University of Zurich
Department of Informatics

Dynamic and Distributed
Information Systems

Tygrstore: a Flexible Framework for High Performance Large Scale RDF Storage

Bachelor Thesis March 18, 2011

Yannick Koechlin
of Basel BS, Switzerland

Student-ID: 03-056-314
yannick@koechlin.info

Advisor: **Cosmin Basca**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/ddis

Acknowledgements

First and foremost i would like to thank my Advisor Cosmin Basca for the Faith, Time and Work he spent with me on this project. Without his help and his numerous recapitulations of some topics, i would not have achieved any working version of Tygrstore. I also would like to thank the fellow students who helped me and especially the SEAL master students in the project room, with whom i had a lot of fun. Last but not least my girlfriend Anna Ehrenklau supported me whenever she could. Thank You! I also had a lot of fun learning Python/Cython and gaining knowledge about Triplestores. Both of these topics where new to me.

Abstract

This Thesis describes the Architecture of a highly flexible Triplestore Framework. Its main features are: pluggable backend storage facilities, horizontal Scalability, a simple API and the generation of endless result Streams. Special attention has been paid on easy extensibility. First a detailed view of the architecture is given, later more details on the actual implementation are revealed. In the end two possible triplestore setups are benchmarked and profiled. It is shown that the currently limiting factors do not lie within the architecture but in the library code for the backends. In the end possible solutions and enhancements to the framework are discussed.

Zusammenfassung

Die vorliegende Arbeit beschreibt und dokumentiert den Tygrstore. Tygrstore ist ein flexibles, performantes Framework um Tripelstores zu erstellen. Die Hauptmerkmale sind auswechselbare Backend-Speicherloesungen, eine einfache API und die Möglichkeit endlose Resultate zu streamen. Tygrstore ermöglicht ausserdem die horizontale Skalierung. Es wurde ein spezielles Augenmerk darauf gelegt, alle Module einfach erweitern zu können. Zu Beginn wird die Architektur besprochen und danach die Einzelheiten der eigentlichen Implementation . Dann werden zwei mögliche Setups getestet und diese werden dann genauer analysiert. Zum Schluss werden noch Wege fuer die Weiterentwicklung vorgeschlagen.

Contents

1

Introduction

This Thesis introduces, documents and evaluates a new triplestore called Tygrstore. