Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Structured metadata for the JULEA storage framework

vorgelegt von

Michael Straßberger

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6527713
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Kira Duwe

Hamburg, 2019-05-21

Abstract

Today's high performance computing produces large amounts of data for scientific research in different areas. To handle such amounts of data different solutions exists in the HPC (high performance computing) field.

In the past, most storage solutions lacked the ability to add own user defined metadata to the files. To accommodate this storage frameworks developed SDDF (self describing data format) standards which stores the metadata alongside with the object data. While SDDF provide a solution for storing metadata, it does not solve the issue of searching in large datasets. A search would have to read every file for its metadata information, instead of searching in a centralized database.

This thesis provides a concept and implementation for a flexible structured metadata management based on the JULEA storage framework.

Contents

1. Introduction

In this chapter, a brief introduction is given. The motivation and the goals of this thesis are discussed. Also, other solutions in the research field of metadata management in high performance storage environments are pointed out.

1.1. Motivation

In recent years, the demand for high throughput and capacity storage system has increased drastically. The reason for this development is the increased processing power due to Moores-Law and the more common usage of hpc in scientific research.

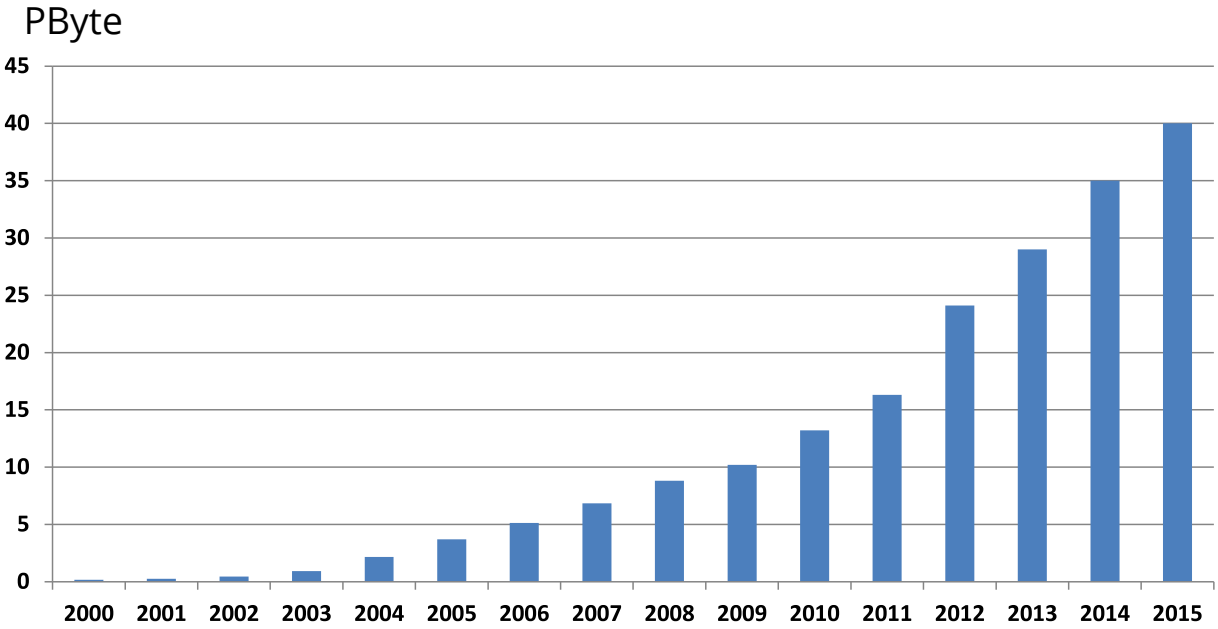

Figure 1.1.: Storage of DKRZ Tape Archive [dkr19]

One major contributor to these big data amounts is the weather and climate research as seen in Figure 1.1). It is foreseeable that this will increase even more in the upcoming years.

One problem of this expansion is the increasing complexity for organizing and searching those big data storages. Most filesystems don't have a way of managing user defined metadata to make those data sources efficiently searchable.

Another big problem is that metadata operations usually tend to be slow and with the increasing size of clusters result into a bottleneck for the storage performance.

Although there exists abstraction software libraries that provide the user with a way to add metadata to their data, the problem of making them easily searchable is not solved.

This thesis will make a contribution to solve the problem of a searchable metadata storage for those scientific data. Other approaches in this field are discussed in Section 1.2.

The metadata will be stored by extending the JULEA storage framework (Section 2.3) with a module for searchable metadata and a reference implementation with SQLite3. The objective of this approach is to provide a swift and accessible way to search a tremendous data source.

At first I will describe what other solutions are to undertake the given problem with metadata management. After that, a brief introduction of mandatory definitions are given to better understand the design and implementation of this work. Later I will evaluate the performance of the reference implementation and give an outlook of future improvements that can be made to this approach.

1.2. Related Work

Managing filesystem and user-defined metadata in a parallel file system is a difficult task. There exists different approaches to make metadata operations more scalable in this environment.

1.2.1. Lustre

Lustre uses a centralized concept of metadata [Bra19, pp 72]. The metadata servers handle the workload of creating files, directories, symlinks and other known operations from local file systems.

Lustre offers to load-balance between multiple metadata servers to allow scalability in high traffic storage environments. The metadata in Lustre also contain information of

the stripe size across multiple object storage servers. This design makes it mandatory to always contact the metadata server prior to opening, writing or reading a file.

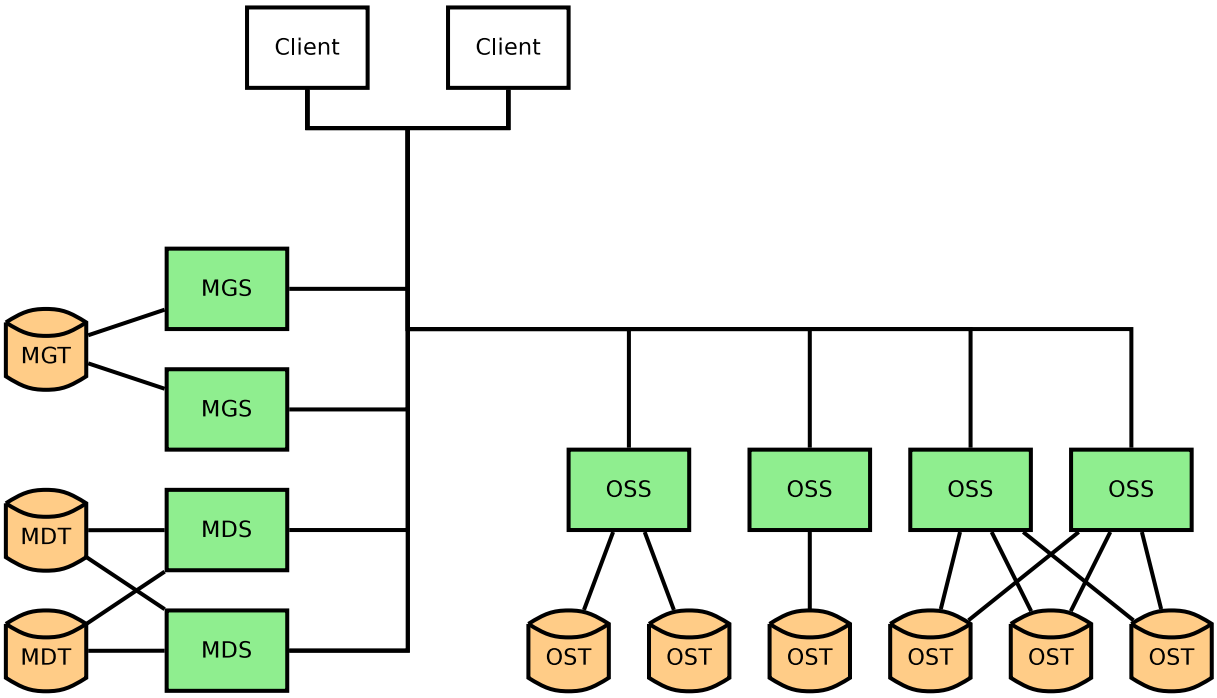

Figure 1.2.: Lustre cluster, taken from [Tho17, p 12]

A Lustre filesystem cluster consists of three server types as shown in Figure 1.2:

- Management Server (MGS)
- Metadata Server (MDS)
- Object Storage Server (OSS)

Management Server

The MGS stores the necessary information of the cluster state in its Management Target (MGT). This information contains all active Lustre server, clients and configuration [lus17].

Metadata Server

The MDS is responsible for providing the file system namespace. It also stores the inodes of the filesystem onto its Metadata Storage Target (MDT) [lus17].

Object Storage Server

The OSS provides the data storage of the Lustre Cluster. It can manage numerous Object Storage Targets (OST). Files can be striped across multiple OST to achieve very high throughput rates [lus17].

1.2.2. Ceph and Rados

Rados is an object store (Section **??**) that tries to eliminate bottlenecks by a highly distributed concept [WBM+06]. It also reduces the metadata load by making the clients calculate, with cluster wide known devices mappings and seeds for random number generators, the location of objects.

In contrast to Lustre, Ceph allows to add additional storage nodes into the cluster and it handles the redistribution of the objects to the new storage devices.

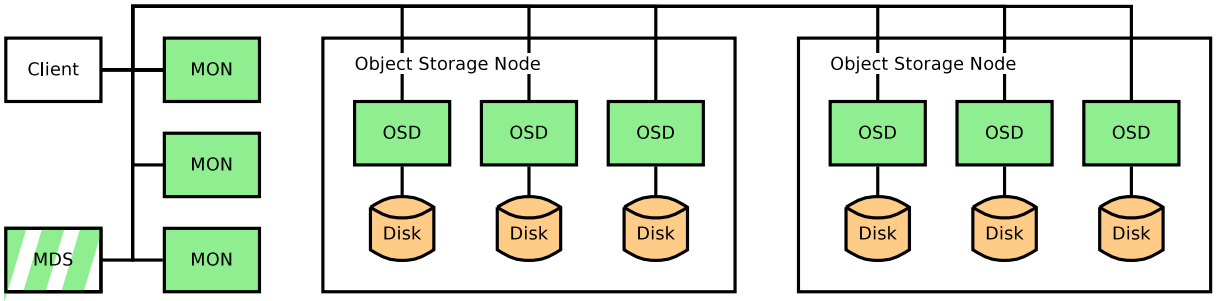

Figure 1.3.: Rados cluster, taken from [Tho17, p 16]

An example architecture of a Rados cluster is presented in Figure 1.3. There has to be an uneven number of Monitoring nodes (MON) in the cluster. This is due to the fact, that decisions in the cluster are made with on a quorum base, to ensure there exists only one consistent state of the cluster.

The object storage nodes can have multiple Object Storage Devices (OSD).

Ceph/CephFS is a POSIX compatible filesystem based on the Rados object store. Ceph treats the mandatory metadata for the hierarchically concept described in Section 2.1.1 as

a serialized object in rados itself. The metadata is then managed by a Metadata Server (MDS) as shown in Figure 1.3.

Rados relies on the user to provide a way to add metadata as described in Section 2.2.2.

1.2.3. LWFS

The Lightweight File System (LWFS) takes the metadata reduction to a bare minimum [OWR+06]. LWFS evaluated a different procedure to tackle the bottlenecks of a PFS by inverting the client/server behavior. This conveys into the IO server being in the role of pulling the data from the client for writes and push the data to the clients for reads. One of the few metadata information in LWFS are authorization and authentication. In contrast to Ceph and Lustre these metadata operations are processed at the storage node itself. This design eliminates the bottleneck of a centralized, load-balanced or distributed metadata management.

