRedux`. The blue dashed vertical lines indicate similarly weighted sample deviation.

A.3 Comparing Scala's int maps

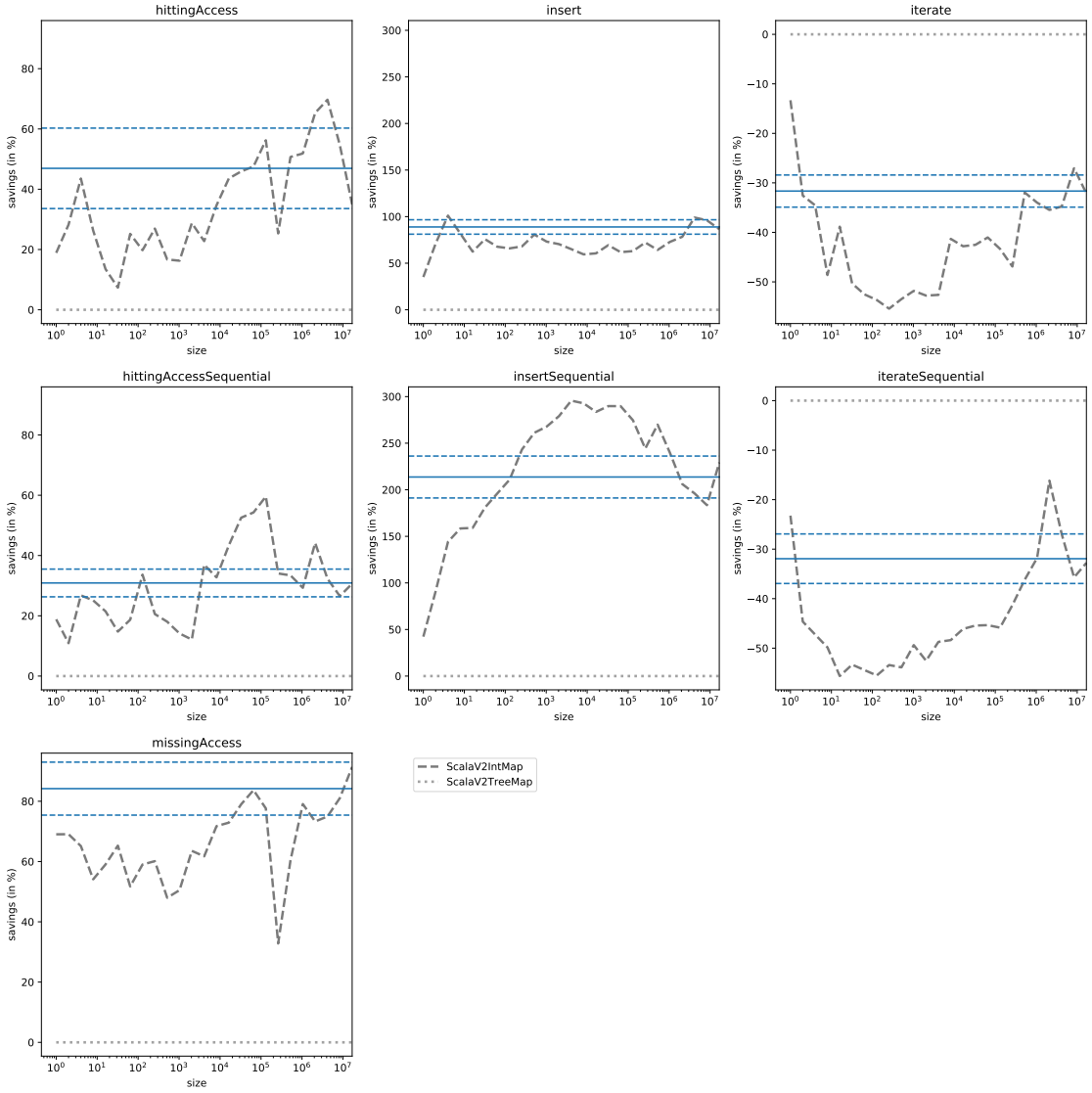


Figure A.3: Visualisation of JMH benchmark results for `ScalaV2IntMap` and `ScalaV2TreeMap`. The density parameter is set to 0.5. On the vertical axis is operations per second normalized against `ScalaV2TreeMap`. On the horizontal axis is the size parameter in logarithmic scale. The blue vertical lines are sample means, weighted by how many elements were added between two consecutive sizes, of the normalized operations-per-second values for the `ScalaV2IntMap`. The blue dashed vertical lines indicate similarly weighted sample deviation.

