Extending Alan-DB

Designing a DBMS for storing and retrieving typed, rooted graph-based data on disk

K. van Loon

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

EXTENDING ALAN-DB

DESIGNING A DBMS FOR STORING AND RETRIEVING TYPED, ROOTED GRAPH-BASED DATA ON DISK

by

K. van Loon

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,

Supervisor:	Dr. ir. J. Hidders,	TU Delft
Thesis committee:	Prof. dr. G. Houben,	TU Delft
	Dr. ir. A. L. Varbanescu,	TU Delft
	Ir. C. Schraverus,	M-industries

TU Delft Delft University of Technology

Contents

1

INTRODUCTION

Since the NoSQL revolution in the database world, many new types of databases have been developed, solving various various weak points of traditional relational databases. One of the things all these databases have in common is that they try to make it easier for the developer to deal with data structures that do not fit the table like paradigm of relational databases. In some NoSQL databases this means ignoring structure all together (key-value stores), and in others this means allowing for more flexibility in data structures (graph databases, document stores).

However, an application always has some kind of data model. In databases with schema flexibility, data migrations are still needed. The data model in these databases is often defined in application code - sometimes multiple times in various code bases. Adjustments to the model also has to be translated to the database through migrations. It might be that this data model is too complex to be expressed in tabular form, or expressing the data model in tabular form might result in a very inefficient relational schema, but to allow for schema flexibility is trouble waiting to happen.

In the end, many developers and businesses are looking for a fast system for large volumes of complex connected data that is robust. M-industries partially solved this with their own technology. They have developed a language in which complex data models can be expressed and they have built a database to efficiently store data in this form in memory. The obvious limitation is the size of data that can be stored which is bound by the amount of available RAM. Most databases persist their data to the file-system or even distribute them among several machines to handle large volumes of data.

This thesis explores what techniques are known to store large volumes of data on disk, and experiments whether they can be of any value to the database build by M-industries.

1.1. M-INDUSTRIES

M-industries[1] is a software company that provides software solutions for large industries. However, unlike regular software companies, M-industries work method revolves around their own modeling languages that they have been developing the last fifteen years. This family of languages can describe data models and business processes of clients, as well as their own data transfer protocols, migrations, imports, transformations and many other internal data structures. Based on the application model written in Alan-Application, a database with a schema and a graphical user interface is generated in which one can manage the client's master data.

Since their current projects all revolve around industrial settings, additional interfaces are defined used for touch screens, lamp towers (for signals) and other devices to coordinate factory machines and processes. These processes can become quite complicated due to many exceptions and special cases, and have to conform to pleasant work methods for the factory employees. Once finished, the solution is able to share very complex data in real-time with various devices and machines.

Another interesting part is that Alan allows them to reduce the gap between business and software by having a common language for their consultants and clients to model in. This, combined with the fact that a compilation of the model instantly creates a working application, makes fast iterations and quick delivery

[1] http://m-industries.com/

possible. Similar projects done by other companies tend to last for years and end up less feature complete, whereas M-industries can do a better job and deliver within months.

1.2. PROBLEM DESCRIPTION

Problem Statement The data models created with Alan form a typed directed rooted graph (TDRG). One of the key features of the database is that it can handle optional data containing subgraphs (for more details see Chapter 2). E.g. consider a user input form with a radio button group of several options, where each option has a different set of associated input fields. Only the input fields corresponding to the selected radio button are theoretically needed to be stored once the user submits the form. Document stores or graph databases can store this kind of flexible data better than relational databases, but lack a schema to describe this. The database created by M-industries, called Alan-DB, already allows for efficient in-memory storage, but this limits its use to projects that only need to store several gigabytes of data. Being able to store on disk allows for a significant increase in data storage. There doesn't exist a database design for storing effectively and efficiently a TDRG with conditional data on disk.

Research Questions

1. What are the possible storage structures for storing and processing a TRDG on disk and what are their features?

2. Which of these storage structures are best suited for the workloads of a typical application at M-industries?

3. How well does the solution perform compared to other types of databases?

Approach There are many different types of databases with different storage structures, strategies, and goals. With reasoning alone, we might be able to determine for some of these strategies whether they are applicable or not for the database of M-industries. The applicability of other strategies might be more ambiguous. Since speed performance, memory usage and disk space usage all influence the quality of the solution, we need a framework in which various implementations of different storage strategies and structures can be compared in order to determine what works and what not. From this position, we can improve the solution in an exploratory manner with real data backing up our decisions. As part of the work of this thesis, the existing database shall be adjusted in order to create a DBMS that can easily compile with different storage techniques. Furthermore, as described in Section 6.1, we develop a benchmark language that is used for describing data initialization and queries that can, among other things, randomly select entries within collections. This language is created using the Alan platform and is used to test the various Alan-DB executables, with different implemented storage techniques, on different machines.

For developing the various implementations, we will look into the internals of existing databases, both relational and NoSQL databases, and reason whether their storage methods are applicable.

Contributions The contributions of this thesis are the following:

- A classification that differentiates between types of database data. This classification helps us to understand what needs to be stored on disk and what should be kept in memory.

- An empirical study on the performance of various software-based paging algorithms. Paging algorithms are used to effectively swap parts of the entire dataset from disk to memory and vice versa.

- Insights into the performance of a final version of a persistent schema graph database.

- A benchmark framework and language for benchmarking Alan-DB.

An adoption of the benchmark language designed in thesis might be interesting especially for the graph database world in order to create common and thorough benchmarks.

1.3. THESIS OUTLINE

In Section 2 we will discuss the relevant techniques and languages used by M-industries as well as give insight on the current work load. Section 3 describes the related work on the internals of existing database systems and benchmarking. Section 4 then incorporates insights from Section 3 and adapt these to design the data layout and the global architecture. Section 5 specifies the different implementations that are going to be compared during the experiments. In Section 6 we present the benchmark language and the benchmarks that were described with it used for the experiments. Section 7 presents the experimental results and analyses them. The final Section concludes this thesis and gives suggestions for future work.

2

Background Knowledge

The platform of M-industries is called Alan (named after Alan Turing) and includes a tool set to write languages. From these languages C++ APIs can be generated that can be used for creating algorithms on instances of those languages. Furthermore, serializers can be generated to serialize and deserialize instances of this language into binary or JSON. This family of languages is an integral part of the platform itself, describing web socket communication protocols, client projects, or even how to write more languages.

To differentiate between their technologies, in this paper M-industries technologies are prefixed with "Alan-". The relevant technologies used in this paper are:

- **Alan-Application** This is the modeling *language* in which client applications are described.
- **Alan-DB** The *database* that stores the data described by Alan-Application.
- **Alan-Instructions** An instruction *language* for the database. Instructions such as data queries and mutations.
- **Alan-Benchmark** A benchmark *language* for Alan-DB specifically created for this thesis (See Section 6.1).

2.1. Alan-DB

Alan-DB is a model-centric, transactional, typed, and rooted hierarchical database, currently only working on a single machine in memory. The database process itself (i.e. with an empty dataset) in idle state consumes around 40 MB. Even though Alan has an instruction language for the database, this is generally not used. Applications send instructions in JSON format, serialized from javascript objects, through a websocket connection that gets deserialized into a C++ instruction object, which then is executed by the database. Responses follow this same process backwards.

One interesting feature is their push-based subscription system in which a client can subscribe to some node and receive the modifications of the graph under that node through notifications. Thus, whenever a mutation is processed, the database needs to check what subscriptions are affected and it sends out the new data through the websocket connections. This eliminates the need for client applications to refresh whole datasets every once in a while in order to keep up to date.

2.2. Alan-Application

Alan-Application is used to define an *application model*. The application model starts out with a root node definition containing property definitions which in turn can contain further node definitions depending on the specific type of property. The resulting model has a tree shape with cross-connections from references thus forming a rooted directed graph. Node definitions are anonymous and defined by brackets containing property definitions. Property definitions are defined by an identifier in single quotes, followed by an arrow, the property type name and further property type specific details.

More specifically, properties can be of type:

- **Number** Defines a number-field (64-bit). By design, numbers cannot be floating point numbers and clients should decide how precise their data has to be saved (e.g. in millimeters or meters). Example:

```
'Price (eurocents)' -> number
```

- **Text** Defines a text-field. Example:

```
'Name' -> text
```

- **Group** A container for a Node definition. Example:

```
'Configuration' -> group (
    'Base Value' -> number
    'Employees' -> dictionary ( )
)
```

- **Dictionary** A key-value list of which the keys are strings and the values are Nodes. Requires a further Node definition. Example:

```
'Users' -> dictionary (
    'Email' -> text
    'Password' -> text
)
```

- **Reference** A reference to a Node under some entry of either a Dictionary or a Matrix. References require a path expression within parenthesis. Example:

```
'Favorite Soup' -> reference ( . 'Soups' )
```

- **Matrix** A Dictionary but with References, instead of string values, as keys. As with references, it requires a path expression within parenthesis and as with dictionaries, it further requires a Node definition. Matrices could be seen as a sub-set selection with metadata. Example:

```
'Friends' -> matrix ( . 'People' ) (
    'Since' -> number
)
```

- **Stategroup** A Stategroup has several states defined where each state contains a Node definition. These should be considered choices of which only one can be selected at a time. Example:

```
'Color' -> stategroup (
    'Red' -> ( )
    'White' -> ( )
    'Blue' -> ( )
)
```

In reality, the Alan-Application language allows for many more constructs and constraints, and the language evolves rather quickly because their platform allows them to do so. An example is a construct called "derived data" such as a summation of number properties, or a derived stategroup that has its state set based on the states of other stategroups. Another example is a construct that makes data change on specific time intervals. However, these other constructs are mostly used for business logic and often an extension to the basic constructs described above. We scope this thesis down to these basic constructs.

Path expressions, used for matrices and references, are relative and start from the node definition they reside in. The first part of the expression travels up, and the second part travels down. Every step starts with a symbol signifying the type of the property followed by either a symbol to indicate a parent step or a property name (within single quotes) to indicate a child step. The symbols corresponding to the property types are displayed in table 2.1.

Table 2.1: Path Expressions Symbols

parent step	^
Dictionary	.
Group	+
Matrix	%
Stategroup	?
State	*

An example of an application model written in Alan-Application can be seen below:

```
root (
    'Orders' -> dictionary (
        'Quantity' -> number
        'Status' -> stategroup (
            'Planned' -> (
                'Scheduled Start Date' -> number
            )
            'Started' -> (
                'Phase' -> number
            )
            'Finished' -> (
                'Result' -> stategroup (
                    'Success' -> ( )
                    'Failure' -> (
                        'Reason' -> text
                    )
                )
            )
        )
    )
    'Company Property' -> group (
        'Tools' -> dictionary ( )
        'Tool Lists' -> dictionary (
            'Tools' -> matrix ( .^ . 'Tools' ) ( )
        )
        'Machines' -> dictionary (
            'Costs per Hour' -> number
            'Current Process' -> group (
                'Order' -> reference ( +^ .^ +^ . 'Orders' )
            )
            'Required Tools' -> reference ( .^ . 'Tool Lists' )
        )
    )
)
```

In the simplified example above, the model describes a list of orders which can be in different states, and a group of company property types including machines on which those orders can be executed. As can be seen, both the different states of a stategroup and the path expressions for references and matrices are defined between brackets. The empty brackets represent empty node definitions.

2.2.1. DATA MODELING COMPARISON

Mainly because of the unique Stategroup construct in Alan, data modeling is a little different then in other modeling paradigms. Simple booleans and enumerations with details can be modeled as Stategroups. More interesting however, is that Stategroups can be used as a substitute for inheritance. Consider the following scenario. A company has an abstract notion of a product which can be either a music album or a book. In the future the company might be willing to extend their line with new kinds of products.

In relational databases, this scenario would often be modeled using structures that reflect inheritance and foreign keys, but the end result would be denormalized to increase performance. The following pseudo-code

describes such a relational model:

```
PERSON
* ID
* name

PRODUCT
* ID
* price
* description

BOOK
* product_id
* author_id
* title
* publication_date

ALBUM
* product_id
* artist_id
* title

TRACK
* ID
* album_id
* name
* duration
```

This would result in several tables that are related through foreign keys. A denormalized version would merge `BOOK` with `PRODUCT` into one table and `ALBUM` with `PRODUCT` into another.

In document or key-value stores, one might create a collection for products and exploit the possibility of flexibility by storing a "type" field which is used by application code to interpret the flexible "details" field. The example below illustrates this approach in pseudo-JSON:

```
PRODUCTS: [
{
    type: "BOOK"
    ID: 1234
    price: 12,34
    description: "..."
    details: {
        author: "#5"
        title: "Some Title"
        publication_date: 20080101
    }
}, {
    type: "ALBUM"
    ID: 1235
    price: 23,45
    description: "..."
    details: {
        artist: "#9"
        title: "Another Title"
        track_list: [ ... ]
    }
}]
```

In Alan modeling, conceptually one would follow this document approach but the modeler has the possibility to define a concrete schema using Stategroups to deal with sub-types:

```
'products' -> dictionary (
    'price' -> number
    'description' -> text
    'type' -> stategroup (
        'book' -> (
            'author' -> reference ( ... )
            'title' -> text
            'publication date' -> number
        )
        'album' -> (
            'artist' -> reference ( ... )
            'title' -> text
            'track list' -> dictionary (
                'name' -> text
                'duration' -> number
            )
        )
    )
)
```

Stategroups allow for more flexibility in schema definition by making it possible to define exceptions and different states. In instance data described by Alan-Application, null-values are not allowed. It should be explicitly modeled that a value could be empty using a stategroup. This approach is very similar to Haskell's Maybe monad[1] and helps to increase the robustness of the system by forcing developers to handle edge cases.

2.3. ALAN-INSTRUCTIONS

Alan-DB can handle several types of instructions.

Collection queries Collection queries travel the tree to some context node. They can iterate through (sub-) collections and filter on text, numbers, or states within stategroups.

Mutations A mutation travels to a specific node and can update any properties in the underlying tree of that node. This including adding/updating/removing entries in collections.

Commands Commands are predefined in the schema. They often are simple mutations such as a number increment or a stategroup switch. They often represent user interaction with client applications.

Subscriptions A client application can subscribe to a node. The database sends notifications with changed data (after mutations/commands) if it affects the subscribed node.

Other instructions are getting the database time, authenticating, unsubscribing, data dumping and importing. This thesis will not further go into these instructions since they do not affect the data layer, are infrequent and take only a small amount of time, with the exception of data dumping and importing. From now on, the term 'instructions' only refer to the 4 above.

Below is an example of a mutation that sets the "Current Order":

```
update + 'Current Process' (
    'Current Order' -> reference "order01232"
    'Is Planned' -> stategroup set to 'yes' (
        'Is Started' -> stategroup set to 'no' ( )
    )
)
```

[1] https://wiki.haskell.org/Maybe

One can distinguish between 3 kinds of path expressions used in the instructions. All of these instructions start at some node as defined by the node path which is a path expression starting from the root. Then for collection queries a further collection path describes a tree of sub-collections (e.g. all 'Tools' used by all 'Machines'). And finally for both mutations and collection queries, several property paths can describe the properties that should be respectively modified or retrieved. Table 2.2 describes the allowed property steps each type of path can make. Note: a property path ends with a leaf, which are shown in the leaf column, and a collection path ends with either a dictionary or a matrix step.

Table 2.2: Property Step

property type	node path	collection path	property path	leaf
dictionary	✓*	✓**	x	x
group	✓	✓	✓	x
matrix	✓*	✓**	x	x
number	x	x	x	✓
reference	✓	✓	x	x
stategroup	✓*	✓*	✓*	✓
text	x	x	x	✓

* Must be accompanied by a key referring to an entry or a state.
** Collection paths must end with a dictionary or a matrix step

2.4. WORKLOAD

In this section we analyze the current workload of the existing database at M-industries. We will look into occupation of the processor and the instructions fired at the database. The insights gained on how the database is currently used by M-industries, can be used later on to determine the preferable storage strategies.