Since LWFS does not provide user defined metadata at all its up to the user to provide an own metadata source or make use of libraries like HDF5 in Section 1.2.5

1.2.4. SoMeta

Scalable Object-centric Metadata Management for High Performance Computing (SoMeta) is aimed to extend existing HPC storage solutions like Lustre, LWFS and Rados with a tag based metadata management [TBD+17].

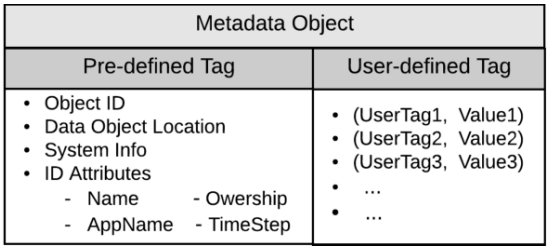

Figure 1.4.: SoMeta metadata object structure [TBD+17]

SoMeta stores with every metadata object pre-defined tags like its ID and system information as shown in figure 1.4 The user is able to add own tags to these metadata objects as key-value pairs. These pairs can be updated, deleted and created dynamically.

SoMeta manages the metadata in memory to provide good performance on all operations and checkpoints it to persistent storage on given time intervals.

In contrast to the approach of this thesis, SoMeta uses a flat namespace.

1.2.5. HDF5

The Hierarchical Data Format (HDF5) is among other similar file formats commonly used in scientific research applications [FHK+11].

HDF5 allows the user to specify groups that can contain other groups, datasets or attributes. The hierarchy of HDF5 is like the directory tree in a POSIX filesystem.

In Figure 1.5 an example group tree is shown for an application that writes its data as checkpoints. Each checkpoint can be annotated with attributes like configuration of the application or other metadata useable to either parse the data or continue from the checkpoint.

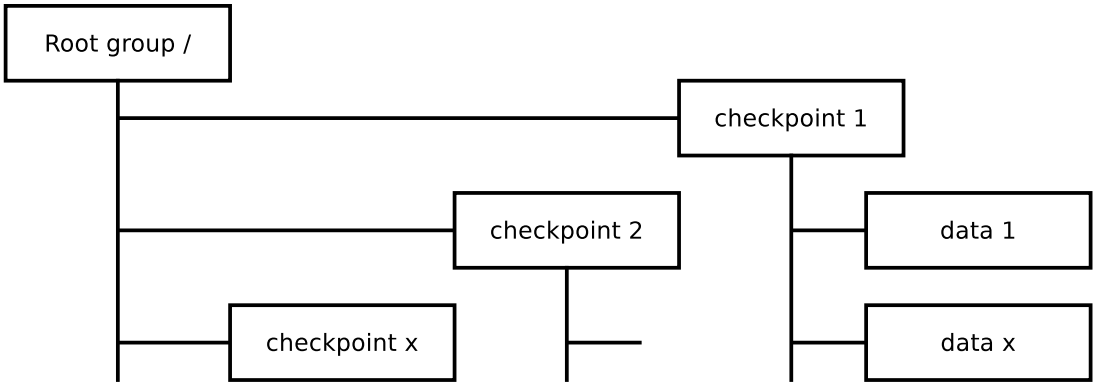

Figure 1.5.: HDF5 group structure on a checkpoint example

HDF5 file format

The file format of HDF5 consists of a superblock like header with general management information like allocated space, group root entry and pointer to several other information objects.

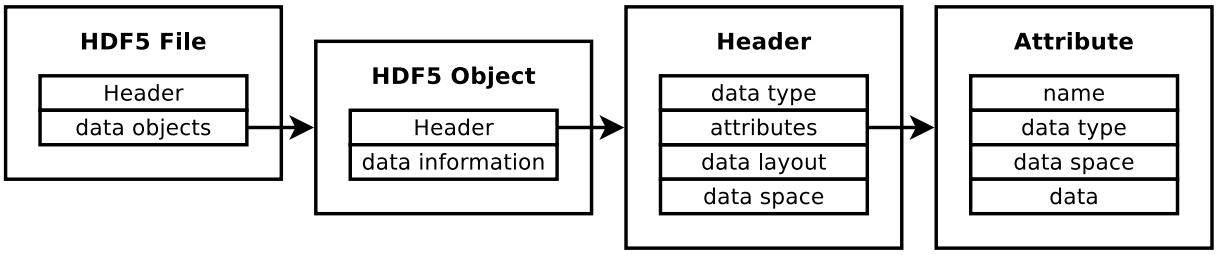

Figure 1.6.: HDF5 abstract data structure with metadata focus [hdf19]

The HDF5 file consists of numerous HDF5 objects, each with their own header and data information as seen in Figure 1.6. The header describes how to interpret data information and additional metadata or pointers to them, either to describe or annotate the object.

The metadata is saved as attribute objects. Attributes are build from:

- A name that is unique within the described object
- The type of data of this attribute for example:
  - Integer
  - Float
  - Date/time
  - String
  - ...
- The dimension of the data and the size of each dimension. This can be used to store small vectors and matrices into the metadata header. For example, a copy of the data with a reduced resolution to create a preview of the data.
- And the data itself.

Since the attributes are saved into the header of the object itself the size of the attributes had to be below 64KB until version 1.8.x of HDF5. With version 1.8.x HDF5 introduced a feature of dense attribute storage. It allows, if a given size threshold is reached, to move the attributes to another location and leaves a reference in the object header instead of the attributes. This behavior is handled automatically by the HDF5 library and can be fine tuned by the user.

2. Background

In this chapter, a definition of filesystems and metadata is given. The focus lies on the difference between metadata produced and needed by the filesystem and user generated metadata. It also gives a brief understanding what the JULEA storage framework is.

2.1. Filesystem

A filesystem organizes files that contain data and have specific metadata. The data of a file can be of any type e.g. text, machine code, image. The filesystem specific metadata in most cases contain information like access rights of users, timestamps of creation and last modification and the size of the data. (see 2.2.1)

The data and metadata of a filesystem gets usually stored in a block storage device. Therefore, the data gets split into blocks according the block size of the storage device. A filesystem may use approaches like extents to efficiently and dynamically store which blocks belong to which file [TB15, pp 284] [ext19a].

Most filesystems provide a way to organize files in a hierarchical manner in so-called directories. The file specific metadata gets stored in information nodes or short inodes, that get stored in a centralized data structure somewhere on the block device [ext19b].

2.1.1. Local Filesystem

A local filesystem is mainly used on single block devices. In some use cases, volume group managers can be used to either group multiple storage devices into one big block device or create redundancy to accommodate hardware failure.

The filesystem has access to every data and metadata block at any given time implied there is no hardware failure preventing access to some blocks of the storage device. Depending on the used storage technology, mechanical disk drives or flash based chips, the access to an specific file has certain latencies ranging from sub milliseconds to 20 ms.

One key property of the local filesystem is that in general it only has one client at a time. In most cases, this client is the operating system kernel which provides an interface for user programs to access the filesystem.

2.1.2. Object Store

Object stores provide a way to access block storage devices as data objects instead of blocks. An object usually consists of the data itself and in some cases a variable amount of metadata. The object gets usually addressed with a UUID (universally unique identifier).

In contrast to a filesystem and local filesystem, an object store does not have a hierarchy and thus requires less management overhead on the running machine [MGR03]. Object stores are used in some cases as an abstraction layer for other filesystems or they get directly accessed from an application.

2.2. Metadata

Metadata provides information about certain aspects of data and describes the content and structure of them [BS94]. Metadata is a loose term since depending on the perspective of the context, it is needed and/or created for different reasons. The main advantage of having good and structured metadata is being able to know what content is in the associated data and to search for specific content.

Outside of scientific research companies invest large sums of money and time into tuning their metadata creation and evaluation. Without good searchable metadata services like web search engines, music catalogues or inventory control systems it would be very inefficient to search for something in them.

For a better distinction, this thesis will differentiate between metadata created and used for the management of a filesystem itself and user provided metadata in form of key-value pairs.

2.2.1. Filesystem Metadata

Metadata of the filesystem usually contains information about [TB15, pp 271] [ext19b]:

- the file name which is used to access
- the owner of a file
- access rights for the users, who should be allowed to read or write
- temporal aspects like, file creation date, modification date and last access time
- location of the data and its size

This metadata is usually stored in a centralized metadata database, which gets stored like a file itself onto the underlying block storage.

In most cases, the filesystem does not have information of the stored data itself. Although some operating systems may implement some sort of full text indexing on top of the filesystem to speed up certain search operations of the user.

If, for example, one wants to search for specific keywords in a directory with text files, the application needs to open and read all files to search for those keywords.

2.2.2. User Metadata

User metadata describes information given by the user/creator of a file. The information provided by this metadata varies heavily depending on the type of data it describes.

Since there is no actual limit for the user metadata, it is difficult for a filesystem or other metadata storage databases to come up with a way to search it efficiently. Scientific storage frameworks like HDF5 and ADIOS provide the standard of SDDF (self describing data format). (see Section 1.2.5) These libraries provide for the user a tool to define metadata for describing the structure of their data.

2.3. JULEA

JULEA is an adjustable storage framework. It provides three contrasting abstracted backend types to interact with it. JULEA can be considered a distributed parallel filesystem since one of its basic design principles is a client-server architecture.

One of the design goals of JULEA is to be able to create rAPId prototypes of new object- or key-value stores in the storage research field. It is also considered to be a learning tool for students new to this subject.

This is achieved by a polymorphic design pattern known as in OOP (Object Oriented Programming) [Mey88, pp. 467]. JULEA defines an abstracted API for its backend types. It then offers multiple interchangeable backend implementations for each backend type. On runtime, JULEA loads the specified implementation of that API as a library for example the MongoDB interface for a key-value store (see Fig. 2.1)

Also, a major contributor to this goal is that JULEA is completely written in the user space. This makes it easy for beginners to write and debug code for the JULEA codebase without knowledge of kernel space development.

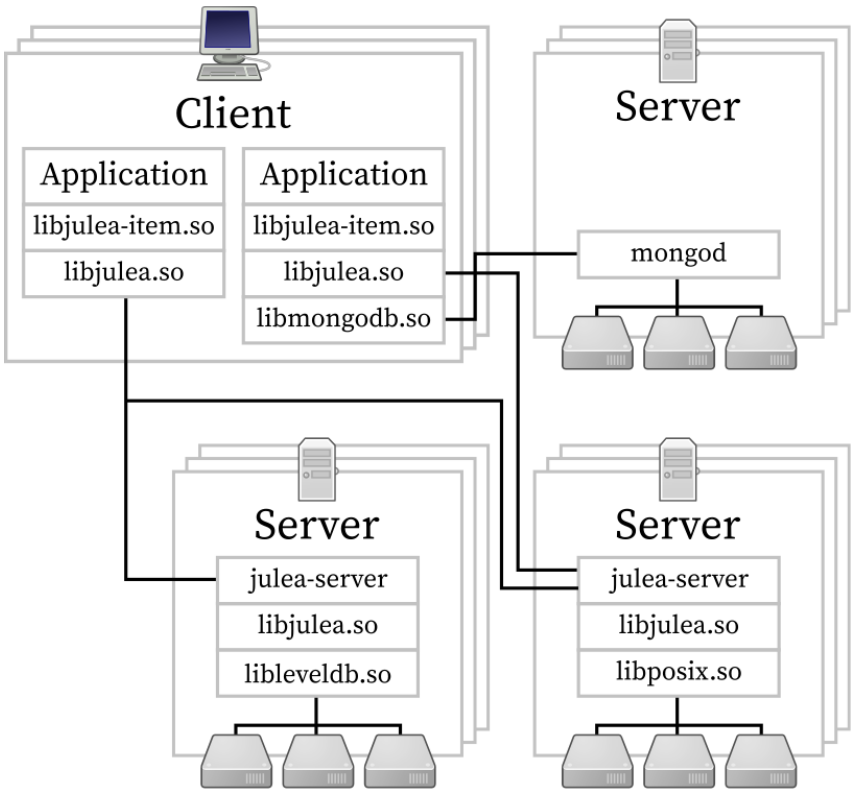

Figure 2.1.: Architecture of Julea with different application configurations [Kuh17]

2.3.1. Client-Server Model

JULEA differentiates between client and server backends:

Client backends are communicating directly with a server running the application itself like MongoDB. In the upper right of figure 2.1, it can be seen an active server handling a MongoDB daemon while the client is communicating straight with the daemon.