A.4 Comparing Scala's int maps without logarithmic scale

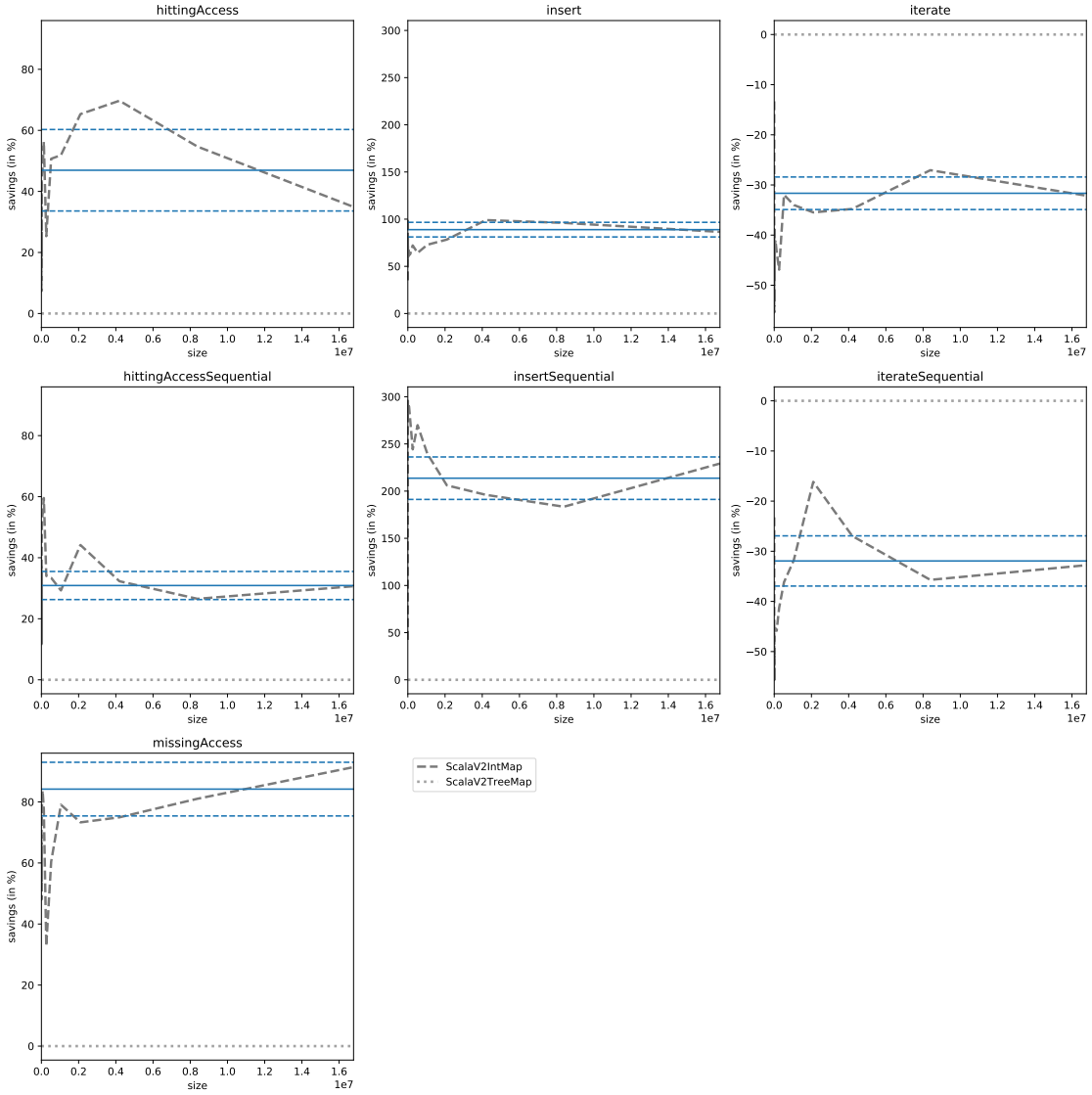


Figure A.4: Visualisation of JMH benchmark results for `ScalaV2IntMap` and `ScalaV2TreeMap`. The density parameter is set to 0.5. On the vertical axis is operations per second normalized against `ScalaV2TreeMap`. On the horizontal axis is the size parameter without logarithmic scale. The blue vertical lines are sample means, weighted by how many elements were added between two consecutive sizes, of the normalized operations-per-second values for the `ScalaV2IntMap`. The blue dashed vertical lines indicate similarly weighted sample deviation.