For a typical industrial solution there are a semi-fixed number of clients (e.g. touch screen operators installed in factory) connected to the database through subscriptions, that send commands to mutate data either due to user interaction or machine pulses. The subscriptions are tied to a node and whenever some data under that node gets modified, the database sends notifications to all affected subscriptions.

Every instruction fired at the database is serialized and stored as a JSON file. Figure 2.1 describes two days of these stored instructions of a project in a staging environment. The y-axis describes the time it took for an instruction already arrived at the server to go to the datastore through a websocket and come back to the server as a reply. In other words, the time for execution of an instruction plus the overhead of transfer time through the websocket. The x-axis describes the amount of seconds, after the first instruction, for the instruction to arrive at the server.

As can be seen, the great majority of instructions (above 95%) are interface command executions centered around 40 microseconds. These are simple data mutation actions predefined in the application model sent by client applications. In general these client applications are touch screen operators installed in a factory hall used by employees to manage machines, resources and orders. Many of these commands are simply machine pulses whilst others come from employees pushing on touch screen buttons to start operations. When one would look at the application model of this project, one can see that the commands are not more than an increment of a number or a switch to another state of a state group (with a simple node initialization). Therefore, it's no big surprise that most of these commands take little processing time. The monotone consumed interface points at the bottom represent time notifications, and the higher ones represent data imports. Both collection queries and mutations come from users that search and adjust data through the master data management interface. One can see there is way more variance in the duration of these instructions.

The processor utilization of another project running in production has been analyzed in order to give insight into how much extra processing utilization a typical project could handle. Figure 2.2 shows the processor utilization in percentages for one day. Of the instructions processed that day, around half were collection queries and the other half were mutations or commands.

In total, the processor worked for about 4.7% of the time that day. Since most of the time the processor is in an idle state, the figure suggests that the processing time of instructions can be increased a number of times with the current hardware without significantly increasing the time the instructions have to wait in the queue. The fat stripe at around 2000 seconds represents a data import and the execution of subsequent instructions that where waiting in the queue for the data import to finish.

Figure 2.1: Instruction Protocol Duration

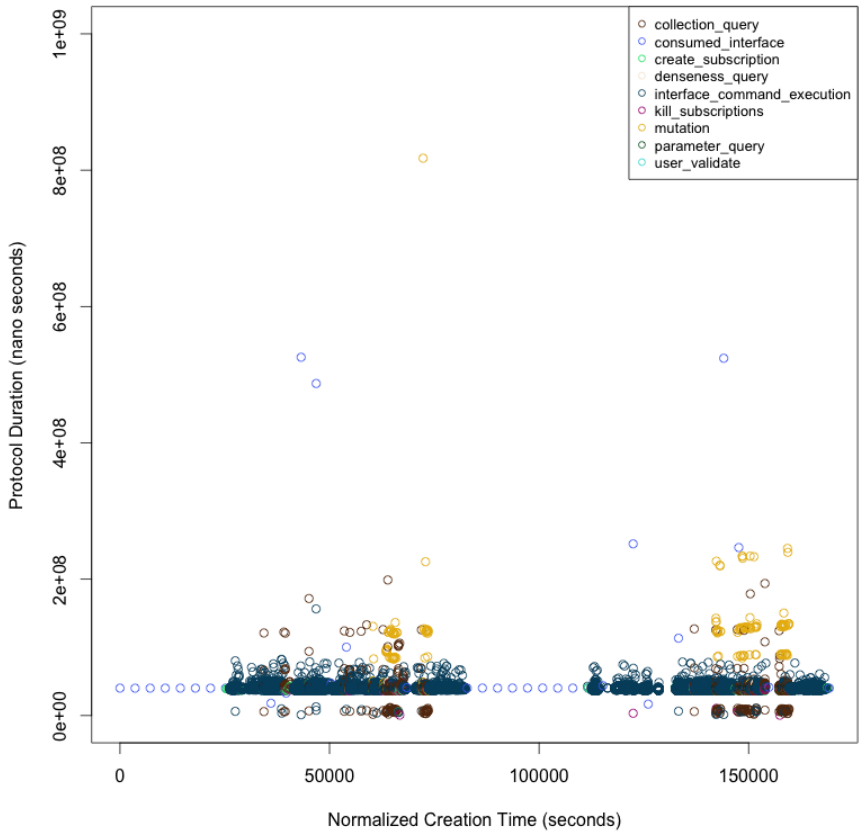
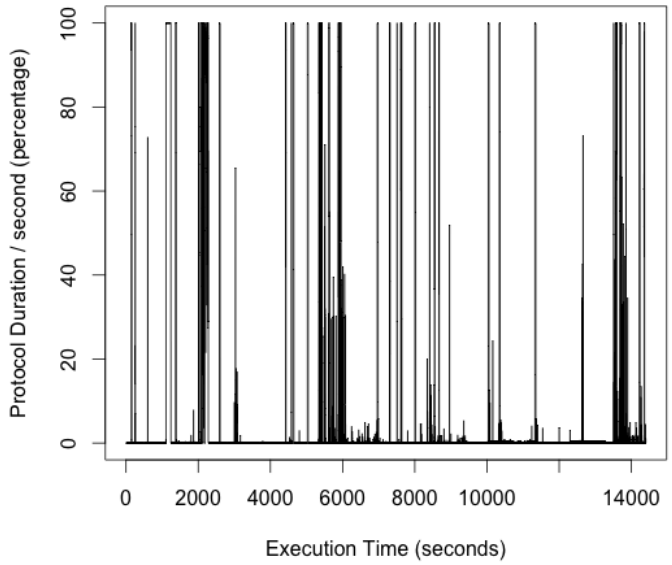


Figure 2.2: Processor utilization



2.5. TYPICAL MODEL

Although the development on the Alan-Application language is a continuous process, by analyzing existing data models of the projects of M-industries, we can gain insight on what to expect on the form of data of a typical project. We can later use this analysis to develop our own representative data model for benchmarking purposes. Table 2.3 sums up the property-type counts of three major projects, anonymized as A, P, and O as well as the maximum number of collection ancestors.

Table 2.3: Property usage in schemas

	A	**P**	**O**
dictionary	22	92	97
group	3	32	13
matrix	3	30	25
number	46	207	60
reference	23	144	118
stategroup	54	173	124
text	33	119	67
max. # collection ancestors	3	3	4

Based on the analysis of the models of three major projects, we can say that the maximum number of nested collection definitions is rather limited. Most top level collections have at least one sub-collection (second level), and only a very few second level collections have further sub-collections.

M-industries adheres to the idea that in modeling, one should strive to capture structure as much as possible. Therefore, a modeler should limit the usage of strings that need further interpretation, but prefer stategroups, references and numbers over text fields. For example, consider a factory employee encountering an error in the operation of a machine. Instead of modeling an error text field in which the employee can describe the error, a stategroup might be preferred where the user can select an error type. The adherence of this idea can be seen in the numbers of table 2.3 as well. Project 'O' is the oldest of the three and relatively has more text fields than the other two.

Another conclusion that can be derived from the principle of capturing structure as much as possible, is that although in theory the database should be able to support text of an unbound size, in practice, one might expect modelers to divide data that could be huge text fields into several smaller fields capturing more structure and lowering the maximum string length. For example, instead of storing a text document as a big Markdown or an HTML file in a text property, a modeler would model a structure with dictionaries containing paragraphs, chapters, links etc. Figure 2.3 supports this assumption. This figure is a histogram of all the unique strings of Project O, both dictionary keys and text properties. One can see the string length not exceeding 60.

2.6. CONCLUSION

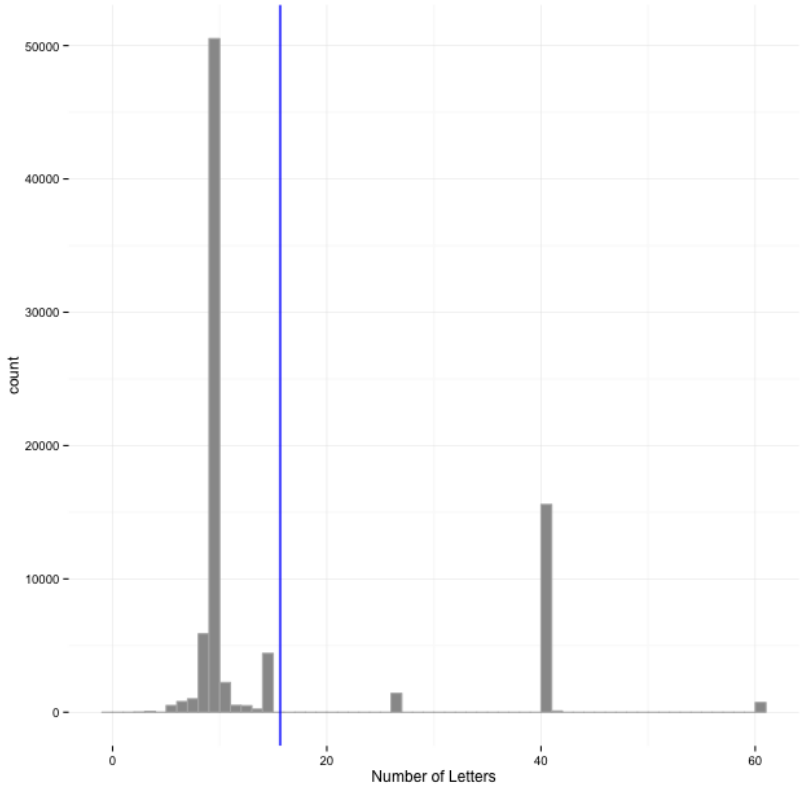
This thesis deals with a type of database that has a strict schema with constructs that allow for complex data structures. These data structures include references, sub-selections of collections (i.e. matrices), and optional data (i.e. stategroups). Both the database, the modeling language and the query language (or instruction language) are unique. Applications of this database are in closed environments (e.g. within a company) and are write intensive due to the constant updating of operational processes by machine pulses and employees.

Because of the hierarchical nature of the schemas and the cross connections through references and matrices, instructions handled by the database often need to traverse several nodes. Furthermore, both data retrieval and data mutations mostly concerns projections of data as opposed to whole documents or aggregates. In other words, the vast majority of read and write instructions are traversal heavy and only affect a subset of properties within a node.

From the analysis of database activity from one of M-industries clients we have seen that more than 95% of the instructions take only around 40 microseconds to process. The analysis of the server processor utilization of a live project shows that the utilization is quite low and thus suggests that there is room for extra instruction processing time without creating significant delays.

The analysis of application models created by M-industries show that their models try to capture structure as much as possible by having nested collections, many stategroups, and references. Finally, the analysis of a dataset dump shows that the strings that are stored are not that big.

Figure 2.3: String length distribution



3

Related Work

This thesis addresses a problem that is fairly unique. First of all, Alan-DB itself is a unique database type. This forces this research to compare for the related work the features of Alan-DB with the features of other databases. Secondly, the well known database types started out with an on-disk implementation, and only later got an in-memory variant. This thesis researches the opposite process of which to our knowledge no related work exists.

Therefore, this chapter first looks into disk-based storage techniques of other types of databases in general. Next, we look into related work on benchmarking.

3.1. Storage Techniques

In order to find ways to best store the data on disk defined by the Alan-Application language, we can look to other databases and determine whether their storage techniques are relevant or not. Many different types of databases might be interesting for various reasons. For example, relational databases can be interesting because they also rely on a schema, graph databases can be interesting for their efficient handling of connected data, and document or XML stores because of the hierarchical nature of the data they store. In the following sections we will look into relational databases, MongoDB, Neo4J, XML stores and OrientDB to analyze their storage techniques.

3.1.1. Files and Pages in RDBMSs

Relational databases have existed for a long time and many variations have been developed, including single-file and in-memory databases. However, most relational databases serve from several files. Records (i.e. table rows), indexes, schema definition, and meta data all is stored in files.

We summarize here the main storage techniques for RDBMSs as they are for example described in [1].

Serving from files provides the possibility to serve an amount of data that exceeds main memory capacity. Therefore, databases must assume that files cannot be fully read into memory. To cope with large files, files are conceptually split into logical units called pages (or sometimes: blocks). Pages are of fixed size so they can be easily managed in-memory by a page buffer manager. Whenever data from a specific page is requested and there is sufficient space inside the buffer, the page is read from the file and put in the buffer. Otherwise, some other page has to be written back to the file to make space for the requested file. A specific page replacement strategy determines what page should be replaced. In most databases, the buffer manager chooses the page that has been used the least lately, however other strategies do exist.

Both HDD and SSD devices read and write in blocks of some minimum size: this is the size required even if a program desires to read or write an amount of bytes below that size, the hardware reads or writes that minimum size anyway. HDD devices typically do IO in blocks of 512 bytes and SSD devices in blocks of 4 or 8 kibibytes, however there exist SSD devices that do IO in blocks of even 32 kibibytes. SSD devices make use of flash memory and are perfect for random access, but HDD devices use magnetic disks for storage that need to spin to the right location for the data to be read or written. Because of this spinning on HDD devices, their sequential access speed is more efficient than their random access speed. So for HDD devices, doing IO in blocks of a small multiplication of 512 bytes takes about the same amount of time. Relational databases nowadays often default to a page size of 8 kibibytes because that's great for most devices.

In relational databases, records can be either of fixed size length or of various length depending on how they are defined in the schema. Fixed size length records are aligned on pages such that no record is split onto two pages (assuming the record size is smaller than a page) but which potentially creates some unused space for each page.

Records with variable length fields are stored in a way called *slotted-page structure*. Here, entries are stored continuously starting from the back of the page. Additionally, the pages contain a header consisting of the number of entries, an array of pointers to the actual entries within that page and a pointer to the end of the free space. Global pointers to entries actually point to the array of pointers inside the header. This layer of indirection is useful for when the entries inside a page need to be reorganized after a deletion or an update of a record.

Relational databases often also have support for storing records that are bigger than a page, or even binary files such as images or pdf documents. These large character and binary objects (clobs and blobs) are stored in a different file and the records that should contain them store just a pointer towards the large object.

Data files themselves can have their data organized in several ways including heap-based, sequentially, and hash-based. In heap-based organization data is placed wherever there is space, which is useful when the order of data isn't important but insertion speed is. Data that is often required sequentially can be organized as such by adding pointers to records to their next record. Furthermore, these records should be stored close to each other in order to reduce page accesses.

When some kind of filtering is required on this sequential data, indexing can be used to reduce the order of retrieval time from linear to logarithmic with regard to the total size of the dataset to be filtered. A binary tree is constructed in a separate file for this purpose that maps a certain field (e.g. a primary key, or a name-field) to the real record or the page it resides on. Another approach to indexing is through a hash table. Hash tables have the benefit that they require less space and can retrieve data based on a search key almost immediately through a hashing of that search key. However, binary trees used for indexing are ordered and therefore are well suited for retrieving ordered subsets of data.

3.1.2. BSON AND MMAP IN MONGODB

This section summarizes the main storage techniques used in MongoDB as described in [2].

MongoDb stores their JSON documents in BSON[1] (i.e. Binary JSON) format. The idea of BSON is to convert JSON documents to a format that is more efficient to navigate through. Structural characters ('{', ':', '"', etc.) are replaced by bytes signifying the type (string, object, number, or list) and, in case of data other then numbers, the length of the upcoming data. Furthermore, numbers are converted to a fixed sized length (4-byte or 8-byte). Using this format, the database engine has no need for a full deserialization of a document, since it can directly scan the document and jump over parts it's not interested in. However, this means searches are still needed to find certain properties meaning that in worst case the entire document needs to be traveled to find a certain property.