Server backends are provided by using the included `julea-server` application which runs a daemon on the server to provide a network socket for a JULEA client application. This daemon behavior can be seen in the bottom of figure 2.1 for the key-value store backend `leveldb` and the object-store backend `posix`.

For the communication between a JULEA client application and a julea-server daemon, the internal JMessage abstraction is used. This abstraction allows to have interchangeable network implementations or even integrate another network abstraction library.

2.3.2. Object-Store Backend

The object-store backend uses a OOP centralized design of a data structure with for the user hidden fields. The reference of this structure is then used for all further function calls to manipulate the object. The API of the object-store backend consists of the following function calls:

```
1 void j_object_create (JObject* object, JBatch* batch)
2 void j_object_delete (JObject* object, JBatch* batch)
3 void j_object_read (JObject* object, gpointer data, guint64
    ↪ length, guint64 offset, guint64* bytes_read, JBatch*
    ↪ batch)
4 void j_object_write (JObject* object, gconstpointer data,
    ↪ guint64 length, guint64 offset, guint64* bytes_written,
    ↪ JBatch* batch)
5 void j_object_status (JObject* object, gint64*
    ↪ modification_time, guint64* size, JBatch* batch)
```

Listing 2.1: JULEA object-store backend API

The create function is by design mandatory to be called before writing to an object, even though a backend implementation may ignore this call, some do need to setup an object first before a write can occur. The behavior if an object gets deleted and a client tries to write or read from it, is as of now undefined and should be avoided.

The read and write operations provide an interface similar to common Input/Output libraries like POSIX, with the exception that every function of the object backend is non blocking and will be scheduled within a batch data structure.

Line 5 of listing 2.1 can be used to retrieve basic metadata of the object like modification time and size.

2.3.3. key-value store backend

The key-value store backend can be used to store arbitrary bson tree metadata or data into a KVS (key-value store). It is considered to be used to save small amounts of data since the object-store backend already provides a efficient interface to operate on large datasets.

The KVS backend provides just three function calls to operate on key-value pairs:

```
1 void j_kv_put (JKV* kv, bson_t* value, JBatch* batch)
2 void j_kv_get (JKV* kv, bson_t* value, JBatch* batch)
3 void j_kv_delete (JKV* object, JBatch* batch)
```

Listing 2.2: JULEA key-value store backend API

The put operation will either create the new key and save its value or updates the value of the specified key. The KVS backend does not allow partial updates of the value of a key. If a key does not exists the get function gives back an empty bson tree.

The KVS backend also provides a way to iterate over all keys in a given namespace or all keys with a specific prefix in that said namespace.

3. Procedure

In this chapter, the design process of the structured metadata interface of JULEA and its reference implementation with the help of SQLite3 are described.

3.1. Design

Since JULEA uses many aspects of OOP (object oriented programming) like the polymorphic and context sensitive objects [Mey88, pp. 467], UML (unified modeling language) is chosen to create a model of the new SMD (Structured MetaData) interface [Boo05]. Before going deeper into the actual interface design, I will describe the basic concept that the SMD interface will follow.

3.1.1. Concept

The SMD backend operates on namespaces. For each namespace, the user has to provide a scheme declaration. This declaration is mandatory to ensure that the used underlying database engine can optimize inserts, gets and searches on that namespace.

The workflow is outlined in figure 3.1. The application provides one or more namespace schemes and then can perform metadata operations on that namespace.

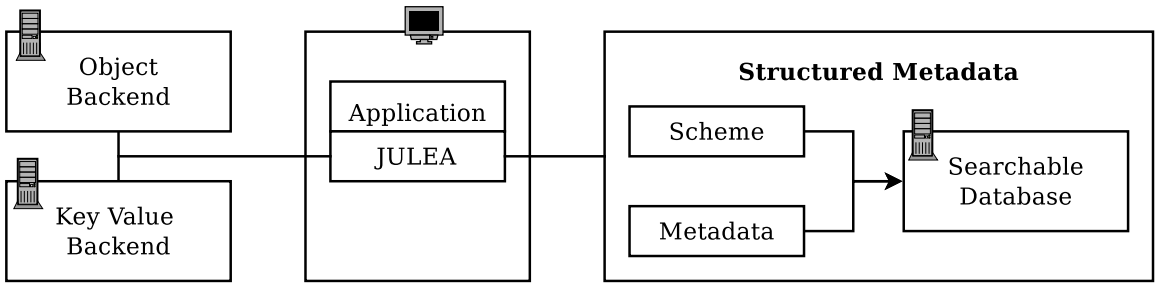

Figure 3.1.: Structured Metadata Concept

3.1.2. Application Programming Interface

The API (Application Programming Interface) of SMD continues the OOP approach of JULEA's interfaces. Therefore, three new classes which will later be used to interact with the SMD backend are introduced.

Namespace Scheme

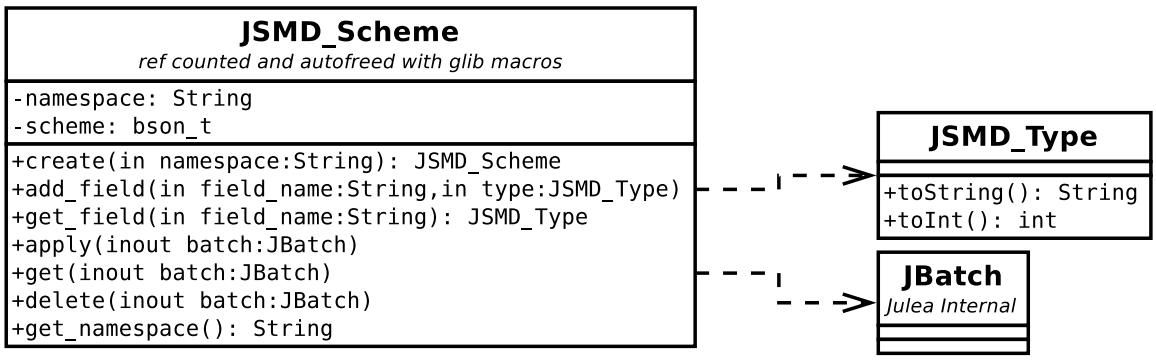

Figure 3.2.: UML class diagram for Namespace scheme api

A `JSMD_Scheme` object consist of its namespace name and a scheme definition with the help of a bson tree. As shown in figure 3.2, the user can differentiate between two sets of methods:

`add_field` and `get_field` are used to manipulate the scheme definition. Those methods ensure that only valid `JSMD_Types` are used as field types for the scheme and that there are no duplicate fields in the definition.

`apply`, `get` and `delete` prepare the `JOperations` and other requirements of the batch system of JULEA and do the memory management necessary to complete the given operation.

Metadata Object

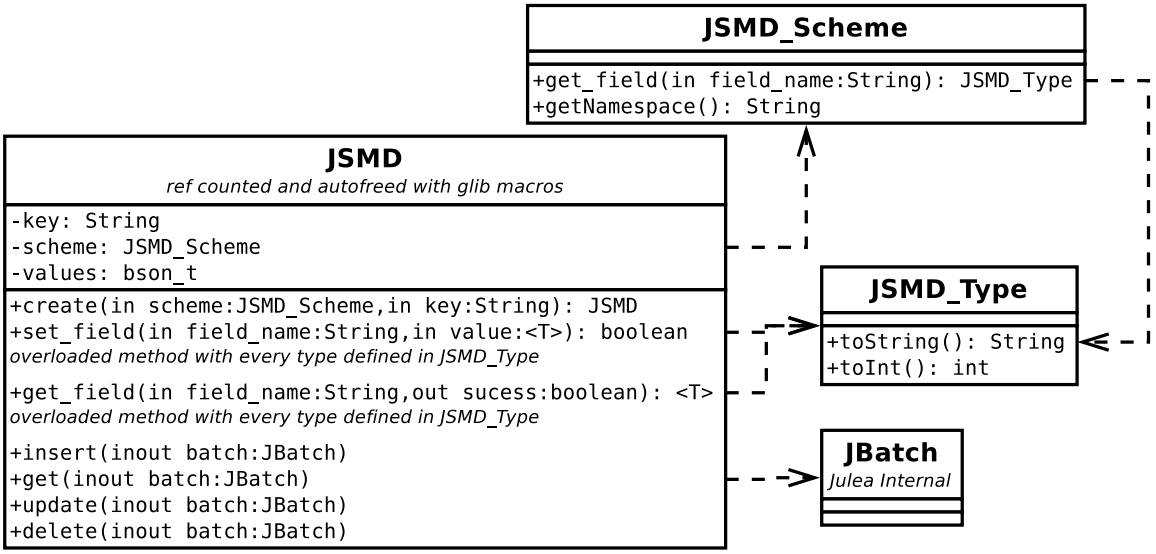

Figure 3.3.: UML class diagram for metadata object API

The metadata object class holds three data field, hidden from the client, to manage its state. The key is a unique identifier in the namespace the object operates on. It also keeps a reference to the scheme object. The metadata values are saved in a bson tree data structure.

`JSMD` offers overloaded methods to either set or get metadata fields in its internal bson tree. The `set_field` and `get_field` operations should also validate with the help of the scheme reference if a given field name exists and the correct `JSMD_Type` is used.

The `get_field` method also provides a call by reference parameter which indicates if the operation succeeded. If a field does not exists in the scheme or the bson tree, if a reference is provided, the `get_field` method will write false into the boolean reference.

The last four methods of `JSMD` manage the insertion of the appropriate JOperations into the given batch data structure and also do the mandatory memory management of it.