A.5 Comparing persistence overhead of data structures filled with numbers $[0, n]$

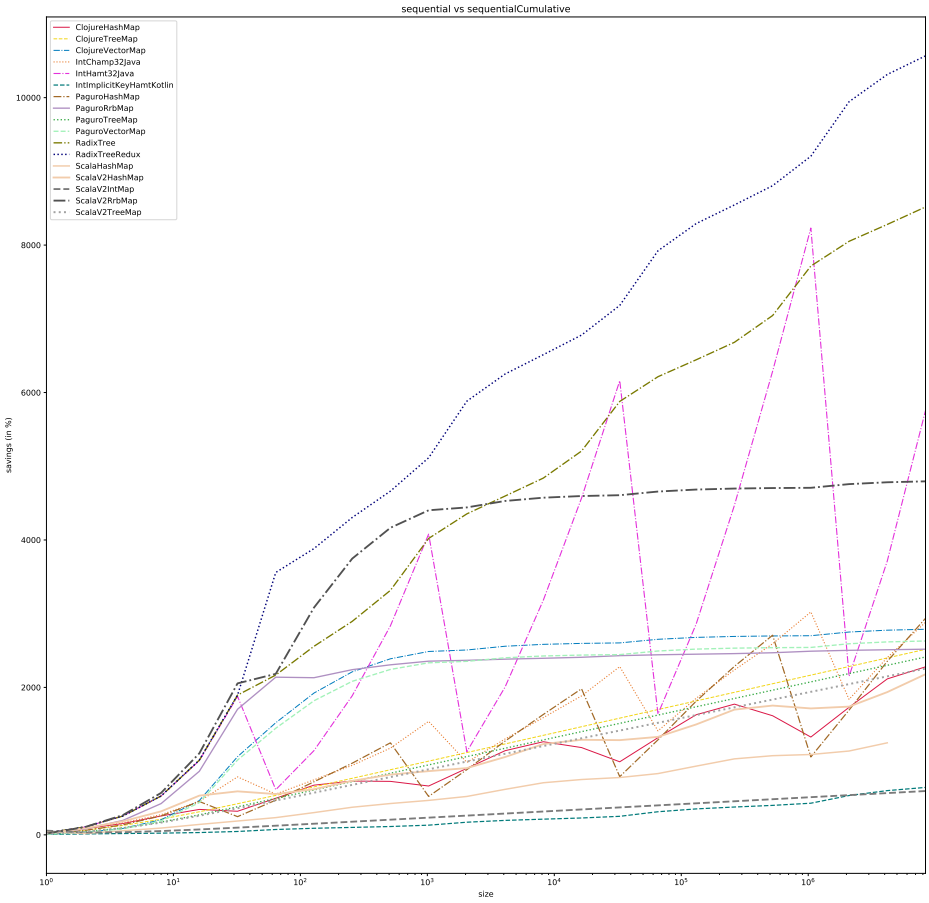


Figure A.5: Visualisation of JOL benchmark result off for the `sequential` benchmark compared against the `sequentialCumulative` benchmark. `ArrayMap` and `SdkMap` were excluded as they were outliers. Some of the data structures whose values were close to another were also excluded for clarity. On the vertical axis is the memory usage of each data structure in the `sequentialCumulative` benchmark normalized against its memory usage in the `sequential` benchmark. On the horizontal axis is the size parameter in logarithmic scale. The closer values are to 1, the smaller the overhead caused by persistence.