In MongoDB The BSON data is spread over several files with preallocated ranges for each collection. This means that a collection could be spread over several files and that data from several collections can reside in a single file. The reason for this approach is as follows. A technique called memory-mapping (MMAP) is used to map the data within the files into memory. Similar to a paging buffer, as described in the previous section, MMAP swaps pages under the hood to limit memory usage but leaves the responsibility to the operating system. Machines with a 32-bit architecture often cannot handle files bigger than 2GB. MongoDB supports these machines to minimize the amount of memory-maps needed (i.e. one for each file), new allocations of space for a collection are directly appended to the existing space allocations and chopped in files of 2GB. Further allocations grow exponentially.

3.1.3. INDEX-FREE ADJACENCY AND FILE ORGANIZATION IN NEO4J

Graph databases have been developed when the need arose to better handle heavily connected data. Traditional SQL databases, as well as NoSQL aggregate stores such as document or column stores, need searches and table joins to connect data. Especially queries that have to travel several connections tend to explode in terms of memory consumption and execution time due to the intermediate and huge tables that have to be constructed in these kind of stores.

In order to overcome this problem, instead of storing foreign keys that connect records, most graph databases store direct pointers that connect nodes with relations. Relations in graph databases are stored

[1] http://bsonspec.org/spec.html

as separate objects containing direct pointers to two nodes. Nodes store a pointer to a linked list of relations. This construction allows the database to travel relations without look ups in a global index table and is called index-free adjacency[3].

In a typical indexing system, records are found by their indexed property in $O(log(n))$ time, where n is the size of the collection. When records are connected through indexed foreign keys, like in relational databases, traversing m records thus at least costs $O(m \cdot log(n))$. To find reverse relations from record A to record B, the system has to scan a whole table or collection for B to find a foreign key that's the same as the ID of A. This will costs $O(n)$ time for each look up, so traversing m reverse relations costs $O(m \cdot n)$ time. Index-free adjacency bypasses searches and allows for direct links making these traversals, including reverse traversals, possible in $O(m)$ time. Furthermore, no intermediate tables need to be constructed resulting in less memory consumption.

Neo4j is a graph database that stores nodes, relations, properties, and labels. There is a file for each of these forms where the data gets stored in and additional files exist for storing indexes, empty spaces and other metadata. Because these different forms of data are of fixed sized length, data retrieval and empty space management is easy and efficient. For example, nodes are all 11 bytes long and consist of various pointers to a list of relations and properties. These pointers are simply numbers that can be multiplied by the length of the respective data type in order to find its position directly within the file, e.g. node 20 is placed at position $20 \cdot 11 = 220$. This differs greatly from SQL storages where indexes form a binary tree that have to be searched through in order to find the specific page the row data is stored on.

3.1.4. HIERARCHICAL STORAGE AND STRING SEPARATION IN XML STORES

XML by its nature is quite verbose and only loosely structured. This makes efficient storing of XML not trivial. To deal with the verbosity, XML Stores apply techniques to compress the Document Object Model (DOM) structure of XML fragments.

In the case of Sedna[4], a loose schema is extracted and updated from inserted XML fragments. This schema takes the form of a tree of element and attribute types and text leaves where each tree-node points to a linked list of corresponding instance data blocks. These instance data blocks, in the case of elements and attributes, contain direct pointers to adjacent instance data. Instance data is retrieved and modified through XPath expressions following the schema tree and the direct pointers of instance data. In contrast to relational databases and MongoDB, the text values are stored separately in another file using the slotted-page structure as described in Section 3.1.1 to maintain fixed sizes of structural instance data.

In contrast to grouping data of the same type, such as all elements or attributes of the same name, some XML storages designs opt for grouping data of whole XML documents together. Where Sedna is an example of the former, also known as schema-based storage, Xindice[5] and MarkLogic[6] are of the latter, subtree-based storage. In Xindice XML documents represent a single item (e.g. a car or an insurance file) and documents with the same tag name are stored in the same file. These documents are stored in pages that are organized as a B-Tree through pointers in the headers of the pages. To save space, the DOM of the XML documents is compressed by replacing tag-names with pointers which can be dereferenced through a symbol table. To query data, whole documents, which can be spread among several pages when they are too big for one, are read in memory and parsed as DOM before an answer can be constructed. Even worse, when the user does not specify in his query what documents Xindice should search, it might brute force through all documents for an answer. MarkLogic doesn't have this problem due to its extensive indexing. It does, however, store many additional files and directories, creating one directory for each collection and a subdirectory for each document containing separate files for tree data, index data, access frequencies, and data quality. To deal with the enormous amount of files and directories MarkLogic generates, besides extensively buffering documents, they distribute processing power among the server cluster.

3.1.5. ORIENTDB

The database that probably is most closely related to Alan-DB is OrientDB[2]. As a database that can handle graph as well as document structures and for which a (partial) schema can be defined for further optimization and validation, OrientDB is a great candidate for comparison. OrientDB has been demonstrated by Dayarathna to be a very competitive database with great performance[7].

The following is a summary of the internal workings of OrientDB as presented in their online documentation[8].

OrientDB uses clusters to group data of a certain type which can conceptually be related to collections

[2] http://orientdb.com/

in document stores or tables in relational databases. Every cluster has two files: one for the data itself in a BSON-like format, and one file to map the cluster position to the physical positions of the data. In addition, OrientDB stores index files for properties that have been indexed.

As with relational databases, OrientDB also uses a pagination system with data aligned to pages, meaning that if the size of an entity does not exceed the page size, it is never spread among several pages. In previous versions, data was loaded from files into memory using memory mapping, but starting from 2.0, OrientDB uses its own pagination system due to the lack of control memory map brings. The cluster IDs of records consist of a number pointing to a page and a number pointing to a relative position within that page. The new IDs resulting of insertions do not fill empty space of previous deletions. Instead, to cope with a growing size of unused space, the user should manually do a database export and import.

The main idea of page caching in OrientDB is to have several cache levels where pages that are more accessed get moved to a more prominent place. The page cache of OrientDB actually consists of two caches, one read cache and one write cache, which contain several queues. The read cache is based on the 2Q page cache algorithm[9]. When a page is not in the read cache, it enters a queue in the read cache dedicated for newly read pages. After a certain threshold for reads, this page then is moved to another queue in the read cache for pages that are accessed a lot called the 'hot queue'. When a page already was in the hot queue it is put at the front of the queue. The write cache is an implementation of the WOW algorithm[10]. This cache works in a separate thread and has pages moved there from the read cache whenever data within that page gets modified. Whenever enough memory is available, the write cache determines what pages it should flush to disk. Pages are sorted and written in groups of four to reduce random access and thus improve write performance.

3.2. BENCHMARKING

On the subject of database benchmarking, researchers often focus more on either micro benchmarking or application-wide benchmarking. In a micro-benchmark, a single instruction is fired several times at a database, often on a simplified dataset, to discover the performance of the instructions. In an application-wide benchmark, a full work load of instructions, sometimes from multiple processes to simulate a multi-user environment, is executed on a dataset that mimics a real world scenario. Here, often the throughput is measured (i.e. the amount of instructions handled by the database within a specific time interval). The approach of the benchmarking done in this research was mainly focussed on micro benchmarking. However, regarding data generation, our approach has more in common with application-wide benchmarking.

In application-wide benchmarking a lot of effort goes to researching a domain to extract a certain representative dataset and workload. For instance, Armstrong et al. analysed current social graphs and query workload on these graphs in order to create a graph and workload generator[11]. Their data model for the generator consists of vertices and edges with arbitrary attribute data. Even though the user of their benchmark platform can tweak the generator through some parameters, in the end the resulting vertices and edges are still quite abstract and arbitrary. In contrast, the generation process of a dataset in the work of this thesis, follows a schema and thus can generate meaningful, representative datatsets. The difference between other generators that generate meaningful datasets, such as Datagen[12], is that within the platform created for this thesis, the user is not bound to a specific set of datasets or schemas. He can easily create any schema and has fine control over the initialization of a dataset. This allows for both use case specific benchmarking and extensive micro benchmarking.

Some graph generators, such as the one created by Erling et al.[13], have features that are not available in the platform created for this thesis. Their generator can apply more sophisticated functions for the distribution of relations to create clusters within a graph. Their approach allows for mimicking various degrees of popularity of people within a social network. If one would construct a social network with the current version of our generator, every person would have the same amount of friends. To extend the benchmark language and its implementation to support new features such as these, however, requires little effort.

The benchmarking itself of this thesis is very similar to the approach of Angles et al.[14]. The similarty lies in generating datasets instead of having a prefabricated dataset and measuring the performance of each instruction instead of measuring the throughput of a workload. However, Angles et al. generate all datasets from one very simple model and only benchmark 12 instructions (all reads).

In the analysis done by Fundulaki for LDBC, they conclude that existing graph database benchmarks often are too simple in their dataset or in their queries[15]. For instance, Dayarathna[7] was able to develop a good system with XGDBench, but lacked a decent set of instructions. The problem many benchmark systems have

is the lack of a possibility to extend or improve their benchmark instructions and datasets once others find it insufficient. The realization that more extensive micro benchmarking is needed and some kind of flexibility for extending the benchmark set is essential, is shared by Afanasiev et al.[16]. In their work, they create a platform for micro benchmarking XQuery where users can submit their own queries to extend the benchmark repository. Although the realization is similar, the solution is different. The flexibility in the work of this thesis is realized with a benchmark language.

3.3. CONCLUSION

This thesis deals with a process that is very uncommon: starting from an in-memory database and moving to an on-disk version. Related work on this process is either hard to find or doesn't exist. With regards to benchmarking however, our work can be related with the work of other research. Even though the creation of a benchmark language is unique, micro-benchmarking and data generation are done before. Furthermore, others have attempted to create a platform to fulfill the need for extensive micro-benchmarking

From the study of various kinds of on-disk databases we have concluded the following. To deal with large amounts of data that do not fit memory, databases synchronize data with the file system through files. The layout of data in the files varies with different databases, but they all have a certain strategy in common: paging. With paging, the database system divides data in to pages of a certain size and only keeps a limited amount in memory through one or several page caches.

For the data layout inside files, broadly two types can be distinguished. The first type can be seen as entity dumps where the data of the entities are of variable sizes and are serialized, and usually compressed, onto pages. Relational databases, aggregate stores (key-value, document, and column), and some XML stores use this type of data layout. To deal with changing sizes over time, these entries are stored on pages with an empty buffer and indirect pointers in order to reorganize entities within the page without breaking references. The other type of data layout, used by graph databases and some XML stores, is that with a focus on direct pointers and fixed sized entries. The downside of the second approach is that data from one aggregate always gets spread over several pages, but the benefit is that traversing entries becomes more efficient. The previous chapter showed that Alan-DB is a write-intensive traversal-heavy database, thus suggesting the second approach is more appropriate.

4

DESIGN

This chapter presents the main design of a disk-based Alan-DB. First the main architecture is presented, containing some parts from M-industries and other parts were designed as part of the work of this thesis. Next we go deeper in the storage layer, argue what needs to be stored and how, and present the data layout of the files.

4.1. ARCHITECTURE

M-industries constantly works on their platform and as a result, they currently have two versions of the transaction and data layer. The first version runs in production, but actually has no clear separation between the data and the transaction layer. The second version was a total redesign and does make a clear separation between the layers. Even thought the second version has not fully implemented all the features of Alan-Application, the performance in general is significantly better both in terms of the duration of instruction handling and as memory usage. Furthermore, the seperation of the two layers in the second version allowed for a disk-based implementation, whereas with the old version, this was not realistic. Therefore, the design and work of this thesis was based on the newer version.

The architecture of Alan-DB consists of four layers, each build on top of the next:

1. **Instruction layer** Gets instruction-requests. Builds replies with the requested data for collection queries. Every instruction that mutates data (Mutations and Commands) is converted to a transaction. Multiple instructions cannot be merged into one transaction. As a result, commands and mutations return a success/failure response. Furthermore, this layer is also responsible for triggering notifications to be send to various clients. Part of the work in this thesis was to port the instruction layer to the new data layer. Due to the restrictions of the new data layer, commands were not fully ported and notifications were not ported at all.

2. **Transaction layer** Builds and executes transactions for mutating data and rolls back the transaction if any data constrains are invalidated. A transaction in Alan-DB can only contain one instruction. The transaction layer is of the new version and was provided by M-industries.

3. **Data layer** The data layer is responsible for getting and setting data on an elementary level. An in-memory implementation of this layer was provided by M-industries and a disk-based implementation was part of the work in this thesis.

4. **Storage layer** The storage layer manages file I/O. This layer was fully written as part of the work in this thesis.

Since the instruction layer and the transaction layer were given by M-industries and are out of scope of this thesis we continue with the data layer.

4.1.1. DATA LAYER

The data layer API consist of declarations for the elementary data structures being Dictionary, DictionaryEntry, Group, Matrix, MatrixEntry, Node, Number, Reference, Stategroup, State, and Text, and various functions

to travel through and modify these structures. The data layer is called by both the instruction layer (in case of query instructions) and the transaction layer. The API described in table 4.1 was provided by M-industries.

Table 4.1: Data Layer API

function	parameters	returns	description
get_node	(entry | state | group | dataset)	node	Returns the node associated with the entry, state, group, or dataset
get_parent	(property, property_definition)	node	Returns the node containing this property
get_property*	(node, property_definition)	property	Returns the property of the node as defined by the definition
to_parent_entry	(node)	entry	Returns the entry associated with a node
to_parent_state	(node)	state	Returns the state associated with a node
find_entry**	(collection, key)	entry	Returns the entry of a dictionary or matrix with this key
get_collection_iterator**	(collection)	iterator	Returns an iterator of a dictionary or matrix
assign_property_value*	(property, value)		Sets a number, text, state, etc. to the property
create_entry**	(collection, key, entry_definition)	entry	Allocates a bare unattached entry
destroy_entry**	(entry, entry_definition)		Removes entry, including its contained node, from memory / disk
create_state	(stategroup, state_definition)	state	Allocates an unattached state
destroy_state	(state, state_definition)		Removes a state, including its contained node, from memory / disk
emplace_entry**	(collection, entry)		Attaches an entry to a collection
erase_entry**	(collection, key)		Detaches an entry from a collection
create_dataset	(dataset_definition)	dataset	Initiates a dataset and allocates the root node
destroy_dataset	(dataset)		Removes all dataset data from memory / disk

* In the real API there is a function for each property. E.g. `get_text`, `assign_number_value` etc.
** In the real API there is a function for both matrices and dictionaries.

The previous version of the data layer, which was an in-memory implementation, contains classes for each Alan-Application construct (Node, Number, etc.) that contained the database data. Furthermore, these constructs had methods defined that allow for mutations and traversal (e.g. to get a child property of a node). The new data layer as presented in table 4.1 uses functions, each with a Alan-Application construct parameter, instead of methods. The Alan-Application constructs in this API are only declared, not defined, and thus allow for multiple implementations.

For the creation of new structures, some additional memory is allocated, fields are (partially) initialized, and the new structure is returned. However, for the on-disk implementations, the structures are defined being empty, making it possible to treat the pointers to them as mere numbers. The real work for the on-disk implementations is then delegated to the storage layer where the numbers representing the structures can be turned into file positions and the relevant associated data is stored in the file on or nearby this position.

4.1.2. STORAGE LAYER

The storage layer, both the API and the implementation, is new and part of the work of this thesis. Similar as in the data layer, the storage layer has a storage structure declared but not defined in order to easily create several implementations. The general idea is that a Storage object should manage a file (or a group of files) given a file name. The storage structure is used by the tables: the `node_table`, `string_table`, `entry_table` and the `hash_table` which are used by the data layer. The different tables get number representations of Alan-Application constructs, calculate the file positions of these representations, and then ask their storage object to get or set values directly on these file positions. The API of the storage layer can be seen in table 4.2.

The `hash_table` is an implementation of Litwin's linear hash table[17] but made persistent (if the storage is implemented like so). Both the `entry_table` and the `string_table` use a `hash_table` for indexing their records. In the case of the `string_table`, the string itself is the key for the `hash_table` and the `hash_table` returns the position of that string inside the `string_table`. For the `entry_table`'s `hash_table`, a unique key is constructed from the numbers representing the collection and the entry key.

4.2. DATA CATEGORIZATION

In order to determine what exactly needs to be stored in files and how, we first are going to distinguish between the different types of database data. This section argues for a differentiation of *schema data*, *instance data*, *optimization data*, and *system persistency data* and discusses their properties.

As we can conclude from the previous chapter, a main difference between schema and schema-less databases is that in schema-less databases the value-data needs to be accompanied with semi-schema data such as property-names and type-labels in order to be meaningful. Because these labels or property names are not well defined and cannot be predicted by a schema, they need to be stored along with the value-data as is the case with BSON and native XML storage. Alan-Application does define a strict schema, so there is no necessity to inline these forms of metadata. Instead, depending on the schema, the system should be able to determine the position of the value-data as is the case in SQL storages.

Schema data, in short, defines how the data is structured, it defines the property names and the type names, and it defines the sizes of the various types. Typical of schema data is that the total size in bytes of this metadata is bounded and small. This means that schema data can be loaded in memory entirely without the worry that it consumes too much space. In the case of Alan, an Alan-Application file can be compiled and loaded into memory and the sizes of the various types can be calculated.

Instance data consists of both the actual value-data such as numbers, strings and selected states, as well as dynamic relation data such as the relation of a node to a stategroup or an entry to a collection. Dynamic relation data can change over time; e.g. the next entry of an entry can change upon a deletion or insertion. Fixed relation data on the other hand such as the relation of a group to a node cannot not change over time. Therefore in theory, fixed relation data should be derivable from schema data. Instance data is the type of data that can grow very large and therefore needs to be stored in files.

Optimization data is data that is not strictly necessary for the system to work, but helps to manage and retrieve data within reasonable time. For example, we can find an entry purely by following and iterating through dynamic relational data of the collection and its entries (i.e. following "next" pointers), but with additional data from indexing, we can reduce the time of finding entries from $O(n)$, where n is the size of the collection, to $O(1)$ time with the use of a hash table. Optimization data thus grows with the instance data and therefore also needs to be stored in files.

System persistency data is data that is needed for the system to initialize its internal state upon restart. This type of data is implementation dependent, but examples are gaps of free space within files, a versioning number, the size of collections, or the iteration state of a clean-up algorithm. Typical for this data is that it is very small and thus can easily be serialized and deserialized from files to memory.

4.3. DATA DESIGN

This section describes the design of the data files and their byte layout categorized in three subsections: strings, nodes, and entry-sets. A guiding principle of the design is to place different kinds of data layouts into different files as opposed to a appending various blocks of data layouts into a single file. Another principle is to separate variable sized data from fixed sized data in order to efficiently handle the calculation of the position of data within a file. These principles were based on the conclusions of the previous chapter for a write- and traversal-heavy database.

Furthermore, since we assume a 64-bit system and the limitation that all data should be able to fit on

Table 4.2: Storage Layer API

function	parameters	returns	description
create_storage	(filename)	storage	Allocates a storage object.
destroy_storage	(storage)		Destroys a storage object.
read_number	(storage, position)	number	Returns a 64-bit integer at `position` of `storage`.
read_bytes	(storage, position, length)	bytes	Returns a byte string at `position` of `storage`.
write_number	(storage, position, number)		Saves a 64-bit number at `position` of `storage`.
write_bytes	(storage, position, byte-pointer, length)		Saves a byte string at `position` of `storage`.
node_table	(storage):		
create	(parent*, property_length)	node*	Allocates space for a node.
remove	(node*, property_length)		Frees space and saves the empty spot.
get_property	(property*)	property*	Returns the value of `property`.
set_property	(property*, value*)		Sets the value of `property`.
string_table	(hash_table, storage):		
optimize	(string)	string*	Returns a number representation of `string`. If it doesn't exist yet, it creates a new entry in the string table, otherwise it increments the reference counter.
get_string	(string*)	string	Returns the real string represented by the number representation.
release	(string*)		Lowers the reference counter and deletes this entry if that becomes 0.
entry_table	(hash_table, storage):		
create	(collection*, key*)	entry*	Allocates space for a new entry.
remove	(entry*)		Frees space and saves the empty spot.
find	(collection*, key*)	entry*	Requests the entry position from the hash table.
get_parent	(entry*)	collection*	Returns the entry's matrix or dictionary property parent.
get_node	(entry*)	node*	Returns the node of this entry.
get_key	(entry*)	string*	Returns the key of this entry.
get_next	(entry*)	entry*	Returns the next entry of the collection.
set_node	(entry*, node*)		Sets the node of the entry.
set_key	(entry*, key*)		Sets the key of entry entry.
hash_table	(storage):		
find	(key)	value*	Returns the value associated with this key.
emplace	(key, value*)		Sets the value for this key.
erase	(key)		Removes this key and its value.

* These parameters or return values are 64-bit numbers that can be converted to file positions.

one machine, we know that both the hard drive and the dataset itself can be no larger than 2^{64} bytes or 16 exbibytes. Therefore, pointers to locations of data in files should be 8 bytes long to guarantee Alan-DB could handle all extreme cases within these assumptions. Finally, we strive to have the execution time of database instructions as deterministic as possible, meaning that we avoid clean up algorithms to unpredictibly execute and delaying these instructions, and to reduce the performance impact of bigger collection sizes to the execution time.

4.3.1. FILE TYPES

We distinguish between three types of data files inspired by the file types used in SQL stores as described in the previous chapter. These are: heap files, index files, and block files.

Heap files simply append new data of various sizes to the file, or fill holes created by deletions whenever possible. To access certain data within a heap file, the system either needs a pointer of a direct position (i.e. an offset) of the data within the file, or a pointer that can be transformed through calculation to achieve the same.

Index files are used by the file-based linear hash table to store 64-bit pointers at file positions derived from the hash function. As is typical for hash tables, at least around 30% of this file would be empty space.

Block files are files that are organized in blocks of data of a fixed size equal to or a multiplication of the operating system page size. These blocks consist of a header, a body with a set of data items, and a footer.

4.3.2. STRINGS

In order to efficiently store and compare strings, Alan-DB makes use of string interning as described in the previous chapter. Another major benefit of string interning is that we can guarantee text properties and entry keys to be of fixed size since they only store a fixed sized pointer to the real string value. As shown in table 4.3, strings are stored by writing the string in a heap file prefixed by its length and a counter that signifies the amount of usages of this string. Since binary blobs in m-industries sometimes are also stored in text values, we settle on a maximum length of 4GB which can be expressed in an 4B integer.

A string-pointer is the direct file position to the length of the stored string. An additional hash-file is used to map strings onto string-pointers. Once a string is fully deleted from the string table (i.e. it has zero references), its position and its length is stored in the empty space manager that returns this position once a new position is needed for the storage of another string with the same length. To increase the efficiency of this empty space manager, the space used for strings actually is rounded to a factor of 8 bytes and so reduces the amount of holes of varying length in the heap file. The manager merges adjacent holes and splits a big hole when space is requested that is at most half of the length of the big hole. This further reduces the total amount of holes of variable length and more importantly deals with very big holes. In real datasets of M-industries, strings are hardly greater than 40 bytes, and so once a user adds a string to the system of 4GB and then deletes it, but never adds a string of 4GB again, it would be an enormous waste of space not to chop this hole in smaller parts.

Table 4.3: String Byte-Design

String		
4 bytes	length	the length of the string
8 bytes	reference counter	the amount of usages of this string
x bytes*	value	the actual string

* the byte size is equal to the length field rounded up to a factor of 8 bytes

4.3.3. NODES

Since data of variable length (strings and entries) are separated from nodes but connected through pointers, all node instances of the same node definition are of the same length. We can pre-calculate property offsets and add it to the in memory schema data and store all the node instance data in a heap file.

Figure 4.1: Node and String example

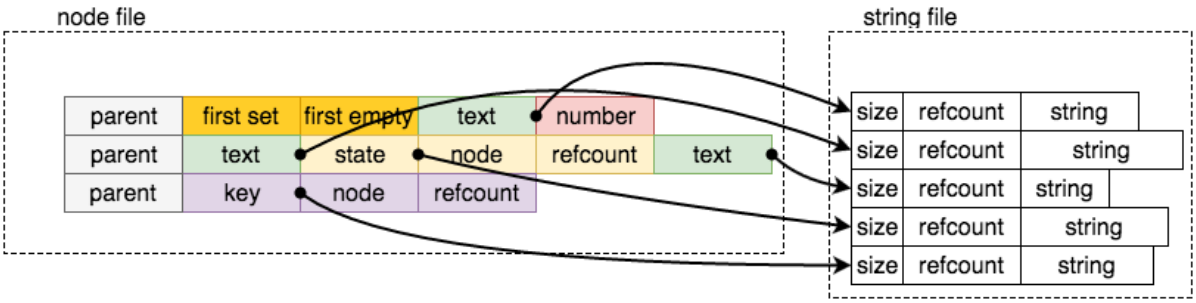


Table 4.4: Node Byte-Design

Node		
8 bytes	parent	pointer to an entry or a state
x bytes	properties	the size is the summation of the size of the properties defined in the schema
Number		
8 bytes	value	a 64-bit integer
Text		
8 bytes	string-pointer	pointer to a string
Group		
0 bytes		the node of a group is inlined
Stategroup		
8 bytes	selected state	string-pointer to the selected state's name
8 bytes	node	node-pointer to the state's node
8 bytes	reference counter	the amount of references to this state
Reference		
8 bytes	key	string-pointer of the referenced entry
8 bytes	node	node-pointer to the reference entry node
8 bytes	reference counter	the amount of references to this state
Dictionary		
8 bytes	first entry-set	a pointer to an entry-set
8 bytes	first empty spot	a pointer to an empty spot within an entry set
Matrix		
8 bytes	first entry-set	a pointer to an entry-set
8 bytes	first empty spot	a pointer to an empty spot within an entry set

Table 4.4 presents the byte layout of a node and figure 4.1 shows a visual representation of the connections between the node and the string table. Every node consists of a pointer to a parent, which can be either an entry or a state (or null in the case of the root node), and a set of properties depending on the type as defined by the schema. References and stategroups keep track of the amount of references to them because a transaction needs to be invalidated whenever it tries to change a stategroup or reference that itself is being referenced to in order to prevent the breaking of references. This means that references and stategroups can only be changed when nothing references to them. Furthermore, the node in which they reside cannot be deleted either. Both matrices and dictionaries keep track of the first entry-set, which is used for iterating the collection and the first empty spot within an entry set.

Another file is used for the serializations of free spot pairs of position and size for the empty spaces. The size of these system persistency data is limited since the number of node definitions and node sizes are limited. Therefore, we can do a simple data load and dump respectively on a program start up and shut down.

4.3.4. ENTRY SETS

Collections in Alan (both matrices and dictionaries) consist of a set of unordered keys mapped to nodes pairs. In the design of storing entries we strive for efficient key-based lookups, iteration, insertion, and deletion. Since in Alan-Application we can define collections in collections, we have no bound on the number of col-

Table 4.5: Entry Set Byte-Design

Page Header		
8 bytes	collection	pointer to a matrix or a dictionary in the node table
Filled Entry*		
8 bytes	key	pointer to a string
8 bytes	node	node-pointer to the state's node
8 bytes	reference counter	the amount of references to this entry
8 bytes	reference**	the referenced node of the matrix
Empty Entry*		
8 bytes	empty	zero-value to indicate this spot is empty
8 bytes	empty	zero-value, not used
8 bytes	next	a pointer to the next empty spot
8 bytes	empty**	zero-value, not used
Page Footer		
8 bytes	next	pointer to the next entry-set or zero if last

* 31 matrix or 42 dictionary entries fit on 1 page of 1KiB
** only for matrix entries

lection instances. Thus, if we would create a file for every collection, we would need to manage open files since operating systems can only have so many files open at the same time (usually 1024).

If we were to store entries in a heap file, we would need to store entries of one collection as a linked list with every entry having a pointer to the collection and to the next entry. Furthermore, entries of the same collection shall over time become spread over the whole file due to the fact that under normal usage, entries from other collections shall be inserted as well, resulting in many fragmented accesses. We also know that since data is retrieved in pages anyway, we do not necessarily need to group all entries of one collection together in the file for optimal iteration access. Therefore, a great solution would be to assign pages to a specific collection. If we know that all entries on a page are of a specific collection, we don't need to store a pointer for every entry to the collection, only one for each page. Furthermore, we don't need to keep a pointer for every entry to the next entry, we can store entries within a page as an array, and have the pages form a linked list. However, if we assign pages to collections with only a few entries, we would waste a lot of space. Therefore, we create smaller blocks for the entries that align to pages. These blocks - with a header for a pointer to the collection, a set of entries, and a footer pointing to the next page - are called entry sets. Table 4.5 describes the byte layout of an entry set for blocks of 1024 bytes.

Empty spots are managed through a linked list starting from a pointer of the collection in the node table. This pointer points to a position within an entry-set that either has a pointer itself to the next empty spot or is the last empty spot itself. Upon insertion the empty spot pointer of the dictionary is swapped with the pointer on the empty spot. Upon deletion the empty spot pointer of the dictionary is placed on the new empty spot, and a pointer to the new empty spot is placed in the dictionary. Thus both insertions and deletions are done in O(1) time. Figure 4.2 shows a visual example of the relations between the node file, the entry-set file, the empty spot list and the entry set list.

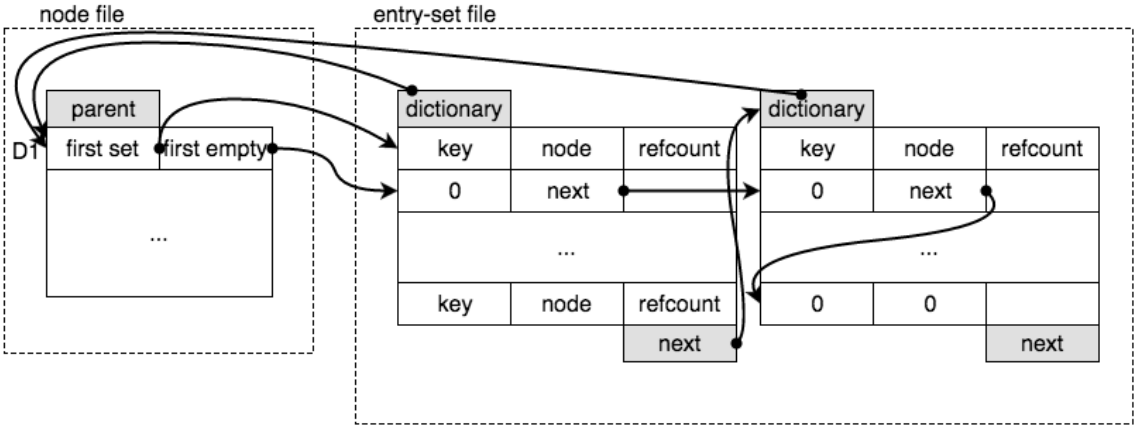
For entry lookups, we add another hash file that maps collection-key pairs to entry positions making lookups possible in O(1) time as well. Additionally, we will also create a heap-file variation for entries for comparison. In this design, all entries store a pointer to its collection and another one to its next entry. For simplicity reasons, dictionary entries also have a 'reference' field although this field isn't used, resulting in a heap file where all entries are 56 bytes long.

In conclusion, through entry sets, empty spot linked lists, and an additional hash table for indexing entries, we should be able to have efficient lookups, iteration, insertion and deletion, efficient storage, and less page requests.