Search Context

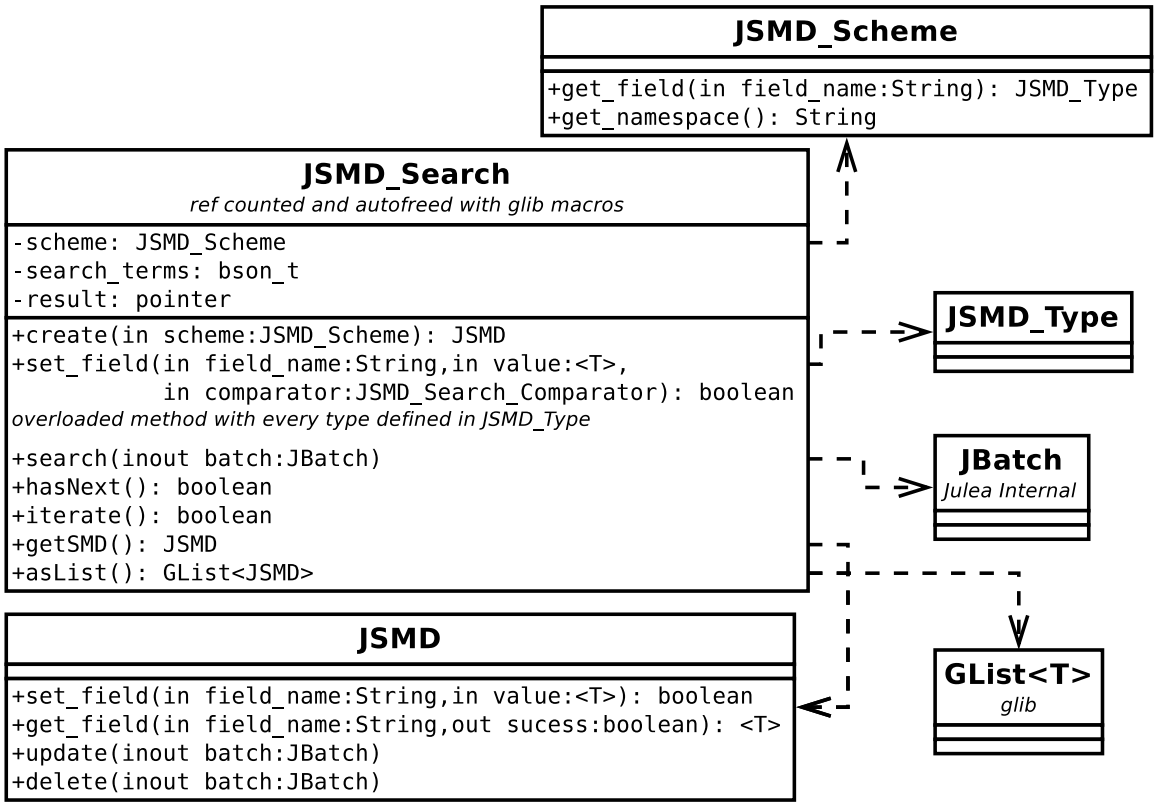

Figure 3.4.: UML class diagram for metadata search API

In figure 3.4 is the proposed search API design for the SMD client API.

The search context consists of a collection of 3-tuple out of `field_name`, `comparator` and `value` which will be, as most of its other complex data structures, be organized in a bson tree. The `set_field` operation checks if for the given `field_name` exists an entry in the scheme and the correct type is used. It also needs to be checked if the type of the field is searchable at all, for example arrays and binary data cannot be filtered by most backends.

After the search has been executed by the batch system of JULEA the search API offers two ways of getting access to the data:

The iterate operation in combination of `getSMD` can be used in a loop to access all the data returned from the backend for the given search result.

The search context also allows to get the results as a `GList` of metadata objects.

3.1.3. SMD Backend API

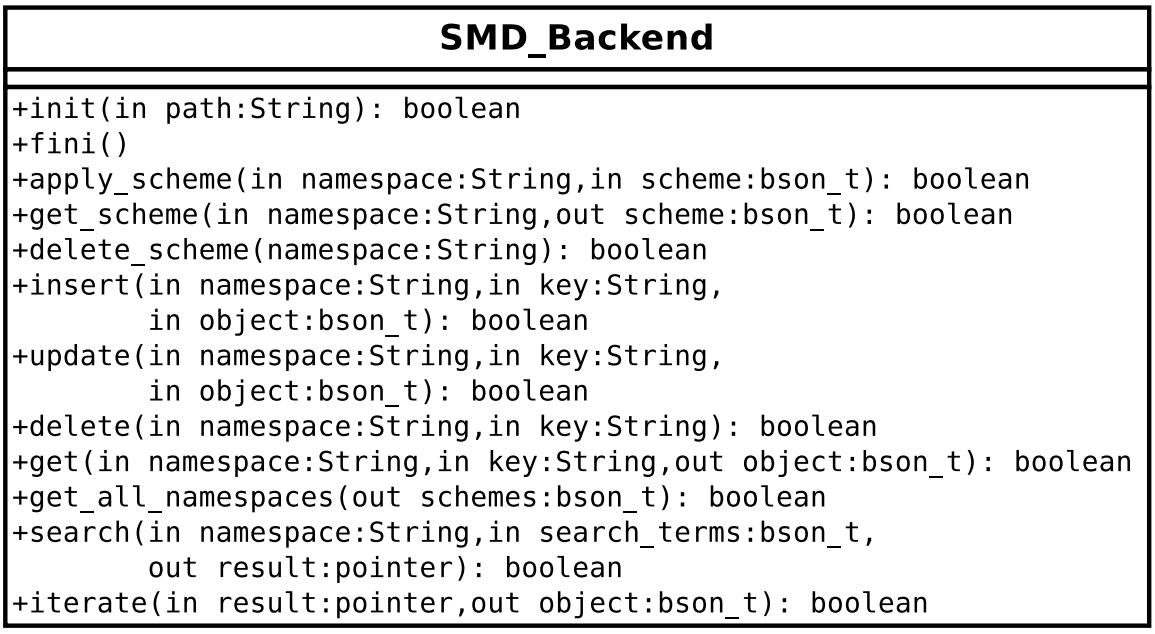

Figure 3.5.: UML class diagram for SMD backend API

The `SMD_backend` class defines the set of operations a backend must offer for the SMD client and server API to be functional.

The Operation `apply_scheme` is used to allow the backend to setup necessary data structures in its used technology. The backend is also required to persistently save this information so a following call to `get_scheme` returns the bson tree scheme information. If a namespace is requested to be deleted it should no longer return a valid result on the call of `get_scheme`. However it is the freedom of the backend how to handle the deletion of the metadata objects by either deleting and cleaning it up or just marking it as deleted for future overwrites.

The `insert` operation is required to ensure, that the given bson tree metadata object only contains valid field names and values for the given namespace. If the given key already exists in the namespace, `insert` should do nothing and return false, to inform the client API that a metadata object with that key is already present.

`update` should behave just like `insert`, with the exception, if a key already exists, the backend should use an appropriate and efficient way to set the changed values for the metadata object in its storage. If a given key does not exists, `update` should insert the provided metadata object and return true on completion.

On deletion of a metadata object with the `delete` operation, the `get` function should

no longer return a valid metadata object for the deleted key. As with the deletion of namespaces, the backend can decide how to handle the deletion process.

`get_all_namespaces` should return all non deleted active namespaces, that are managed by this backend. In a distributed environment the backend only needs to return the namespaces it is managing and its up to the client API to request the other metadata server for their namespaces.

The search operation should operate as described in section 3.1.2 and figure 3.4.

3.1.4. Network Messages

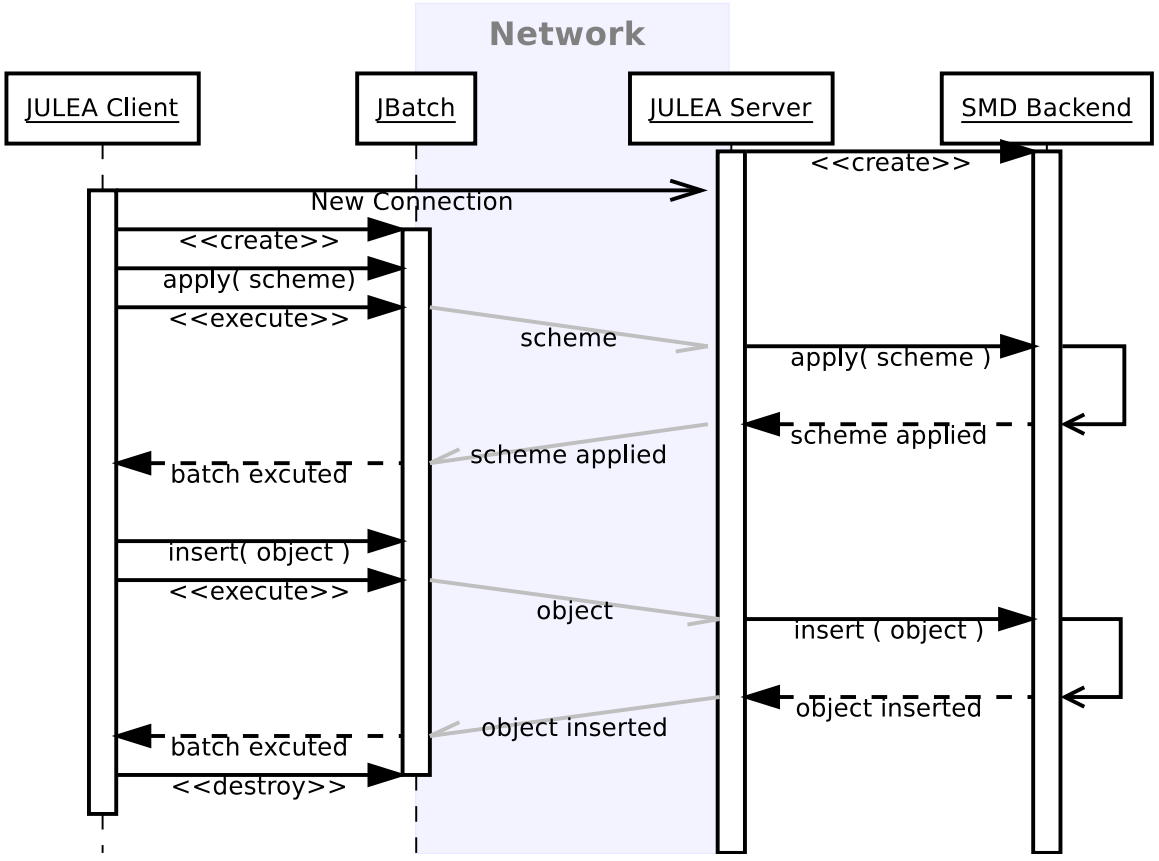

Figure 3.6.: Simple network communication of the SMD component

Figure 3.6 shows how a interaction between a client and an SMD metadata server should be implemented.

It is mandatory, that the client has to apply a scheme for a namespace before inserting metadata objects. This is due to the fact, that most searchable databases need to setup

structures for their self management, also it is needed for the validation of metadata objects before inserting them.

However it is not necessary to execute the batch before scheduling inserts, but the client application has to ensure, if it is executed in a parallel environment with multiple clients, that the apply scheme message for that namespace arrives the metadata server before any inserts, otherwise the inserts, updates will fail.

3.2. Implementation

The implementation consists of four changes that need to be done to the JULEA source code.

- Definition of the api of figure 3.5 as a new backend type
- Writing the reference implementation with SQLite3
- Writing the client operations of section 3.1.2
- Writing the server operations of section 3.1.2

3.2.1. SMD types

```
1  JSMD_REGISTER_TYPE(JSMD_TYPE_INVALID_BSON, "ERROR: type of
     ↪ bson value must be of STRING or INTEGER")
2  JSMD_REGISTER_TYPE(JSMD_TYPE_UNKNOWN, "ERROR: unknown type
     ↪ in bson value")
3  JSMD_REGISTER_TYPE(JSMD_TYPE_INTEGER, "integer")
4  [...]
5  JSMD_REGISTER_TYPE(JSMD_TYPE_UNSIGNED_INTEGER, "unsigned
     ↪ integer")
6  [...]
7  JSMD_REGISTER_TYPE(JSMD_TYPE_FLOAT, "float")
8  [...]
9  JSMD_REGISTER_TYPE(JSMD_TYPE_TEXT, "text")
10 JSMD_REGISTER_TYPE(JSMD_TYPE_DATE_TIME, "date time")
```

Listing 3.1: SMD types for usage in scheme definitions

Listing 3.1 describes how SMD handles metadata types. This is necessary to have a consistent representation of the numerous types available throughout the JULEA project.