A.6 Comparing effects of branching factor for implementation of HAMT made for this thesis

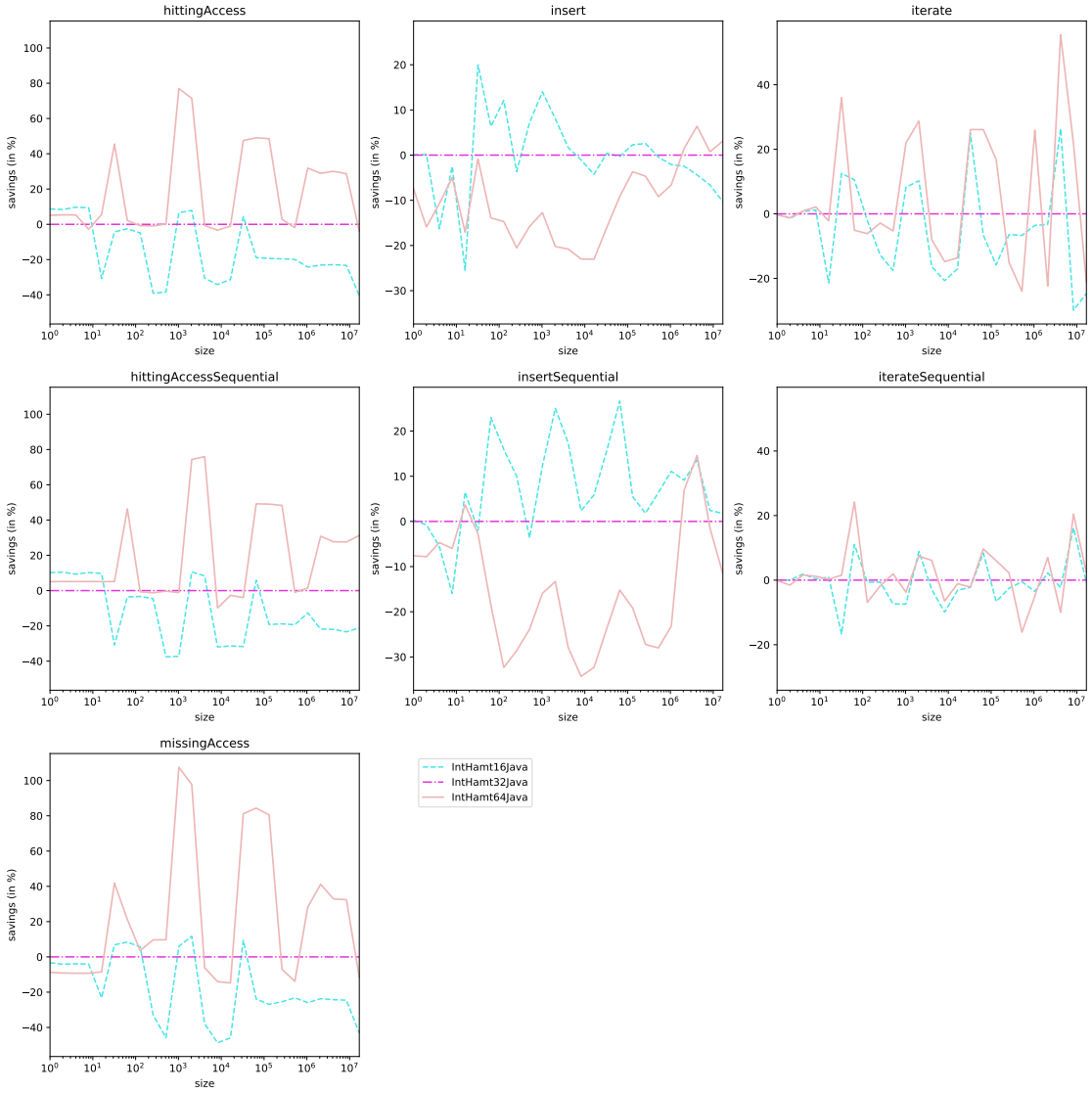


Figure A.6: Visualisation of JMH benchmark results for integer keyed HAMT implemented as part of this thesis with different branching factors. The density parameter is set to 0.5. On the vertical axis is operations per second for each data structure normalized against `IntHamt32Java`. On the horizontal axis is the size parameter in logarithmic scale. The results for this graph were run on a different server, so cannot directly be compared against other results in the thesis.