4.4. CONCLUSION

For the disk-based Alan-DB, an additional architectural layer, called the storage layer, was added that manages the mapping between the data layer and file I/O. In order for this to work a different implementation of the data layer API also had to be made. We distinguished between several forms of database data: schema data, instance data, optimization data, and system persistency data. Of these forms, only instance data and

Figure 4.2: Entry-Set example



optimization data need proper management (i.e. more than a simple serialization dump) since only these forms of data can grow out of memory. Furthermore, a data layout was presented for these forms of data with its design decisions based on the conclusions of the previous chapter.

5

IMPLEMENTATION

In this chapter we present the various implementations of the storage layer we compare in the experiments of the next chapter. These implementations only vary in the implementation of the storage layer described in the previous chapter.

5.1. STORAGE LAYER IMPLEMENTATIONS

This research started off with an in-memory implementation of the data layer. Apart from this given implementation, several other disk-based implementations where developed using an exploratory approach, meaning that implementations where tested and measured on small datasets leading to new insights in how to go forward. This set of implementations consists of a naive implementation, several custom paging implementations, an implementation using MMAP, and an implementation in OrientDB used for comparison to the outside world.

5.1.1. NAIVE

The naive implementation uses a file for each storage instance and directly delegates to seek, read and write system calls whenever a get or set method in the storage layer is called.

For example, when the execution needs to know the value of a number property, the number property pointer is cast to a 64-bit integer representing the position of the number property within the file, a 'seek' is called to that position, and 8 bytes immediately are being read from the file.

5.1.2. PAGING

Inspired by other databases and to reduce the amount of I/O operations, a set of implementations using several paging algorithms where created. So instead of one or two system calls for every `get` or `set` method in the storage layer, pages of data are swapped into memory from the file managed by the storage object. The page size depends on the hardware architecture, but for the machines used for this research was 4KB.

There are many different paging algorithms, some very similar to others, so for this research three were selected that were significantly different. These were the NFU (not frequently used), the Clock, and a Random algorithm.

Due to the complexity and the time it takes for implementing, no multi-threaded algorithms were implemented. Work on this was started, but due to discouraging intermediate measurements of a performance decrease of as much as five times caused by of the overhead of multithreading, this work was aborted. Furthermore, the implementation of entry-sets diverted a bit so that all entry-sets are 1kB in size. The original 4kB resulted in too many empty space being wasted.

NFU The NFU keeps an array of pages in memory with the amount of reads and writes on these pages from the time they were first loaded into memory. Whenever a new page is needed, the page with the minimum amount of accesses divided by the time since it was first loaded is replaced by the new page. Finding a page costs $O(n)$ time and replacing a page costs $O(n)$ time as well. The idea however is that pages that are needed a lot do stay in memory which would save expensive read and write system calls.

Clock The Clock algorithm is simply an efficient FIFO algorithm. Pages are buffered in an array of fixed length and the clock hand (a pointer) points to the current 'hour' (i.e. an array slot). A new page loaded into memory replaces the page at the current hour. Then, the current hour itself is incremented or when it would exceed the length of the array, reset to the first hour. This results in a circular buffer, hence the name Clock, with efficient inserts of $O(1)$ time. Finding a page however costs $O(n)$ time, n being the size of the buffer. The main benefit of the Clock algorithm over the NFU is that it guarantees that a page is not being replaced right away as well, but has more efficient inserts.

Random The random algorithm uses an array to cache the pages where the position of a page is determined by a modulo of the length of the array for the page id. For example, if the array is of length 10 and a page with id 512 is requested, the algorithm looks at position $512 \mod 10 = 2$ in the array for that page. If the page is not there, the page will be read and swapped with the page that is currently occupying its position. This algorithm both does finds and inserts in O(1) time, but the arbitrary page swapping might be less optimal.

5.1.3. MMAP

As described in Chapter 2, there is a system call named MMAP on Posix systems that can map a file to memory. The programmer specifies an address range that should be reserved for MMAP and the file MMAP should map to, and all data requested and modified within that range eventually is written to that file. MMAP internally uses paging to lazy load and store data within the addressing range since the addressing range can greatly exceed the RAM limit. This allows the programmer to program as if all data is already in memory without the need to worry about memory usage.

The main benefit of MMAP is that it leaves the work of the paging algorithm to the OS, which might have a very optimal implementation. This can at the same time also be a downside since this makes custom paging strategies impossible. With MMAP, one can not effectively reason about the pages being cached and therefore we can not optimize further if necessary. Furthermore, the necessity to specify an address range beforehand either forces a limit to the maximum dataset, or some kind of MMAP managing that would not be so different from the actual page caching itself.

The MMAP system call in this implementation uses a random access strategy (as opposed to sequential).

M-industries has a preference of keeping control on performance as much as possible. Therefore, the MMAP implementation is only used as a reference and does not constitute for a valid solution.

5.2. CONCLUSION

This chapter described the five implementations of the storage API (described in the previous chapter) used for the experiments in the next chapter. Besides these implementations, another executable for OrientDB was made that hard coded specific benchmarks. We broadly described the different benchmark sets and the approach in developing them.

6

BENCHMARKING

This chapter presents the benchmark language that was created as part of the work of this thesis in order to extensively benchmark the various implementations. Furthermore, the second section describes the benchmark sets that were created with this language and were used in the experiments. This chapter ends with the approach of the custom implementations of several benchmarks in OrientDB.

6.1. BENCHMARK LANGUAGE

In order to test and compare different storage techniques, a benchmark language, from now on referred to as the Alan-Benchmark language, was designed and created within the Alan-framework. With a benchmark language, we can easily define a wide variety of (complex) benchmarks, thoroughly benchmark the data and storage layer implementations, and be transparent on what and how we benchmark. A *benchmark project* consists of an application model written in Alan-Application and a *benchmark definition* written in Alan-Benchmark. That project then can be the input of the various implementations, which include the database and a command for executing benchmarks.

The benchmark language could be seen as a merge and modification of both Alan-Application and Alan-Instruction. A benchmark definition consists of a part that describes how the application model should be initialized and a part that describes what instructions how many times should be fired at the database. The data initialization definition semantically follows the application model and allows the writer to define sizes for collections and values for leaf-property initialization. Furthermore, references can be defined to be generated randomly and leaf-properties can be defined to have a value randomly selected from a set of values at initialization. Finally, stategroups have weights defined to their states indicating the chance of the stategroup being instantiated with that state.

The instructions definition part of the benchmark language first allows the writer to define the amount of repetitions for the execution of the set of instructions. The instructions definition themselves are a modification of the normal instruction language. The writer can define several comma-separated instructions and has access to two additional keywords: *index* and *random*. Index refers to the current iteration of the execution of the instructions (i.e. if the instructions are repeated N times, the index is a number ranging from 0 to $N-1$). In the case of entry or state steps, the random keyword is used to randomly select one of such. In other cases, the random keyword is followed by a number to specify the upper bound; the lower bound being 0. The keys for entries for both dictionaries and matrices are simply a number transformed to a string ranging from 0 to $N-1$ (N being the maximum amount of entries).

Consider the following example application model:

```
root (
    'd1' -> dictionary (
        'n' -> number
        'sg' -> stategroup (
            's1' -> ( )
            's2' -> ( )
        )
    )
```

```
)
```

Listing 6.1: Example Application Model

The example benchmark definition below corresponds with the model above and defines the dataset to be initialized with the dictionary $d1$ with 10000 entries of which every node's property n must be initialized with 1. The stategroup sg, due to its stategroup weights, is expected to be instantiated with $s1$ 7000 times and with $s2$ 3000 times. Then it defines to repeat a mutation instruction for a 1000 times that sets the property n of a random entry to 10.

```
initialize (
    'd1' -> dictionary 10000 (
        'n' -> number 1
        'sg' -> stategroup (
            's1' -> 7 ( )
            's2' -> 3 ( )
        )
    )
)

repeat 1000 (
    update . 'd1' : random ( 'n' -> number 10 )
)
```

Listing 6.2: Example Benchmark Definition

The various implementations described in 5.1 are embedded in their own executables that can run the benchmarks. This executable can run a given benchmark and measures the time it took for each instruction to execute. As a result, the executable prints the quartiles (i.e. minimum, 1st quartile, median, 3rd quartile, maximum) of the durations.

6.2. BENCHMARKS

This section describes the benchmarks, and the motivation for their creation, that were written as part of the work for this thesis.

Data requests (i.e. collection queries and subscription notifications) can in theory easily be distributed horizontally. Transactions (from both mutations and commands) in a server cluster should be handled by the master server and are blocking. Therefore the most important question is whether scaling up to secondary storage can be done with a minimal loss in transactional performance.

The execution and coverage of these benchmarks in general should ideally tell us two things. Firstly, how various implementations compare to each other. And secondly, given a dataset and an instruction, can we now estimate the performance of the instruction?

With the benchmark language it is easy to describe a wide variety of benchmark sets. So in order to guide the creation of the benchmark sets, we intend to cover the various aspects of Alan with the following questions:

1. How do basic mutations perform?
2. How does the size of collections influence instruction duration?
3. How does the number of path steps influence instruction duration?
4. How does filtering affect instruction duration?
5. How do failures and rollbacks perform?
6. How do more complex use case instructions perform?

Based on these seven questions, we develop seven corresponding benchmark sets. The following subsections broadly describe the seven benchmark sets. The full specifications can be found in Appendix B.

6.2.1. B1: MUTATIONS

This benchmark set describes different simple mutations such as adding or removing an entry to a collection, switching a state in a state group, incrementing a number, or setting a reference. The goal of this benchmark set is to find differences within the costs in time of the different simple mutations.

For adding entries, we have one benchmark that adds entries with empty nodes, and one benchmark that adds entries that have nodes with 10 properties in order to see how the size of nodes influence the mutation cost. Furthermore we both benchmark the adding of dictionary entries and matrix entries with empty entries, to determine whether there is any significant difference between addition costs of the two collection types. Finally, for removing entries, we both remove all 1000 entries in one benchmark and remove 1000 entries in another benchmark with a collection size fo 100,000 in order to determine whether the collection size significantly influences this instruction.

6.2.2. B2: COLLECTION SIZES

This benchmark set consists of two subsets: one for mutations and one for queries. All benchmarks in this set have the same application model, but the size of the collection differs. The sizes of the collections range from 10 to 100,000 in multiplication steps of 10. This allows us to see whether instruction time is influenced by the collection size, either linearly or exponentially, or not for both types of instructions.

The introduction of random entry selections in these instructions force the database to touch different parts of the dataset and make the page cache swap more than if the same entry was selected for every run.

6.2.3. B3: DEPTH

The goal of this benchmark is to see how much traveling through subcollections affect instruction duration. We keep the data size relatively constant, but create an array of application models that increase in depth. The benchmarks with less depth have bigger collections in order to keep the data size constant so that depth is the only variable. Using random path steps and a fairly big data size, we try to maximize the work for the paging algorithms.

6.2.4. B4: FILTERING

The benchmark set B2 has provided a way to find out how much collection sizes influence instruction execution time, and this benchmark aims to find the difference in time filtering adds. Number, text and state groups fields can be filtered. This benchmark is designed to answer the question whether the complexity of a filter has any significant performance influence or not. For number filtering we have three benchmarks with collections of sizes 100, 1000 and 10,000 in order to determine whether size has even more influence when filtering. For states and texts we just use a collection size of 1000 and can compare results with the results from B2.

6.2.5. B5: FAILURE

This benchmark set describes mutations that are intended to fail and are very similar to some of the successful mutations in B1. A failure should result in a rollback and we expect their execution time to be not so different from their B1 counterparts. The benchmarks in this set consist of one benchmark that tries to set a stategroup to a state in which it already is, one in which an entry gets added with a key that already exists, one with a removal of an entry that does not exist, and one that tries to travel a path with a state step that doesn't exist.

6.2.6. B6: USE CASE

In this benchmark set, we fill the datastore with an amount of data representable for a production environment. All the benchmarks in this set have the same model and are initialized the same. The model and the instructions were inspired by other models and instructions that where analyzed in 2.4. The goal of this benchmark set is to see how realistic instructions would perform on a representative dataset. One benchmark fires several subsequent filtering collection queries that mimic the instructions fired at the server from a user typing in a search field and getting results while typing. Another benchmark requests some entries of all the sub-collections combined. Then there is a benchmark that imitates a user setting a current task of a machine which includes the setting a stategroup and initiating a node. Another benchmark is a simple collection query of entries with big nodes and the last benchmark is a collection query that travels references and filters on a stategroup to find the machines that process orders of premium clients.

6.2.7. B7: BIG USE CASE

This benchmark is the same as the previous benchmark but differs in size. The sizes of some collections have been increased to create total size of more than 8 GB. Furthermore, due to a technical restriction described in 7.3, we cannot query collections that are too big and therefore only do three of the five benchmarks described in B6: `many_references`, `task_start`, and `search`. More details on this are described in section 7.3.

6.3. ORIENTDB COMPARISON

Since Alan-DB is proprietary and unknown, an OrientDB implementation has been created to compare the performance of Alan-DB with something known to the general public. OrientDB is a hybrid database that can manage both documents and graphs efficiently. Furthermore, in OrientDB one can define a schema that should lead to an increase in performance and the performance of OrientDB in general has proven to be very competitive in comparison to other graph databases[18]. This, combined with the fact that OrientDB is open source, makes it the closest comparison possible.

The implementation in OrientDB is not an implementation of Alan-DB's storage API, but a custom implementation of a subset of benchmarks described in 6.2 and its only purpose is to have a reference for the performance of Alan-DB compared to other databases. Even though transactions and a schema are not required for OrientDB, they are required for Alan-DB, thus for a fair comparison, the OrientDB implementation defines schemas and makes use of transactions. The overall goal is to mimic AlanDB instructions as close as possible.

Since these custom benchmark implementations were very time consuming to develop, only the three benchmarks of B7 where implemented.

OrientDB has three APIs: a document, a graph and an object API. Both the graph and the object API are extensions of the document API and have traded performance for convenience. The document API already supports direct pointers, so since this was the most efficient and bare-bone, this API was chosen.

For the comparative experiment with OrientDB, certain design decisions were made that strove for a fair comparison of the two database types. The Alan-Application language has two constructs that are not available in OrientDB, namely stategroups and matrices. To represent these constructs as close as possible, we decided to model a stategroup as an Class (OrientDB terminology for a node definition) with two properties, a number representing the current state choice, and a link to the node of that state. For example, consider the example in section 2.2.1 where a 'Product' has a stategroup 'type' which can be set to either 'book' or 'album'. The JSON representation in OrientDB of this stategroup set to 'book' would look like the following:

```
...
'type': {
    'state': 0, // The first state is 0, the second 1, etc.
    'node': '#3.14' // A reference to a node of type `Book`
}
...
```

Since the link can be to any node of any type, OrientDB cannot guarantee the validity of a stategroup using a schema alone.

Matrices where modeled as a Class containing two LinkMaps: one for the node-keys and one for the node-values. Consider a matrix of friends of which for each friend the date since the friendship started is saved. The OrientDB JSON representation of this would look like the following:

```
...
'friends': {
    'keys': { 'alice': '#1.11', 'bob': '#1.12' }, // references to nodes of
    type 'Person'
    'nodes': { 'alice': '#2.11', 'bob': '#2.12' } // references to nodes of
    type 'Friend'
}
...
```

Furthermore, in OrientDB using transactions for databases is optional, but since Alan-DB uses transactions, we chose for OrientDB to do so as well. Table 6.1 gives an overview of the property mapping from Alan to OrientDB.

Table 6.1: OrientDB - Alan property mapping

Alan construct	**OrientDB construct**
Node	Class
Number	Long
Text	String
Dictionary	LinkMap
Reference	EmbeddedSet(String key, Link node)
Group	Embedded
Matrix	EmbeddedSet(LinkMap keys, LinkMap values)
Stategroup	EmbeddedSet(Short state, Link node)

6.4. CONCLUSION

This chapter presented the benchmark language where one can describe benchmarks for Alan-DB with. The language consists of a part that defines how a data model should be initialized, and a part that describes what instructions have to get fired at the database. These instruction definitions can include keywords not available in the normal instruction language for randomly selecting entries in a collection, among other things, in order to provide an easy way to simulate random data access.