`JSMD_REGISTER_TYPE` is a macro used to create an enumeration in C and a matching string array.

3.2.2. Reference Backend With SQLite3

SQLite3 is chosen for the reference implementation of a SMD backend. Since SQLite3 is already used as a KV backend it will not add much complexity to JULEA's building process. Another advantage is that using SQLite3 is straightforward, since it requires no upfront configuration from the user and therefore is very good to validate the API design.

To use the full potential of a relational database (RDB) the bson tree used in the API of SMD needs to be converted into SQL statements. This is done by implementing a basic query building function that iterates over the trees.

As an example, the functional principle of the `apply_scheme` and `insert` operations of the SMD backend will be explained.

Apply Scheme

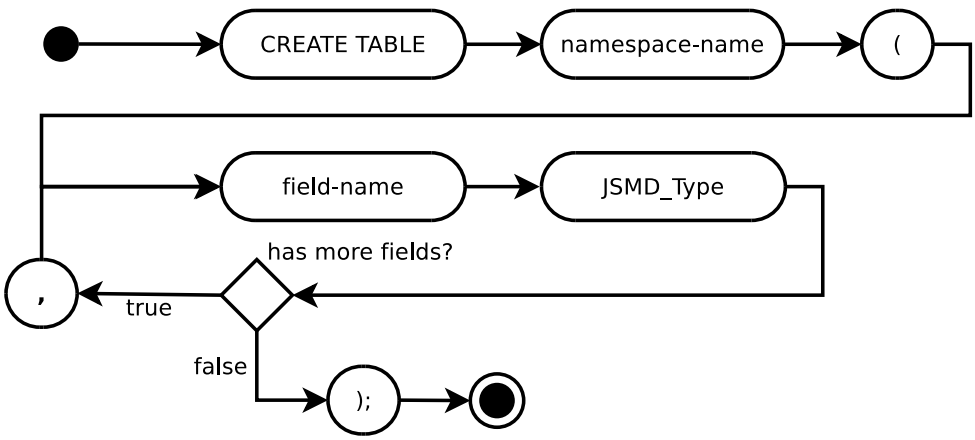

Figure 3.7.: create table builder

Figure 3.7 shows the used algorithm for building the SQL statement of applying a namespace scheme. It consists of the static fragment *CREATE TABLE* in combination of the **namespace-name**.

It then proceeds to iterate over the whole bson tree. For every bson tree leaf it inserts the **field-name** and the *JSMD_TYPE*. The builder operates on a closest match principle for the requested type and if no matching SQLite33 is present falls back to saving it as binary into the database.

The built SQL statement will be used to create the table. If the execution of it was successful it proceeds to save the bson tree as a blob into the SMD management lookup table. The management lookup table is used if a client request the scheme of a namespace.

Insert metadata object

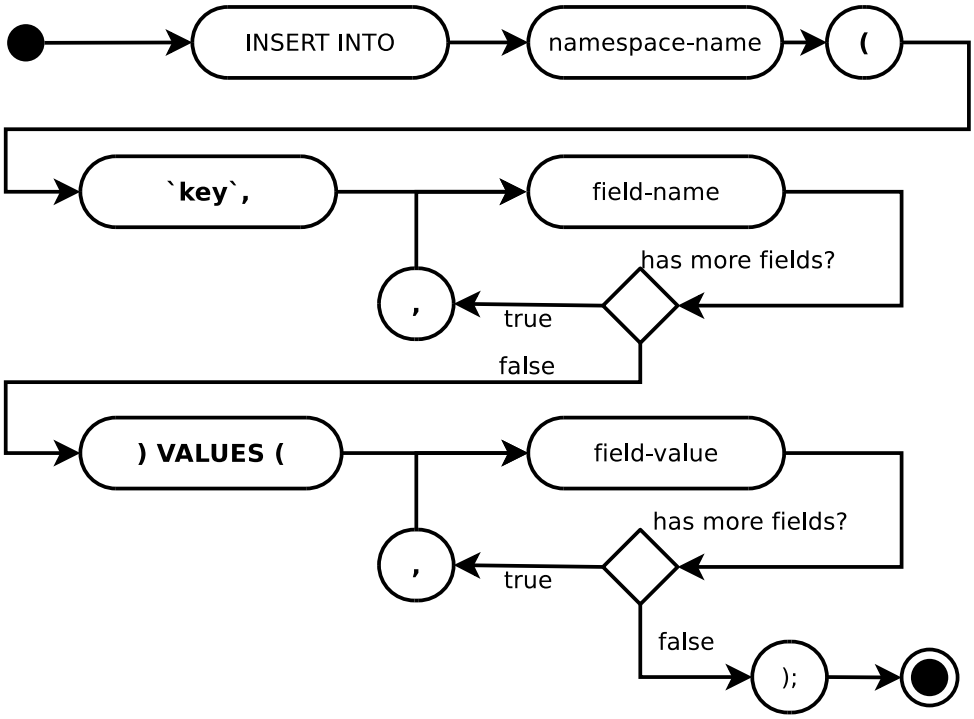

Figure 3.8.: insert builder

Preparing a SQL statement to insert into a namespace is, as can be seen in figure 3.8, already more complex than creating the namespace table structure.

The first loop inserts the field names into the query and checks if they exist in the namespace scheme definition. If the above succeeded it continues by comparing the type of the values in the metadata object with the type declarations in the scheme and inserts them into the query string.

The query gets executed by the SQLite3 engine and the return values are checked. If for example the key was already present in the namespace it sets the return value of the insert

to false as described in section 3.1.3.

4. Evaluation

In this chapter, I will introduce a way to measure the performance of the newly implemented SMD client, server and backend. I then proceed to compare the null and sqlite backends of the smd backend type and kv backend type to get an understanding of the performance and overhead of SMD.

4.1. Benchmarking

JULEA already provides a framework for measuring operations per second and throughput of data. The JULEA benchmark suite extended for all major operations of the namespace scheme and metadata object API.

- namespace API
  - apply
  - delete
  - get
- metadata object API
  - insert
  - update
  - delete
  - get

4.1.1. Setup

The following benchmark runs are done with a Lenovo Thinkpad T480 with an intel i5-8250U processor and 8GB of DDR4 memory in single channel mode. Further hardware details, such as applied CPU microcode patches, can be found in Appendix A.

To ensure a consistent performance, the CPU dynamic frequency scaling is disabled and fixed at the maximum frequency of 3.4GHz. To further reduce fluctuations of the processor, all powerlimits, which are transparent to the linux kernel, are set to their maximum.

The used Linux kernel version is 5.1.2 as compiled by the arch linux distribution.

```
1  sudo tee
     ↪ /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
     ↪ <<< "performance"
2  sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_*_freq
     ↪ <<< "3400000"
3  sudo tee /sys/devices/system/cpu/cpu*/cpufreq/
     ↪ energy_performance_preference <<< "performance"
4
5  # Set Timings
6  sudo tee
     ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:1/
     ↪ constraint_1_time_window_us <<< "10000"
7  sudo tee
     ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
     ↪ constraint_1_time_window_us <<< "10000"
8  sudo tee
     ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:1/
     ↪ constraint_0_time_window_us <<< "1000000"
9  sudo tee
     ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
     ↪ constraint_0_time_window_us <<< "1000000"
10 sudo tee
     ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
     ↪ intel-rapl:0:0/constraint_0_time_window_us <<< "5000"
11 sudo tee
     ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
     ↪ intel-rapl:0:1/constraint_0_time_window_us <<< "5000"
12 sudo tee
     ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
     ↪ intel-rapl:0:2/constraint_0_time_window_us <<< "5000"
```

```
13
14 WATT=50
15 # convert watt to microwatt
16 UW=$(($WATT*1000*1000))
17 # Power limit of the whole CPU
18 PACKAGE=$UW
19 # Power limit of the CPU cores
20 CORE=$UW
21 # Power limit of CPU cache, MMU
22 UNCORE=$UW
23 # Power limit of the memory
24 DRAM=$UW
25
26 # Write power limits to the exported intel rapl registers
27 sudo tee
   ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:1/
   ↪ constraint_1_power_limit_uw <<< $PACKAGE
28 sudo tee
   ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
   ↪ constraint_1_power_limit_uw <<< $PACKAGE
29 sudo tee
   ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:1/
   ↪ constraint_0_power_limit_uw <<< $UW
30 sudo tee
   ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
   ↪ constraint_0_power_limit_uw <<< $UW
31 sudo tee
   ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
   ↪ intel-rapl:0:0/constraint_0_power_limit_uw <<< $CORE
32 sudo tee
   ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
   ↪ intel-rapl:0:1/constraint_0_power_limit_uw <<< $UNCORE
33 sudo tee
   ↪ /sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/
   ↪ intel-rapl:0:2/constraint_0_power_limit_uw <<< $DRAM
```

Listing 4.1: commands for maxing out the cpu performance under linux

4.1.2. Results

The benchmark was run three times and the average of all runs are taken for the evaluation. The raw benchmark data can be seen in section B.3. The runs were scheduled with a shell script, that also generated the averages and is listed in section B.1 The visualization was done in gnuplot with the scripts found in section B.4

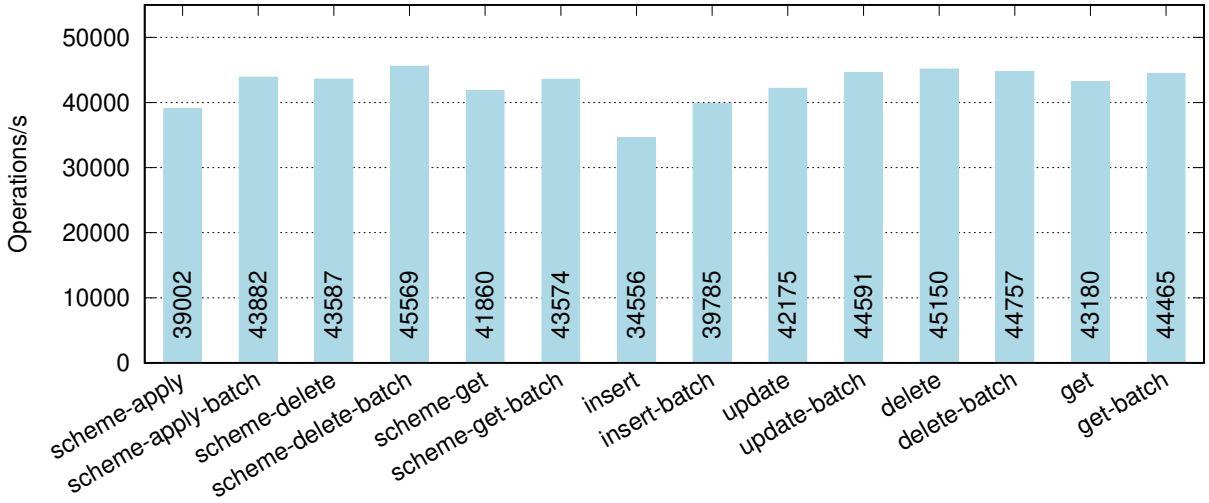

Figure 4.1.: Benchmark null smd backend

Figure 4.1 shows the upper bound for the operations per second of the smd backend. The client and server were run on the same machine. In figure 4.2 the achieved operations per second of the SQLite3 reference implementation are plotted.

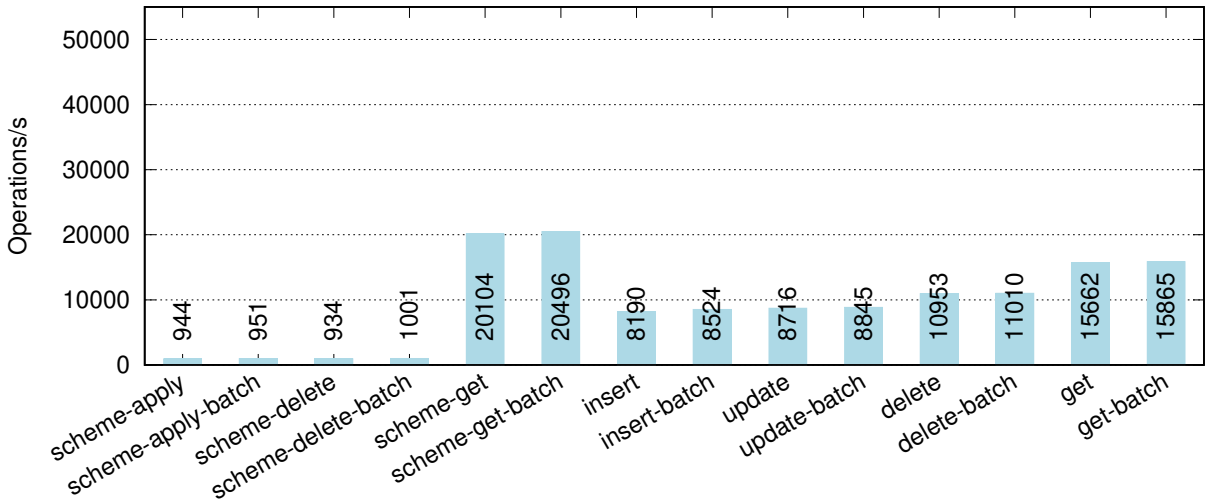

Figure 4.2.: Benchmark sqlite smd backend

The namespace apply and delete operations perform the worst in the SQLite3 backend. A

possible reason for this behavior is, that on creation of a new table SQLite3 has to grow the database file and preallocate storage for the table structure and its data. In contrast to apply scheme, the insert operation with 8500 OP/s performs pretty decent. Since its more common in a database to insert, update, delete or get table entries than creating new tables, it is not surprising that SQLite3 tradeoffs create table performance in favor of managing table entries.

Compared to the kv SQLite3 backend in figure 4.3, the overhead of the query builder used by the SQLite3 backend, seems to be much less than expected, with only a difference of 1500 OP/s (smd- insert / kv-put).

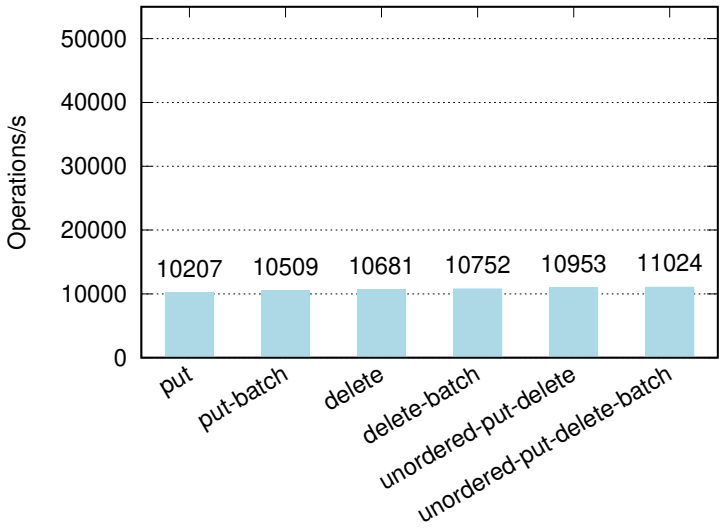

Figure 4.3.: Benchmark sqlite kv backend

4.1.3. API Overhead Evaluation

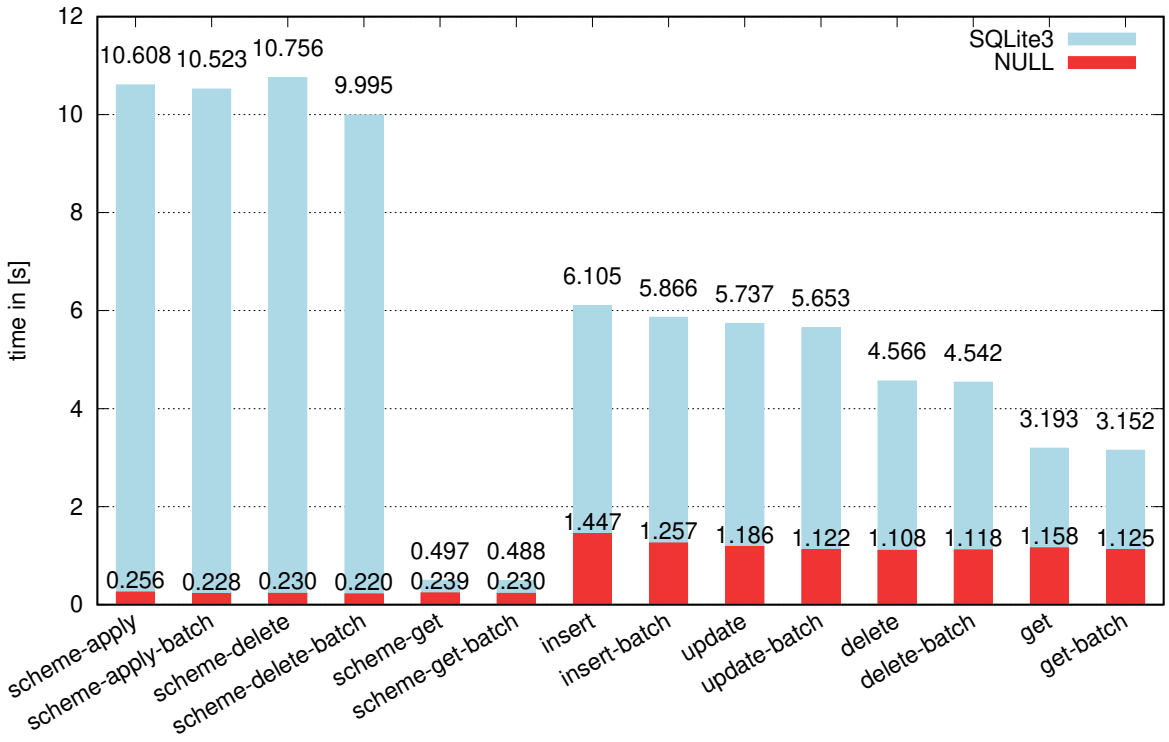

Figure 4.4.: API Overhead SMD

Figure 4.4 shows the difference of elapsed benchmark time between the SQLite3 (blue) and NULL backend (red). The measurement includes the time spend in the client and server API, since the call in the null backend also go through the entire network stack of JULEA.

The insert, update and delete operations spend nearly 30 % of their time in functions responsible for the communication between client and server. This performance loss is expected, because of the polymorphic design we need additional memory management for the wrapping data structs. Another aspect is that we need to package the client data into a JMessage and extract it at the server. A SQL database which already comes with a optimized server daemon might lower the performance loss, because less time is spend in JULEA's own wrapping libraries.

5. Summary, Conclusion, Future Work

In this chapter, a summary of the thesis and its outcome will be given. Also an outlook is presented to what needs to be done next to further increase the value of the newly introduced SMD backend.

5.1. Summary

This thesis introduced a new concept of storing user defined metadata for HPC storage applications. In contrast to other solutions, like the ones mentioned in Section 1.2, the SMD backend for JULEA uses a polymorphic design, instead of a fixed database setup.

The interface of the SMD backend was designed with the help of concepts from OOP (object oriented programming) and UML (unified modelling language) to reduce the possibility of API/ABI breaks in future iterations.

Section 3.2 implemented the concept and design of Section 3.1. To test its applicability a reference implementation of a SMD backend was developed with the help of SQLite3. A set of automated test cases were built to help others implement their own backends and ensure they follow the API design of SMD.

Chapter 4 initiates a reproducible way to measure the performance of a given backend and further provides a testing environment to ensure the correctness of an SMD backend implementation.

5.2. Conclusion

To answer the question, if the newly designed and implemented SMD backend for JULEA, is a suitable candidate to solve the metadata problem of todays HPC storage cluster systems, further work has to be done.

The polymorphic design of JULEA allows it to adapt to new object, key-value and database technologies for its backends. This is a major benefit compared to other solutions, since with this design it is able to further increase its performance, without breaking older applications build with JULEA.

A drawback of this design is the introduced overhead of managing the JULEA objects and states. It would need further research and benchmarks to investigate if the performance loss of this approach outweighs its flexibility and practicality.

5.3. Future work

With the basic functionality set, SMD now needs a implementation of the in section 3.1 proposed search API. It will also need to be evaluated, if the API overhead found in section 4.1.3 can be reduced with more sophisticated memory management and/or algorithms.

Implementing more SMD backends with, for example PostgreSQL or MongoDB, would be beneficial. These two database technologies have builtin cluster, replication and scale options, which would be interesting to examine.

The currently builtin HDF5 connector of JULEA also needs to be extended, to use the SMD API for the attributes of HDF5. This would allow, to test the SMD API with real work application built on HDF5.

Also JULEA needs internal support for various data types, for example 128bit float values. This would ensure a consistent handling of all the data types supported by SMD throughout the project.

In summary, to fully evaluate the proposed concept and design, the above mentioned additions need to be made.

Bibliography

[Boo05] Grady Booch. *The unified modeling language user guide*. Pearson Education India, 2005.

[Bra19] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[BS94] Francis P Bretherton and Paul T Singley. Metadata: A user's view. In *Seventh International Working Conference on Scientific and Statistical Database Management*, pages 166–174. IEEE, 1994.

[dkr19] Dkrz storage poster. `https://www.dkrz.de/en-pdfs/en-poster/en-poster2016/en-Poster_DKRZ_HPSS_DE.pdf?lang=de`, 2019. [Online; accessed 16-May-2019].

[ext19a] Ext4 extent table. `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Extent_Tree`, 2019. [Online; accessed 16-May-2019].

[ext19b] Ext4 inode table. `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Inode_Table`, 2019. [Online; accessed 16-May-2019].

[FHK+11] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[hdf19] Hdf5 file format specification version 3.0. `https://portal.hdfgroup.org/download/attachments/52627880/HDF5_File_Format_Specification_Version-3.0.pdf?api=v2`, 2019. [Online; accessed 16-May-2019].

[Kuh17] Michael Kuhn. Julea: A flexible storage framework for hpc. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, pages 712–723, Cham, 2017. Springer International Publishing.

[lus17] Introduction to lustre architecture. `http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf`, 2017. [Online; accessed 16-May-2019].

[Mey88] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[MGR03] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, Aug 2003.

[OWR+06] Ron A Oldfield, Lee Ward, Rolf Riesen, Arthur B Maccabe, Patrick Widener, and Todd Kordenbrock. Lightweight i/o for scientific applications. In *2006 IEEE International Conference on Cluster Computing*, pages 1–11. IEEE, 2006.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[TBD+17] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. Someta: Scalable object-centric metadata management for high performance computing. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*, pages 359–369. IEEE Computer Society, 2017.

[Tho17] Lars Thoms. *Suitability Analysis of Object Storage for HPC Workloads*. Bachelor's thesis, Universität Hamburg, 2017.