Furthermore, we described the actual benchmark sets that were created for the experiments done as part of the work of this thesis. These benchmark sets are designed to cover all aspects of the modeling language. Moreover, they include a benchmark set with a model and initialization that would resemble a dataset more close to a dataset in a live production environment.

7

Experiments and Analysis

In this chapter, the performance of the various implementations as described in the previous chapter are compared with each other. The experiments done in this chapter were a result of an exploratory approach. First a range of benchmarks are executed to get an insight into how basic operations on basic datasets perform. The results of these micro benchmarks raise new questions which are answered in the subsequent experiments. In the first section, we run the experiments on a common laptop that has an SSD device. Then we run the same experiments on a server with an HDD device but with more RAM and a faster CPU. After that, we increase the data size of several benchmarks and run the experiments on a server with less RAM than the data size. Finally, we run the benchmarks against the custom OrientDB benchmarks.

All benchmarks where run a 1000 times and the quantiles of the instruction execution times where used to draw the box plots. Note that many, but not all, box plots in the subsequent sections do not show a maximum whisker (a dotted line on top). This means that the maximum is more than 1.5 times the interquartile range (i.e. length of the box). These maxima were caused during the warm up of the benchmark executable (i.e. the first couple of runs of the 1000) and are therefore not representative of a running database.

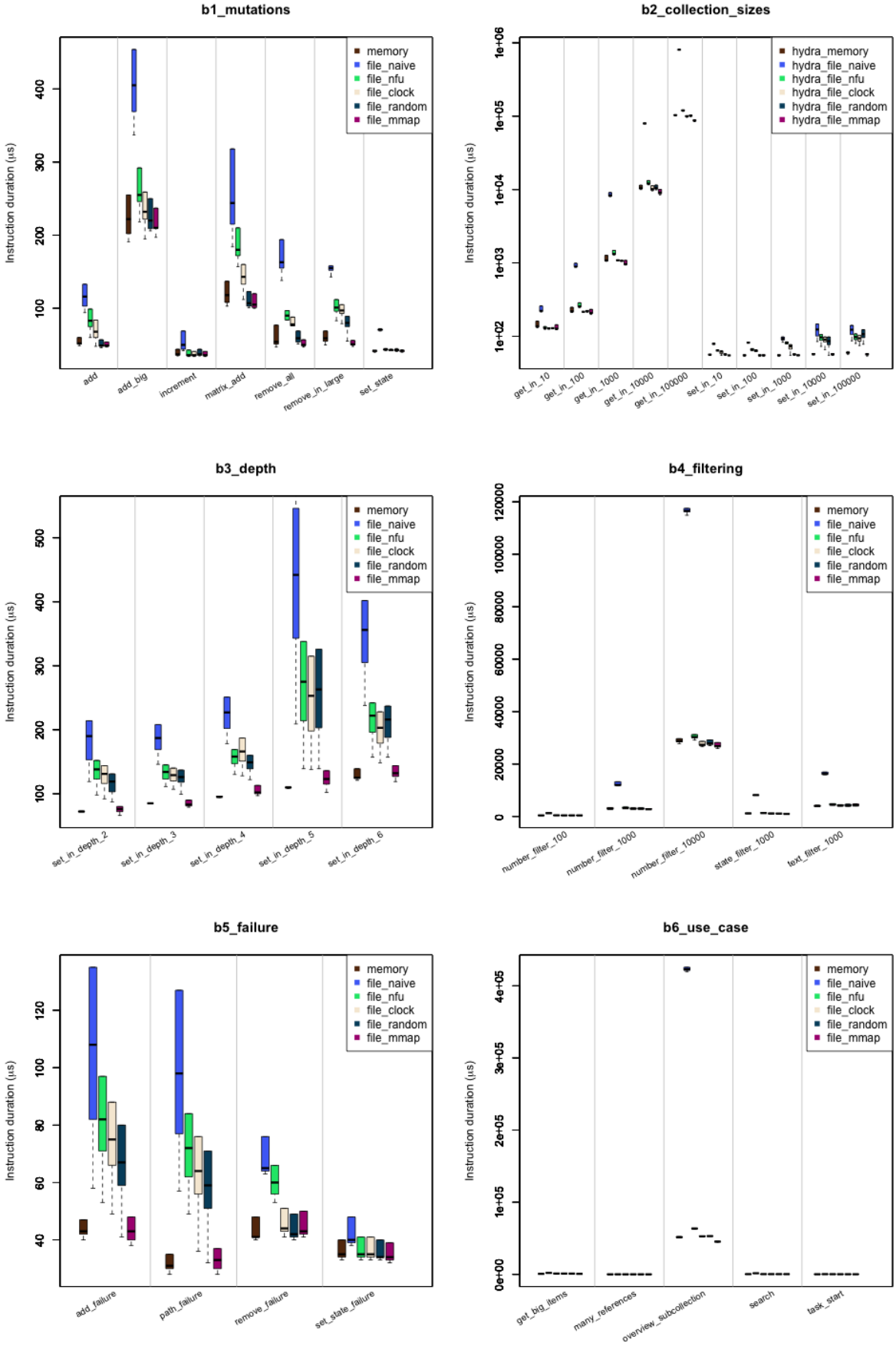
7.1. SSD

With the first range of experiments, we intend to get an insight on the performance differences of the different implementations on elementary instructions and small datasets. These micro benchmarks were run on a MacBook Pro (mid 2014) with the following specifications: four 2.60GHz cores, 8 GB RAM and a write speed of 0.7 GB/s. Note that y-axis of the second benchmark set B2 is in a logarithmic scale.

The first thing to notice is that the benchmarks mutating data (B1, second half of B2, B3, B5, and B6 with the exception of `overview_subcollection`), regardless of the implementation, do not exceed an average instruction duration of 450 microseconds. The only benchmarks that exceed this duration have to iterate through collections: the collection queries of B2, the filtering collection queries of B4 and the aggregation of sub-collections of B6. Secondly, depth traversal, whether through collection steps or through reference steps, do not significantly add to the instruction duration. Processing more properties, however, does significantly add to the instruction duration as can be seen in the add_big benchmark of B1. Thirdly, as expected, the naive implementation always has the worst performance, often two or even three times as slow in comparison to the memory implementation. MMAP performs very similar to the memory implementation, both performing the best and both having a low standard deviation. In addition, on queries over large datasets, MMAP even outperforms the memory implementation. Fourthly, the execution time of collection queries follows the collection sizes. This is a result of the implementation of a collection query execution by the instruction layer. As a design choice, M-industries chose to always iterate through the whole collection, even if the query only asks for a limited number, because the response also needs to contain the collection size. In case of filtering, this collection size cannot be determined beforehand, and therefore as a default all collection queries iterate through whole collections.

What is not shown in these graphs are the differences between tweaks of the page cache sizes for the nfu, the clock, and the random algorithms, but only the best setting after a number of experiments. The results of those experiments were that both the nfu and the clock algorithm performed better when the cache size was very small: 2 pages per file. The random algorithm on the other hand did not perform differently with

Figure 7.1: Benchmark sets b1, ..., b6 on a SSD drive (8GB RAM)



adjustments to the page cache size. The explanation for this is that for each page access by a page id, the other two algorithms needed either a lookup in a binary tree or an iteration of an array to retrieve the right page, whereas the random algorithm could directly get to the right page (if it was present in the cache).

Since traversing deeper in the tree doesn't significantly increase the instruction duration, we can conclude that the implemented index-free adjacency works as intended. Furthermore, there are very little performance differences between the algorithms and storage structures, with the exception of the naive implementation. Moreover, the performance difference between the disk-based implementations and the memory implementation is almost negligible. Since disk I/O should in theory be significantly slower than main memory access, the next experiment looks into the possibility of the SSD causing these results.

7.2. HDD

In the second range of experiments we intend to discover whether there is a difference in performance behavior between HDD and SSD devices for the implementations. These range of experiments were done on a virtual machine on a test server provided by M-industries which has four 3.30GHz cores, 24 GB of RAM, and a HDD write speed of 2.1 GB/s. Figure X below shows the result of the benchmark sets. Note that y-axis of the second benchmark set B2 is in a logarithmic scale.

Compared to the previous benchmarks on a SSD machine, the most significant difference is that the naive implementation performs relatively worse in the SSD experiments. This is to be expected since SSD devices read and write in blocks of at least 4KB whereas HDD devices do this in blocks of 512B. Because most of the disk I/O of the naive implementation are reads and writes of 8B, the relative waste of potential is greater on SSD than on HDD. Furthermore the random and the clock implementations both often perform the same as the memory implementation. Other than that, the results are very similar.

In conclusion, it seems that the MMAP implementation is the best file-based implementation and that it doesn't really matter whether we use an SSD or an HDD on this machine. This experiment shows that the use of an SSD device over an HDD could not be the cause for the unexpected great performance results.

7.3. Paging on a large dataset

In this experiment we look whether there is some form of I/O caching influencing the measurements of the previous experiments. To rule this possibility out, the performance of the file based implementations on a machine with less available RAM and a dataset far exceeding the RAM limit should follow the expectations the previous experiments gave us. So for this experiment, we create a virtual machine on the same server provided by M-industries as in the previous experiment, but with a RAM limit of 2GB. Furthermore, an SSD drive was assigned to this virtual machine in order to allow testing both on SSD and HDD. The collection sizes of the benchmarks were adjusted in order to generate a big enough dataset resulting in a total dataset of 8.7GB.

The implementation of handling collection queries in the instruction layer limits our possibilities for benchmarking these on large datasets. The collection query first loads all entries of all (sub)collections queried in memory, and then iterates over all of these to apply a filter. This means that we cannot query collections that are too large for that will result in the operating system using the swap memory to load everything in memory and thus will take an unreasonable amount of time. Direct filtering during the initial iteration and halting the iteration once the requested limit is reached, would make collection queries on very large collections possible. A current requirement at M-industries however, is that the result of a collection query contains a number of the total amount of entries that satisfy the filter and thus needs a full iteration.

In the figure below, with the y-axis in a logarithmic scale, the brown box plots represents benchmarks done on the 24GB RAM machine and is used as a reference. The benchmarks in blue and green are done on an HDD and an SSD respectively with 2GB of RAM. All of the benchmarks were run with the random implementation.

We can see that the performance can drastically decrease up to more than 10,000 times for HDD and 50 times for SSD once the dataset certainly doesn't fit in memory anymore. This is the case even though all benchmarks were run with an implementation that only uses a page cache of around 1GB. This behavior can be explained due to the fact that the operating system uses spare memory, that is not assigned to processes, for caching files. In fact, this can be demonstrated by running the `top`[1] command during the execution of a benchmark. Therefore, on operating systems that support this implicit caching of files, with a large enough

[1] http://www.unixtop.org/man.shtml

Figure 7.2: Benchmark sets b1, ..., b6 on a HDD drive (24GB RAM)

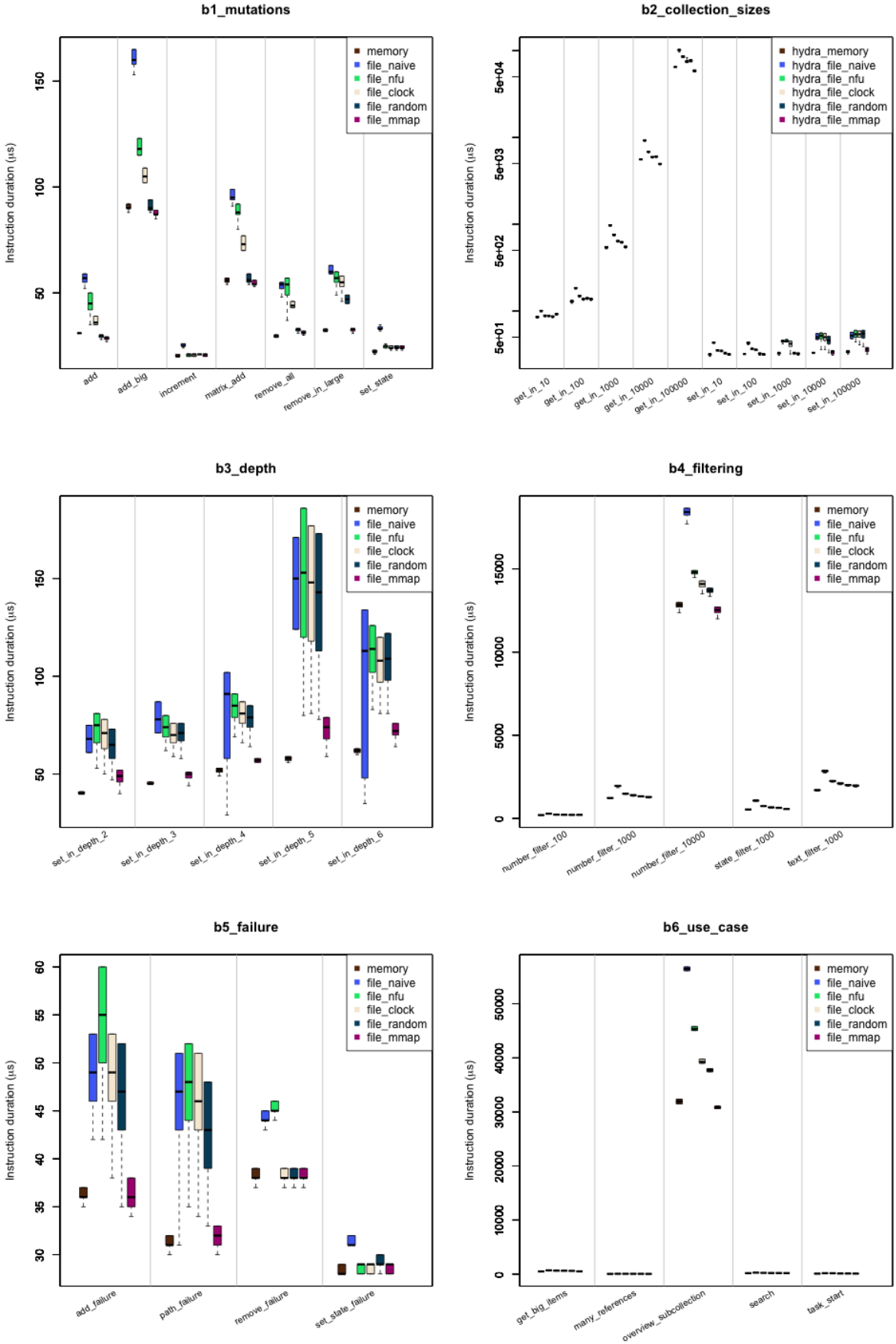
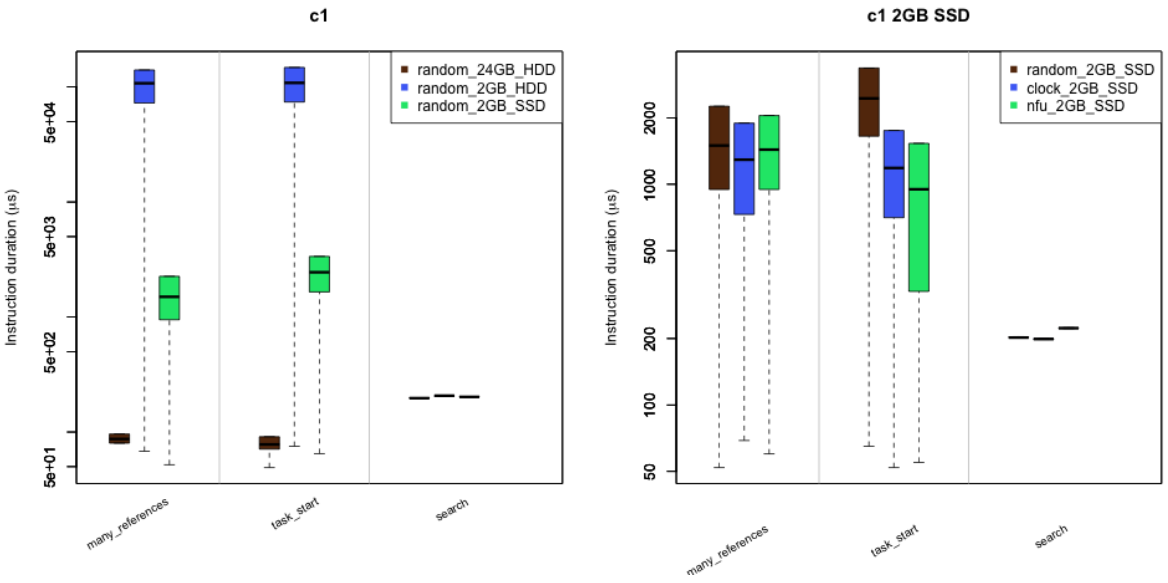


Figure 7.3: Benchmark set b7 on various machines



RAM size, the disk-based implementations in effect are not so different from the memory based implementation.