[WBM+06] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

Appendices

List of Figures

List of Listings

List of Tables

A. Hardware setup

A.1. CPU

```
vendor_id   : GenuineIntel
cpu family  : 6
model       : 142
model name  : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
stepping    : 10
microcode   : 0x9a
cpu MHz     : 3400.400
cache size  : 6144 KB
physical id : 0
siblings    : 8
cpu cores   : 4
apicid      : 7
initial apicid  : 7
fpu     : yes
fpu_exception   : yes
cpuid level : 22
wp      : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep
   ↪ mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
   ↪ sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
   ↪ constant_tsc art arch_perfmon pebs bts rep_good nopl
   ↪ xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq
   ↪ pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3
   ↪ sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
   ↪ popcnt tsc_deadline_timer aes xsave avx f16c rdrand
   ↪ lahf_lm abm 3dnowprefetch cpuid_fault epb
   ↪ invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi
   ↪ flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1
   ↪ avx2 smep bmi2 erms invpcid mpx rdseed adx smap
   ↪ clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves
   ↪ dtherm ida arat pln pts hwp hwp_notify hwp_act_window
   ↪ hwp_epp flush_l1d
```

```
bugs        : cpu_meltdown spectre_v1 spectre_v2
   ↪ spec_store_bypass l1tf mds
bogomips    : 3601.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

A.2. Memory

```
Handle 0x0004, DMI type 17, 40 bytes
Memory Device
    Array Handle: 0x0003
    Error Information Handle: Not Provided
    Total Width: 64 bits
    Data Width: 64 bits
    Size: 8192 MB
    Form Factor: SODIMM
    Set: None
    Locator: ChannelA-DIMM0
    Bank Locator: BANK 0
    Type: DDR4
    Type Detail: Synchronous Unbuffered (Unregistered)
    Speed: 2400 MT/s
    Manufacturer: SK Hynix
    Serial Number: 5243A758
    Asset Tag: None
    Part Number: HMA81GS6AFR8N-UH
    Rank: 1
    Configured Memory Speed: 2400 MT/s
    Minimum Voltage: Unknown
    Maximum Voltage: Unknown
    Configured Voltage: 1.2 V
```

B. Benchmark

B.1. Benchmark script

```
#!/bin/bash

LC_NUMERIC=en_US.UTF-8

backend=$1
runs=3

for (( runnr=1; runnr<=runs; runnr++ ))
do
  killall -9 julea-server
  rm -rf /tmp/julea
  ./scripts/benchmark.sh --path="/smd" --machine-readable
    ↪ --machine-separator="," >
    ↪ benchmark-${backend}-smd-r${runnr}.csv
  killall -9 julea-server
  rm -rf /tmp/julea
  ./scripts/benchmark.sh --path="/kv" --machine-readable
    ↪ --machine-separator="," >
    ↪ benchmark-${backend}-kv-r${runnr}.csv
done

# Strip head labels from csv

for (( runnr=1; runnr<=runs; runnr++ ))
do
  tail --lines=+2 benchmark-${backend}-r${runnr}.csv >
    ↪ benchmark-${backend}-smd-r${runnr}.csv.headless
  tail --lines=+2 benchmark-${backend}-kv-r${runnr}.csv >
    ↪ benchmark-${backend}-kv-r${runnr}.csv.headless
done

# calculate averages
```

```
27
28 for t in "smd" "-kv"
29 do
30   head -n 1 benchmark-${backend}${t}-r1.csv >
       ↪ benchmark-${backend}${t}.csv
31   paste -d"," benchmark-${backend}${t}-r*.csv.headless | sed
       ↪ 's/,/ /g' |  nawk -v s="$runs" '{
32     elapsed=0.0
33     operations=0.0
34     total_elapsed=0.0
35     for(i=0;i<=s-1;i++)
36     {
37       t=2+(i*5)
38       elapsed=elapsed+$t
39       t=3+(i*5)
40       operations=operations+$t
41       t=5+(i*5)
42       total_elapsed=total_elapsed+$t
43     }
44     printf "%s,%.5f,%.5f,-,%.5f\n",
         ↪ $1,elapsed/s,operations/s,total_elapsed/s
45   }' >> benchmark-${backend}${t}.csv
46 done
47
48 rm *.headless
```

Listing B.1: benchmark.sh

B.2. Averaged Data

B.2.1. Null

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	0.256397666666667	39002.1005493333	-	0.471086
scheme-apply-batch	0.227967333333333	43881.6333846667	-	0.441497
scheme-delete	0.229531333333333	43587.0994806667	-	0.462642666666667
scheme-delete-batch	0.219577333333333	45569.4266376667	-	0.446008666666667
scheme-get	0.238917333333333	41860.073015	-	0.682392666666667
scheme-get-batch	0.229831333333333	43573.534526	-	0.682489333333333
insert	1.447409	34555.792704	-	1.44748333333333
insert-batch	1.25744633333333	39785.3848883333	-	1.25752766666667
update	1.18564533333333	42175.394014	-	2.40930366666667
update-batch	1.12169966666667	44591.3395813333	-	2.34069866666667
delete	1.10790833333333	45150.1851163333	-	2.37846833333333
delete-batch	1.11804	44756.9195406667	-	2.37486366666667
get	1.158027	43179.866671	-	2.407721
get-batch	1.12513066666667	44464.7921556667	-	2.37073266666667

Table B.1.: benchmark-null-smd.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
put	4.57802866666667	43698.6944356667	-	8.66388966666667
put-batch	4.29361966666667	46585.3064543333	-	8.39066966666667
delete	4.34855533333333	46000.898981	-	8.61580466666667
delete-batch	4.17412466666667	47914.6506933333	-	8.453155
unordered-put-delete	4.32007233333333	46300.2476106667	-	4.32007433333333
unordered-put-delete-batch	4.185854	47780.342585	-	4.18585633333333

Table B.2.: benchmark-null-kv.csv

B.2.2. SQLite3

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	10.607793	943.520574666667	-	20.874795
scheme-apply-batch	10.522952	951.212735666667	-	20.7795543333333
scheme-delete	10.7559643333333	933.989579333333	-	21.2685026666667
scheme-delete-batch	9.994962	1000.511472	-	20.3410533333333
scheme-get	0.497474666666667	20104.3129526667	-	21.035908
scheme-get-batch	0.488211666666667	20495.5332996667	-	20.892216
insert	6.10491933333333	8190.15219033333	-	6.107885
insert-batch	5.866054	8524.37698366667	-	5.86892766666667
update	5.736914	8715.63925266667	-	11.5516423333333
update-batch	5.65318633333333	8844.61117066667	-	11.4883413333333
delete	4.56555333333333	10953.296673	-	10.4300686666667
delete-batch	4.541844	11010.338918	-	10.3913666666667
get	3.19276333333333	15662.110302	-	9.062783
get-batch	3.15198433333333	15864.8162213333	-	9.017099

Table B.3.: benchmark-sqlite-smd.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
put	19.6172156666667	10206.9897086667	-	19.6172156666667
put-batch	19.0376913333333	10508.546577	-	19.0376913333333
delete	18.7259703333333	10680.5340873333	-	18.7259703333333
delete-batch	18.6022903333333	10751.9123973333	-	18.6022903333333
unordered-put-delete	18.2597543333333	10953.2810253333	-	18.2597543333333
unordered-put-delete-batch	18.1438506666667	11024.2265616667	-	18.1438506666667

Table B.4.: benchmark-sqlite-kv.csv

B.3. Raw Data

B.3.1. Null

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	0.256748	38948.696777	-	0.467246
scheme-apply-batch	0.222071	45030.643353	-	0.431575
scheme-delete	0.223242	44794.438323	-	0.467399
scheme-delete-batch	0.212639	47028.061644	-	0.436573
scheme-get	0.240038	41660.070489	-	0.674186
scheme-get-batch	0.226611	44128.484495	-	0.661552
insert	1.422831	35141.207916	-	1.422906
insert-batch	1.298634	38501.995173	-	1.298721
update	1.171842	42667.868194	-	2.373112
update-batch	1.112070	44961.198486	-	2.309991
delete	1.080540	46273.159716	-	2.334609
delete-batch	1.086474	46020.429389	-	2.313459
get	1.146636	43605.817365	-	2.411303
get-batch	1.090713	45841.573356	-	2.313161

Table B.5.: benchmark-null-smd-r1.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	0.255609	39122.253129	-	0.470536
scheme-apply-batch	0.229678	43539.215772	-	0.444324
scheme-delete	0.230114	43456.721451	-	0.457647
scheme-delete-batch	0.225713	44304.049833	-	0.453711
scheme-get	0.235460	42470.058609	-	0.682532
scheme-get-batch	0.220972	45254.602393	-	0.676582
insert	1.435489	34831.336221	-	1.435560
insert-batch	1.228401	40703.320821	-	1.228479
update	1.184131	42225.057869	-	2.415164
update-batch	1.101602	45388.443376	-	2.314944
delete	1.105458	45230.121814	-	2.415176
delete-batch	1.161646	43042.372633	-	2.390571
get	1.170177	42728.578668	-	2.384466
get-batch	1.128338	44312.962960	-	2.402603

Table B.6.: benchmark-null-smd-r2.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	0.256836	38935.351742	-	0.475476
scheme-apply-batch	0.232153	43075.041029	-	0.448592
scheme-delete	0.235238	42510.138668	-	0.462882
scheme-delete-batch	0.220380	45376.168436	-	0.447742
scheme-get	0.241254	41450.089947	-	0.690460
scheme-get-batch	0.241911	41337.516690	-	0.709334
insert	1.483907	33694.833975	-	1.483984
insert-batch	1.245304	40150.838671	-	1.245383
update	1.200963	41633.255979	-	2.439635
update-batch	1.151427	43424.376882	-	2.397161
delete	1.137727	43947.273819	-	2.385620
delete-batch	1.106000	45207.956600	-	2.420561
get	1.157268	43205.203980	-	2.427394
get-batch	1.156341	43239.840151	-	2.396434

Table B.7.: benchmark-null-smd-r3.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
put	4.565768	43804.240601	-	8.632799
put-batch	4.352246	45953.284810	-	8.413732
delete	4.343697	46043.727267	-	8.624611
delete-batch	4.189317	47740.478937	-	8.510249
unordered-put-delete	4.341906	46062.719921	-	4.341908
unordered-put-delete-batch	4.188114	47754.191982	-	4.188116

Table B.8.: benchmark-null-kv-r1.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
put	4.492584	44517.809795	-	8.536265
put-batch	4.252275	47033.646695	-	8.354839
delete	4.278172	46748.938565	-	8.525892
delete-batch	4.159291	48085.118353	-	8.380032
unordered-put-delete	4.358977	45882.325142	-	4.358979
unordered-put-delete-batch	4.198800	47632.656950	-	4.198802

Table B.9.: benchmark-null-kv-r2.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
put	4.675734	42774.032911	-	8.822605
put-batch	4.276338	46768.987858	-	8.403438
delete	4.423797	45210.031111	-	8.696911
delete-batch	4.173766	47918.354790	-	8.469184
unordered-put-delete	4.259334	46955.697769	-	4.259336
unordered-put-delete-batch	4.170648	47954.178823	-	4.170651

Table B.10.: benchmark-null-kv-r3.csv

B.3.2. SQLite3

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	10.378016	963.575312	-	20.439579
scheme-apply-batch	10.271407	973.576454	-	20.213930
scheme-delete	10.040549	995.961476	-	20.429391
scheme-delete-batch	9.986674	1001.334378	-	20.287342
scheme-get	0.505772	19771.754862	-	20.994574
scheme-get-batch	0.505069	19799.274951	-	20.835024
insert	6.117117	8173.785134	-	6.119789
insert-batch	5.842920	8557.365153	-	5.845774
update	5.767909	8668.652713	-	11.597198
update-batch	5.667443	8822.320754	-	11.528565
delete	4.646761	10760.183276	-	10.640267
delete-batch	4.619402	10823.911840	-	10.553023
get	3.237976	15441.745090	-	9.180194
get-batch	3.197239	15638.493087	-	9.139078

Table B.11.: benchmark-sqlite-smd-r1.csv

B.4. GNUplot