Furthermore, the standard deviation of `many_references` and `task_start` have increased from almost 0 to 500 compared to the previous experiments. The minima of these benchmarks are about the same as in the previous experiments. Because of the random collection steps the instructions take in these benchmarks, there can be quite some dispersion between the needed pages for subsequent instructions. In the worst case, all steps of subsequent instructions are different and thus a maximum number of pages need to be replaced. In the best case, the instructions are the same, and all pages are already in memory. In other words, what is already in the cache due to previous instructions heavily influences the instruction duration.

Real world cases would have what is called hot data in the cache which is data that is frequently accessed. For a fair real world conclusion per instruction, we would have to know what data we realistically should label as hot data. The approach of random steps in instructions in these experiments however, do give insights on the differences of instruction execution duration on hot data versus on cold data.

Unfortunately, when considering the processor load of an Alan-DB currently running for a client of M-industries analyzed in Section 2.4, a 50 times increase in instruction duration might not be a sufficient solution.

7.4. COMPARISON TO ORIENTDB

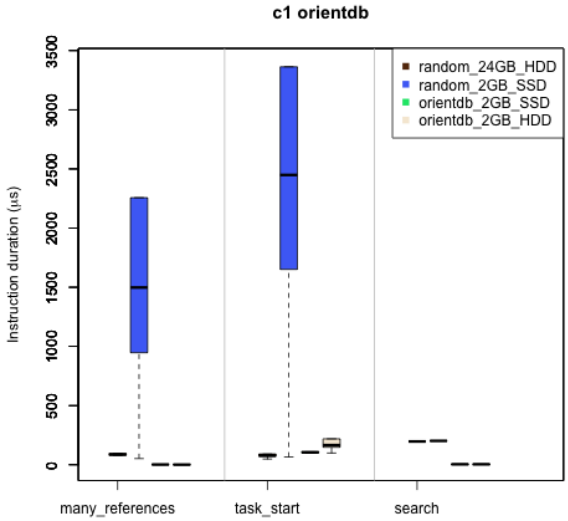
The experiments for OrientDB were done on the same server machine provided by M-industries (2GB RAM), both on an HDD and an SSD device. The total size of the data in OrientDB is 6.7GB as opposed to the 8.7GB of Alan-DB. The following figure shows the results for the benchmark set c1 with the SSD results of the previous benchmark as a reference.

From these experiments, we can conclude that OrientDB more efficiently can retrieve data than Alan-DB. Either on HDD or SSD, the read performance for OrientDB hardly differs. Write performance of the `task_start` benchmark on the other hand decreases slightly on HDD for OrientDB. The 24GB execution of Alan-DB performs very similar to OrientDB and even slightly better for the `task_start` benchmark. Since the OS of the 24GB machine in effect caches the data files into memory, the 24GB suggests that with a more sophisticated caching mechanism for Alan-DB, it can perform as well as OrientDB.

7.5. CONCLUSION

These experiments have shown that properly benchmarking databases is not trivial. Caching due to previous instructions heavily influences the performances of the current instruction. A fair benchmark would start with a page cache only consisting of realistic hot data (i.e. data that is frequently accessed). However, to

Figure 7.4: OrientDB on benchmark set b7



determine what realistic hot data is (or if it even exists) is not possible within the current set up. The box plots do give insight on the boundaries of the instruction execution time.

The micro-benchmarks have shown that in an ideal caching situation, the overhead of the storage layer hardly decreases the performance of Alan-DB. Even though various paging algorithms do show some performance differences, these differences dwarf in comparison to the impact real disk I/O, outside the OS file cache, has. Finding and replacing pages in the page cache thus isn't the bottleneck. Compared to the in memory implementation, however, the file based implementations performed very similar on machines with sufficient RAM. From this we can conclude that there is nothing to significantly improve in the disk based implementation of the data layer. Different implementations of byte layouts however, that strive to group hot data together, could improve performance.

The instruction layer, which was out of scope of this thesis, needs improvement or otherwise the datastore could not even handle collection queries on large datasets. Finally, the performance results of OrientDB have shown that Alan-DB needs some improvement in the caching mechanism and the implementation of a sophisticated mechanism similar to that of OrientDB's might be beneficial.

8

Conclusion and Future Work

This research addressed the question on how to best store a typed rooted graph with conditional data on disk for a write-heavy datastore named Alan-DB. Since this data format is quite unique we had to look and compare aspects of Alan-DB and its schema language Alan-Application to existing database types and reason based on the similarities on how to best achieve a reasonable solution. In order to determine whether the solution would perform well, we developed a benchmark language with the goal of being able to test this solution in a wide variety of scenarios, both for examining elementary instructions and datasets, and complex real world situations.

8.1. Main Findings

Benchmarking in the database world is often done rather minimally with different papers contradicting each other and easily can lead to misleading results. This thesis demonstrates the difficulties of benchmarking as well. The main insight and contribution of this thesis is that a benchmark language simplifies the effort of benchmarking extensively, which enables the possibility of further pushing the boundaries of analyzing performance.

The results of the experiments have shown that performance of instructions for the greatest part is dependent on the caching mechanism and what already is in the cache. Other than that, the performance overhead, as a result of rewriting the database in order to support disk persistence, hardly decreased in comparison to the already existing memory implementation.

8.2. Answers

8.2.1. What are the possible storage structures for storing and processing a TRDG on disk and what are their features?

All persistent storage structures involve writing data into files and only reading parts of these files back to memory. Since hardware is designed to read and write blobs of data of several kilobytes called pages, databases use paging mechanisms that swap a limited amount of pages from files to main memory. There are many paging algorithms and strategies. Some are simple and efficient in the actual page swapping, such as using a first-in-last-out buffer or random swapping. Other paging algorithms keep track of the frequency pages are requested or modified and try to reduce page swaps at the cost of more calculations and searches per page access.

There are two main data layout categories in general. The first one is the aggregate-based layout where data that is part of the same aggregate is stored close together, preferably on one page, and where links are created through indexed foreign keys. The main benefit of this approach is that retrieving and storing full aggregates can be done efficiently and storage requires less space then the other approach. The second one is the connection-based layout where data is organized as fixed sized objects of the same type. Entities in this organization might be spread throughout several files and pages, but the connections between them are direct. The main benefit of this approach is that creating, managing, and traversing complex structures is more efficient.

8.2.2. WHICH OF THESE STORAGE STRUCTURES ARE BEST SUITED FOR THE WORKLOADS OF A TYPICAL APPLICATION AT M-INDUSTRIES?

Alan-DB is a write-intensive and traversal-heavy database of which most mutations consist of small changes to existing data structures. This is because Alan-DB is mostly used in solutions for factories where machines, orders and the likes constantly are being updated. For this reason, the connection-based layout is better suited for Alan-DB.

In terms of paging algorithms, the experiments have shown that all paging algorithms that were benchmarked, excluding the naive, perform very similar.

8.2.3. HOW WELL DOES THE SOLUTION PERFORM COMPARED TO OTHER TYPES OF DATABASES?

As a comparison we benchmarked the solution presented in this thesis against OrientDB. When the dataset size did not exceed the available RAM, OrientDB performed very similar to Alan-DB. However, OrientDB significantly performed better when the dataset size exceeded available RAM. Moreover, the performance of Alan-DB in this case is probably too bad for a production environment database. OrientDB uses a more sophisticated page cache that has several layers of caching and separate caches for reading and writing. This suggests that the data layer solution for Alan-DB is good, but that a multi layer caching strategy might be essential for a future solution.

8.3. FUTURE WORK

The benchmark language could be extended and improved in order to develop a framework that could (roughly) predict performance given an Alan-Application data model. Real applications in production both fire automatic commands and commands as a result of user interaction to the database. Furthermore, the database sends push-based notifications to client applications. So with a method and a language to also describe an instruction load for the database, a simulation of a database in production could be generated that could be used to predict whether Alan-DB could handle a certain data size and instruction load. This framework furthermore could be used to investigate the hotness of certain data within a dataset (i.e. what parts of the dataset are being frequently requested or modified), which could be used to further improve the caching mechanism.

A

THESIS CODING WORK

This appendix gives an overview of the coding work that was done as part of the work of this thesis. The following table shows the lines of code (loc) and the number of files that were created or modified for this thesis.

Table A.1: Code

category	language	loc	files
instruction log analysis	C++	353	1
R files	R	463	4
data layer disk implementation	C/C++	933	1
storage layer	C/C++	2485	15
instruction layer	C++	6889	114
benchmark grammar and schema	Alan-Grammar, Alan-Schema	1434	2
benchmark implementation	C++	1961	9
benchmarks	Alan-Benchmark	1231	39
orientdb benchmark	Java	597	2
total		16346	187

These logged instructions are used by M-industries in case of a crash to recreate the dataset by applying these instructions on the latest data dump. The instruction log analysis code can read a folder containing logged instructions and was used to create the CSV data needed for the R code that created figure 2.1.

The instruction layer contains a lot of code, but most of this code was ported, meaning that the bulk of the code and its algorithmic complexity was already there. Most of the work in this layer was a rewrite of several data and transaction function calls.

The storage layer was completely written, both its API and its implementation as part of the work of this thesis. Both the storage and the data layer used some C++ features but were very close to pure C.

The benchmarking platform in code consists of several elements. The first part is the grammar written in one of Alan's languages which describes the grammar (or syntax) of Alan-Benchmark. Then, the schema code describes the connection between the grammar and existing Alan-Application and Alan-Instruction constructs. This code then is compiled by the Alan platform which generates C++ APIs (which are very large and not shown in the table) that are used in the benchmark implementation code. The benchmark implementation code sets up a datastore, fills it according to the benchmark, rewrites the benchmark instructions to Alan-Instructions, executes these instructions the number of times described by the benchmark and prints out the quantiles of the execution times of the instructions.

Finally, the OrientDB benchmarks were created in one simple Java project. This code first fires up a datastore, then creates a data model and then runs three benchmarks, every time truncating the datastore before executing the benchmark. The execution of a benchmark consists of filling the datastore with data and then sending OrientDB SQL statements following the benchmarks of B7 as close as possible. The quantiles of the execution times of the SQL statements are printed as a result.

B

BENCHMARKS

B.1. MUTATIONS

B.1.1. ADD

```
// application model
root (
    'd' -> dictionary ( )
)

// benchmark
initialize (
    'd' -> dictionary 0 ( )
)
repeat 1000 (
    update (
        'd' -> dictionary index ( )
    )
)
```

B.1.2. ADD_BIG

```
// application model
root (
    'd' -> dictionary (
        't1' -> text
        't2' -> text
        't3' -> text
        't4' -> text
        'n1' -> number
        'n2' -> number
        'n3' -> number
        'n4' -> number
        'sg1' -> stategroup (
            's1' -> ( )
            's2' -> ( )
            's3' -> ( )
        )
        'sg2' -> stategroup (
            's1' -> ( )
            's2' -> ( )
            's3' -> ( )
        )
    )
```

```
)

// benchmark
initialize (
    'd' -> dictionary 0 (
        't1' -> text "a"
        't2' -> text "b"
        't3' -> text "c"
        't4' -> text "d"
        'n1' -> number 1
        'n2' -> number 1
        'n3' -> number 1
        'n4' -> number 1
        'sg1' -> stategroup 's1' ( )
        'sg2' -> stategroup 's1' ( )
    )
)
repeat 1000 (
    update (
        'd' -> dictionary index (
            't1' -> text "e"
            't2' -> text "f"
            't3' -> text "g"
            't4' -> text "h"
            'n1' -> integer 2
            'n2' -> integer 2
            'n3' -> integer 2
            'n4' -> integer 2
            'sg1' -> stategroup 's2' ( )
            'sg2' -> stategroup 's3' ( )
        )
    )
)
```

B.1.3. INCREMENT

```
// application model
root (
    'n' -> number
)

// benchmark
initialize (
    'n' -> number 1
)

repeat 1000 (
    update (
        'n' -> integer 2
    )
)
```

B.1.4. MATRIX_ADD

```
// application model
root (
    'd' -> dictionary ( )
    'm' -> matrix ( . 'd' ) ( )
)

```

```
// benchmark
initialize (
    'd' -> dictionary 1000 ( )
    'm' -> matrix 0 ( )
)

repeat 1000 (
    update (
        'm' -> matrix random 1000 ( )
    )
)
```

B.1.5. REMOVE_ALL

```
// application model
root (
    'd' -> dictionary ( )
)

// benchmark
initialize (
    'd' -> dictionary 1000 ( )
)

repeat 1000 (
    update (
        'd' -> dictionary index remove
    )
)
```

B.1.6. REMOVE_IN_100000

```
// application model
root (
    'd' -> dictionary ( )
)

// benchmark
initialize (
    'd' -> dictionary 100000 ( )
)

repeat 1000 (
    update (
        'd' -> dictionary random 100000 remove
    )
)
```

B.1.7. SET_STATE

```
// application model
root (
    'sg' -> stategroup (
        's0' -> ( )
        's1' -> ( )
    )
)

// benchmark
```

```
initialize (
    'sg' -> stategroup 's0' ( )
)

repeat 500 (
    update (
        'sg' -> stategroup 's1' set to ( )
    ) ,
    update (
        'sg' -> stategroup 's0' set to ( )
    )
)
```

B.1.8. SET_REFERENCE

```
// application model
root (
    'd' -> dictionary ( )
    'r' -> reference ( . 'd' )
)

// benchmark
initialize (
    'd' -> dictionary 10000 ( )
)

repeat 1000 (
    update (
        'r' -> reference random 10000
    )
)
```

B.2. COLLECTION SIZES

This benchmark set consists of two ranges of benchmarks where the application model as well as the instructions are the same for each range. The only difference is the size of the collections. The following two subsection describe these benchmark, but *X* should be replaced by a number specifying the collection size. The values for this number can be 10, 100, 1.000, 10.000, 100.000, 1.000.000 or 10.000.000. So in total this benchmark set consists of 12 benchmarks.

B.2.1. SET_IN_X

```
// application model
root (
    'd' -> dictionary (
        'n' -> number
    )
)

// benchmark
initialize (
    'd' -> dictionary X (
        'n' -> number 1
    )
)

repeat 1000 (
    update . 'd' : random (
        'n' -> integer 2
```

```
    )
)
```

B.2.2. GET_IN_X

```
// application model
root (
    'd' -> dictionary (
        'n' -> number
    )
)

// benchmark
initialize (
    'd' -> dictionary X (
        'n' -> number 1
    )
)

repeat 1000 (
    collection select . 'd'
    'n' -> number : 'n'
    limit 10
)
```

B.3. DEPTH

B.3.1. SET_IN_DEPTH_2

```
// application model
root (
    'd1' -> dictionary (
        'd2' -> dictionary (
            'n' -> number
        )
    )
)

// benchmark
initialize (
    'd1' -> dictionary 1000 (
        'd2' -> dictionary 1000 (
            'n' -> number 1
        )
    )
)

repeat 1000 (
    update . 'd1' : random . 'd2' : random ( 'n' -> integer 2 )
)
```

B.3.2. SET_IN_DEPTH_3

```
// application model
root (
    'd1' -> dictionary (
        'd2' -> dictionary (
            'd3' -> dictionary (
                'n' -> number
```

```
            )
        )
    )
)

// benchmark
initialize (
    'd1' -> dictionary 100 (
        'd2' -> dictionary 100 (
            'd3' -> dictionary 100 (
                'n' -> number 1
            )
        )
    )
)

repeat 1000 (
    update . 'd1' : random . 'd2' : random . 'd3' : random ( 'n' -> integer 2 )
)
```

B.3.3. SET_IN_DEPTH_4

```
// application model
root (
    'd1' -> dictionary (
        'd2' -> dictionary (
            'd3' -> dictionary (
                'd4' -> dictionary (
                    'n' -> number
                )
            )
        )
    )
)

// benchmark
initialize (
    'd1' -> dictionary 100 (
        'd2' -> dictionary 100 (
            'd3' -> dictionary 10 (
                'd4' -> dictionary 10 (
                    'n' -> number 1
                )
            )
        )
    )
)

repeat 1000 (
    update . 'd1' : random . 'd2' : random . 'd3' : random . 'd4' : random ( 'n
    ' -> integer 2 )
)
```

B.3.4. SET_IN_DEPTH_5

```
// application model
root (
    'd1' -> dictionary (
        'd2' -> dictionary (
            'd3' -> dictionary (
```

```
                'd4' -> dictionary (
                    'd5' -> dictionary (
                        'n' -> number
                    )
                )
            )
        )
    )
)

// benchmark
initialize (
    'd1' -> dictionary 100 (
        'd2' -> dictionary 10 (
            'd3' -> dictionary 10 (
                'd4' -> dictionary 10 (
                    'd5' -> dictionary 10 (
                        'n' -> number 1
                    )
                )
            )
        )
    )
)

repeat 1000 (
    update . 'd1' : random . 'd2' : random . 'd3' : random . 'd4' : random . 'd5' : random ( 'n' -> integer 2 )
)
```

B.3.5. SET_IN_DEPTH_6

```
// application model
root (
    'd1' -> dictionary (
        'd2' -> dictionary (
            'd3' -> dictionary (
                'd4' -> dictionary (
                    'd5' -> dictionary (
                        'd6' -> dictionary (
                            'n' -> number
                        )
                    )
                )
            )
        )
    )
)

// benchmark
initialize (
    'd1' -> dictionary 10 (
        'd2' -> dictionary 10 (
            'd3' -> dictionary 10 (
                'd4' -> dictionary 10 (
                    'd5' -> dictionary 10 (
                        'd6' -> dictionary 10 (
                            'n' -> number 1
                        )
                    )
```

```
                    )
                )
            )
        )
)

repeat 1000 (
    update . 'd1' : random . 'd2' : random . 'd3' : random . 'd4' : random . '
    d5' : random . 'd6' : random ( 'n' -> integer 2 )
)
```

B.4. FILTERING

B.4.1. NUMBER_FILTER_X

There are three benchmarks for the number filter to explore whether the collection size influences filtering. The values for *X* in this benchmark are 100, 1.000, and 10.000.

```
// application model
root (
    'd1' -> dictionary (
        'n' -> number
    )
)

// benchmark
initialize (
    'd1' -> dictionary X (
        'n' -> number random 1000
    )
)

repeat 1000 (
    collection select . 'd1'
    'n' -> number : 'n' > 100
    limit 10
)
```

B.4.2. TEXT_FILTER_1000

```
// application model
root (
    'd1' -> dictionary (
        't' -> text
    )
)

// benchmark
initialize (
    'd1' -> dictionary 1000 (
        't' -> random ( 'aaaaaa' 'bbbbbbb' 'ccccccc' )
    )
)

repeat 1000 (
    collection select . 'd1'
    't' -> text : 't' ( [ 'a' ] * )
    limit 10
)
```

B.4.3. STATE_FILTER_1000

```
// application model
root (
    'd' -> dictionary 1000 (
        'sg' -> stategroup (
            's0' -> ( )
            's1' -> (
                'n' -> number
            )
        )
    )
)

// benchmark
initialize (
    'd' -> dictionary 10000 (
        'sg' -> stategroup (
            's0' -> 1 ( )
            's1' -> 1 (
                'n' -> number 1
            )
        )
    )
)

repeat 1000 (
    collection select . 'd'
    'n' ->  ? 'sg' * 's1' number : n
    limit 10
)
```

B.5. FAILURE

B.5.1. SET_STATE_FAILURE

The instruction tries to set the state group in a state it already is in.

```
// application model
root (
    'sg' -> stategroup (
        's0' -> ( )
        's1' -> ( )
    )
)

// benchmark
initialize (
    'sg' -> stategroup (
        's0' -> 1 ( )
        's1' -> 0 ( )
    )
)

repeat 1000 (
    update (
        'sg' -> stategroup 's0' set to ( )
    )
)
```

B.5.2. PATH_FAILURE

All stategroups are initialized at $s0$ but the instruction expects it to be at state $s1$.

```
// application model
root (
    'd' -> dictionary (
        'sg' -> stategroup (
            's0' -> ( )
            's1' -> (
                'n' -> number
            )
        )
    )
)

// benchmark
initialize (
    'd' -> dictionary 10000 (
        'sg' -> stategroup (
            's0' -> 1 ( )
            's1' -> 0 (
                'n' -> number
            )
        )
    )
)

repeat 1000 (
    update 'd1' : random ? 'sg' * 's1' (
        'n' -> number 2
    )
)
```

B.5.3. ADD_FAILURE

The instruction tries to add an entry with a key that already exists.

```
// application model
root (
    'd' -> dictionary ( )
)

// benchmark
initialize (
    'd' -> dictionary 10000 ( )
)
repeat 1000 (
    update (
        'd' -> dictionary random 10000 ( )
    )
)
```

B.5.4. REMOVE_FAILURE

```
// application model
root (
   'd' -> dictionary ( )
)

// benchmark
initialize (
```

```
    'd' -> dictionary 1000 ( )
)
repeat 1000 (
    update (
        'd' -> dictionary 10 remove
    )
)
```

B.6. USE CASE

All benchmarks within the Use Case benchmark set conform to the same model and the same initialization. Therefore, the model is presented once below and every following subsection shows only the instruction.

```
initialize (
  'Clients' -> dictionary 100 (
    'Is Premium Client' -> stategroup (
      'Yes' -> 2 ( )
      'No' -> 8 ( )
    )
    'Name' -> text "Client Name"
    'Contact' -> group (
      'Phone' -> number 0123456789
      'Address' -> text "Some street"
      'E-mail' -> text "contact@acme.com"
    )
  )
  'Materials' -> dictionary 10000 (
    'Description' -> text "some description..."
    'Available In Stock' -> number random 1000
  )
  'Products' -> dictionary 30 (
    'Description' -> text "some other description..."
    'Price' -> number random 10000
  )
  'Orders' -> dictionary 100 (
    'Client' -> reference random 100
    'Product' -> reference random 30
    'Quantity' -> number random 1000
    'Modifications' -> dictionary 5 (
      'Description' -> text "Modifications description"
      'Status' -> stategroup (
        'Finished' -> 0 ( )
        'Not Finished' -> 1 ( )
      )
      'Added Materials' -> matrix 100 (
        'Quantity' -> number random 10
      )
    )
    'Status' -> stategroup (
      'Scheduled' -> 8 ( )
      'Released' -> 1 ( )
      'Started' -> 1 ( )
      'Unplanned' -> 5 ( )
    )
    'Connected To Sales Order Line' -> stategroup (
      'No' -> 2 ( )
      'Yes' -> 5 (
        'Sales Order Line' -> text "description..."
      )
    )
    'Due Date' -> number random 10000000
```

```
    'Scheduled Start Date' -> number random 10000000
    'Scheduled End Date' -> number random 10000000
    'Planner Group' -> text "group x"
    'Requested Group' -> text "group y"
    'First Order' -> number random 10
    'Planning Pool' -> text "pool z"
  )
  'Machines' -> dictionary 300 (
    'Current Task' -> stategroup (
      'Selected' -> 8 (
        'Current Order' -> reference random 100
        'Status' -> stategroup (
          'Started' -> 5 (
            'Starting Time' -> number random 10000000
            'Iteration' -> number random 1000
          )
          'Canceled' -> 1 (
            'Last Iteration' -> number random 10
            'Reason' -> text "error"
          )
          'Not Started' -> 2 ( )
          'Finished' -> 2 (
            'Process Duration' -> number random 10000
          )
        )
      )
      'Not Selected' -> 2 (
        'Reason' -> stategroup (
          'Machine Unavailable' -> 1 ( )
          'Machine Not Planned' -> 10 ( )
        )
      )
    )
  )
)
```

B.6.1. SEARCH

This benchmark describes a set of successive collection queries that represent a user narrowing his search by typing in a search input.

```
repeat 1000 (
    collection select . 'Clients'
    'Name' -> text : 'Name' ( [ 'a' ] * )
    limit 10 ,
    collection select . 'Clients'
    'Name' -> text : 'Name' ( [ 'ab' ] * )
    limit 10 ,
    collection select . 'Clients'
    'Name' -> text : 'Name' ( [ 'abc' ] * )
    limit 10 ,
    collection select . 'Clients'
    'Name' -> text : 'Name' ( [ 'abcd' ] * )
    limit 10 ,
    collection select . 'Clients'
    'Name' -> text : 'Name' ( [ 'abcde' ] * )
    limit 10
)
```

B.6.2. OVERVIEW_SUBCOLLECTION

This benchmark represents a user requesting all the items of a subset. Specifically, the user wants to see what and how many extra materials where needed for all orders that have been registered.

```
repeat 1000 (
    collection select . 'Orders' . 'Modifications' % 'Added Materials'
    'Quantity' -> number : 'Quantity'
    limit 10
)
```

B.6.3. TASK_START

This benchmark represents the initialization of a task on a terminal.

```
repeat 1000 (
    update . 'Machines' : index (
        'Current Task' -> stategroup 'Selected' set to (
            'Product' -> reference "5"
            'Status' -> stategroup 'Started' (
                'Starting Time' -> integer 12345
                'Iteration' -> integer 1
            )
        )
    )
)
```

B.6.4. GET_BIG_ITEMS

This benchmark represents a retrieval of items with many properties.

```
repeat 1000 (
    collection select . 'Orders'
    'Product' -> reference : 'Product'
    'Status' -> stategroup : 'Status'
    'Connected To Sales Order Line' -> stategroup : 'Connected To Sales Order
    Line'
    'Quantity' -> number : 'Quantity'
    'Due Date' -> number : 'Due Date'
    'Scheduled Start Date' -> number : 'Scheduled Start Date'
    'Scheduled End Date' -> number : 'Scheduled End Date'
    'Planner Group' -> text : 'Planner Group'
    'Requested Group' -> text : 'Requested Group'
    'First Order' -> number : 'First Order'
    'Planning Pool' -> text : 'Planning Pool'
    limit 10
)
```

B.6.5. MANY_REFERENCES

```
repeat 1000 (
  collection . 'Machines' : random select
  'Premium' -> ? 'Current Task' * 'Selected' > 'Current Order' > 'Client'
    stategroup : 'Is Premium Client'
  limit 10
)
```

B.7. ORIENTDB

OrientDB SQL notation is slightly different from regular SQL notation. For example, one can directly assign embedded JSON to properties. Besides that, note that variable assignment and dollar string interpolation

is not official OrientDB SQL notation, but should be considered syntactic sugar hiding Java code for these examples, with '$index' representing the current iteration of the total amount of instructions fired at the datastore.

B.7.1. MANY_REFERENCES

```
SELECT machines['$index'].currentTask.node.currentOrder.client.isPremiumClient.
    node AS Premium
    FROM Root
    WHERE machines['$index'].currentTask.state == 0
    LIMIT 1
```

B.7.2. SEARCH

```
SELECT name From Client WHERE name like 'C%' LIMIT 10;
SELECT name From Client WHERE name like 'Cl%' LIMIT 10;
SELECT name From Client WHERE name like 'Clie%' LIMIT 10;
SELECT name From Client WHERE name like 'Clien%' LIMIT 10;
SELECT name From Client WHERE name like 'Client%' LIMIT 10;
```

B.7.3. TASK_START

This mutation was a little harder to model since it actually inserts two nodes on an update of another. These statements in addition to a SELECT lookup had to be separated, resulting in four SQL statements representing one Alan instruction.

```
order = SELECT @rid FROM Order WHERE quantity > 5 LIMIT 1

started = INSERT INTO Started2 (startingTime, iteration) VALUES (12345, 1)
    RETURN @rid;

selected = INSERT INTO Selected
    SET currentOrder = $order, status = {"@class": "Stategroup", "@type": "d",
    "state": 0, "node": "$started"}
    RETURN @rid

UPDATE Root
    SET machines['$index'].currentTask = {"@class": "Stategroup", "@type": "d",
     "state": 0, "node": "$selected"}
    RETURN AFTER @this.machines['$index'].currentTask.node
```

BIBLIOGRAPHY

[1] A. Silberschatz, H. F. Korth, S. Sudarshan, *et al., Database system concepts,* 6th ed. (McGraw-Hill New York, 2011).

[2] K. Chodorow, *MongoDB: the definitive guide* (" O'Reilly Media, Inc.", 2013).

[3] I. Robinson, J. Webber, and E. Eifrem, *Graph databases* (" O'Reilly Media, Inc.", 2013).

[4] I. Taranov, I. Shcheklein, A. Kalinin, L. Novak, S. Kuznetsov, R. Pastukhov, A. Boldakov, D. Turdakov, K. Antipin, A. Fomichev, *et al., Sedna: native xml database management system (internals overview),* in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (ACM, 2010) pp. 1037–1046.

[5] *Xindice internals,* https://xml.apache.org/xindice/dev/guide-internals.html, accessed: 2016-05-14.

[6] J. Hunter, *Inside MarkLogic Server* (2013).

[7] M. Dayarathna and T. Suzumura, *Xgdbench: A benchmarking platform for graph stores in exascale clouds,* in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on* (IEEE, 2012) pp. 363–370.

[8] *Xindice internals,* http://orientdb.com/docs/last/index.html, accessed: 2016-05-14.

[9] T. Johnson and D. Shasha, *X3: A low overhead high performance buffer management replacement algorithm,* (1994).

[10] B. S. Gill and D. S. Modha, *Wow: wise ordering for writes-combining spatial and temporal locality in non-volatile caches,* in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies-Volume 4* (USENIX Association, 2005) pp. 10–10.

[11] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, *Linkbench: a database benchmark based on the facebook social graph,* in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (ACM, 2013) pp. 1185–1196.

[12] A. Prat and A. Averbuch, *Benchmark design for navigational pattern matching benchmarking,* http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf (2014).

[13] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, *The ldbc social network benchmark: interactive workload,* in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (ACM, 2015) pp. 619–630.

[14] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey, *Benchmarking database systems for social network applications,* in *First International Workshop on Graph Data Management Experiences and Systems* (ACM, 2013) p. 15.

[15] I. Fundulaki and A. Averbuch, *Overview and analysis of existing benchmark frameworks,* (2013).

[16] L. Afanasiev, I. Manolescu, and P. Michiels, *Member: a micro-benchmark repository for xquery,* in *International XML Database Symposium* (Springer, 2005) pp. 144–161.

[17] W. Litwin, *Linear hashing: a new tool for file and table addressing.* in *VLDB*, Vol. 80 (1980) pp. 1–3.

[18] Y. Abubakar, T. Adeyi, and I. G. Auta, *Performance evaluation of nosql systems using ycsb in a resource austere environment,* Performance Evaluation **7** (2014).