```
1  set terminal postscript eps size 5,2.5 enhanced color font
     ↪ 'Helvetica,14'
2  set ylabel "Operations/s"
3
4  set key autotitle columnhead
5
6  set grid ytics
7
8  set boxwidth 0.5
9  set style fill solid
10 set datafile separator ","
11
12 set xtics rotate by 30 right
13 set bmargin 10
14
15 set yrange [0:55000]
16
17 set output 'benchmark-null-smd.eps'
18
```

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	11.053893	904.658657	-	21.739416
scheme-apply-batch	10.987474	910.127296	-	21.800338
scheme-delete	11.776398	849.156083	-	22.517244
scheme-delete-batch	10.031676	996.842402	-	20.435282
scheme-get	0.493821	20250.252622	-	21.079478
scheme-get-batch	0.476619	20981.119091	-	20.948414
insert	6.110326	8182.869457	-	6.113819
insert-batch	5.942597	8413.829846	-	5.945403
update	5.709497	8757.338869	-	11.519333
update-batch	5.637527	8869.137123	-	11.461463
delete	4.521534	11058.193967	-	10.318625
delete-batch	4.500330	11110.296356	-	10.300227
get	3.181541	15715.654772	-	9.032426
get-batch	3.141848	15914.200814	-	8.975870

Table B.12.: benchmark-sqlite-smd-r2.csv

```
19 mk_label(x)=sprintf("%.0f",x)
20
21 plot "benchmark-null-smd.csv" using 3:xticlabel(1) with
      ↪ boxes lt rgb "light-blue" notitle, \
22 '' using 0:0:(mk_label($3)) with labels rotate left offset
      ↪ 0,1 notitle
23
24 unset size
25 set terminal postscript eps size 4,2.5 enhanced color font
      ↪ 'Helvetica,14'
26 set output 'benchmark-null-kv.eps'
27
28 plot "benchmark-null-kv.csv" using 3:xticlabel(1) with boxes
      ↪ lt rgb "light-blue" notitle, \
29 '' using 0:3:(mk_label($3)) with labels center offset 0,1
      ↪ notitle
```

```
1 set terminal postscript eps size 5,2.5 enhanced color font
     ↪ 'Helvetica,14'
2 set ylabel "Operations/s"
3
4 set key autotitle columnhead
5
6 set grid ytics
7
```

Name	Elapsed	Operations	Bytes	Total Elapsed
scheme-apply	10.391470	962.327755	-	20.445390
scheme-apply-batch	10.309975	969.934457	-	20.324395
scheme-delete	10.450946	956.851179	-	20.858873
scheme-delete-batch	9.966536	1003.357636	-	20.300536
scheme-get	0.492831	20290.931374	-	21.033672
scheme-get-batch	0.482947	20706.205857	-	20.893210
insert	6.087315	8213.801980	-	6.090047
insert-batch	5.812645	8601.935952	-	5.815606
update	5.733336	8720.926176	-	11.538396
update-batch	5.654589	8842.375635	-	11.474996
delete	4.528365	11041.512776	-	10.331314
delete-batch	4.505800	11096.808558	-	10.320850
get	3.158773	15828.931044	-	8.975729
get-batch	3.116866	16041.754763	-	8.936349

Table B.13.: benchmark-sqlite-smd-r3.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
put	20.574005	9721.004734	-	40.412357
put-batch	19.495495	10258.780298	-	37.921057
delete	18.819501	10627.274336	-	37.785930
delete-batch	18.790558	10643.643472	-	37.730118
unordered-put-delete	18.343035	10903.321070	-	18.343038
unordered-put-delete-batch	18.398733	10870.313733	-	18.398735

Table B.14.: benchmark-sqlite-kv-r1.csv

```
 8 set boxwidth 0.5
 9 set style fill solid
10 set datafile separator ","
11
12 set xtics rotate by 30 right
13 set bmargin 10
14
15 set yrange [0:55000]
16
17 set output 'benchmark-sqlite-smd.eps'
18
19 mk_label(x)=sprintf("%.0f",x)
20
21 plot "benchmark-sqlite-smd.csv" using 3:xticlabel(1) with
   ↪ boxes lt rgb "light-blue" notitle, \
```

Name	Elapsed	Operations	Bytes	Total Elapsed
put	19.163727	10436.383278	-	37.538720
put-batch	18.867387	10600.301992	-	37.262547
delete	18.727022	10679.754635	-	37.638171
delete-batch	18.511475	10804.109343	-	37.489996
unordered-put-delete	18.291105	10934.276524	-	18.291106
unordered-put-delete-batch	18.091328	11055.020394	-	18.091331

Table B.15.: benchmark-sqlite-kv-r2.csv

Name	Elapsed	Operations	Bytes	Total Elapsed
put	19.113915	10463.581114	-	37.364351
put-batch	18.750192	10666.557441	-	36.993576
delete	18.631388	10734.573291	-	37.436337
delete-batch	18.504838	10807.984377	-	37.284071
unordered-put-delete	18.145123	11022.245482	-	18.145125
unordered-put-delete-batch	17.941491	11147.345558	-	17.941493

Table B.16.: benchmark-sqlite-kv-r3.csv

```
22 '' using 0:0:(mk_label($3)) with labels rotate left offset
     ↪ 0,1 notitle
23
24 unset size
25 set terminal postscript eps size 4,2.5 enhanced color font
     ↪ 'Helvetica,14'
26 set output 'benchmark-sqlite-kv.eps'
27
28 plot "benchmark-sqlite-kv.csv" using 3:xticlabel(1) with
     ↪ boxes lt rgb "light-blue" notitle, \
29 '' using 0:3:(mk_label($3)) with labels center offset 0,1
     ↪ notitle
```

C. Network packages

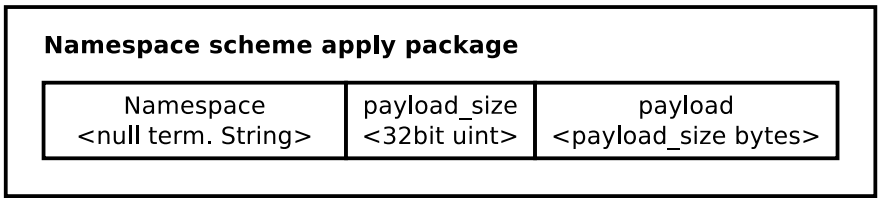

Figure C.1.: Network package scheme apply

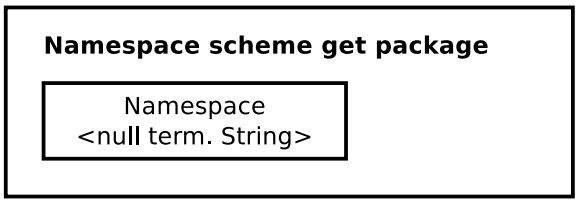

Figure C.2.: Network package scheme get

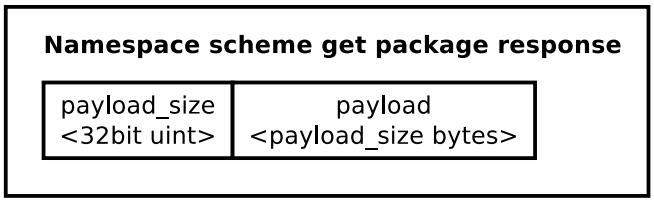

Figure C.3.: Network package scheme get response

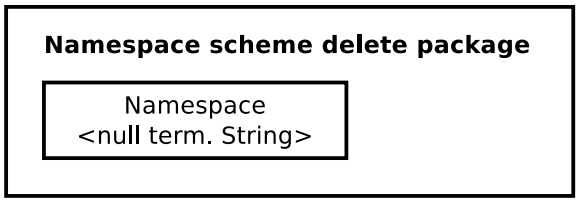

Figure C.4.: Network package scheme delete

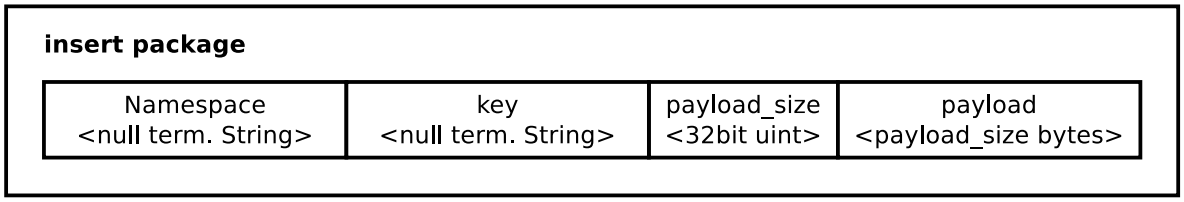

Figure C.5.: Network package metadata object insert

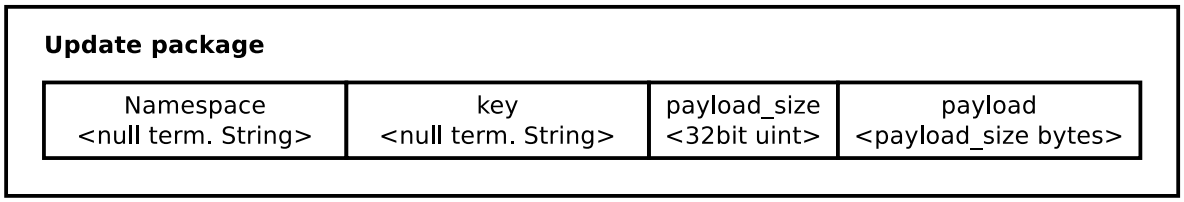

Figure C.6.: Network package metadata object update

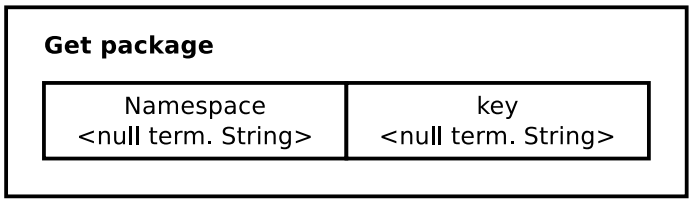

Figure C.7.: Network package metadata object get

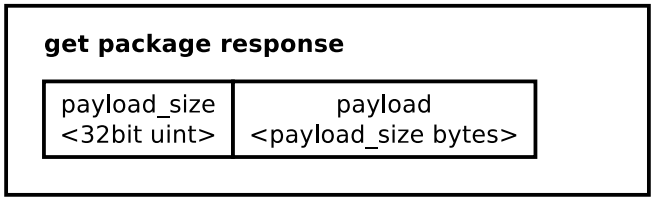

Figure C.8.: Network package metadata object get response

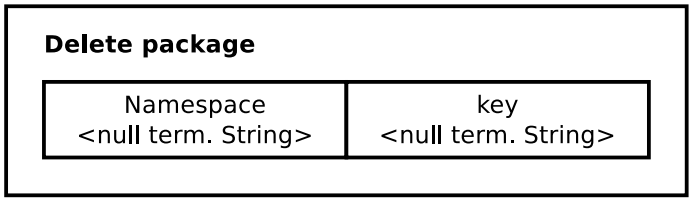

Figure C.9.: Network package metadata object delete

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Bachelor of Science Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift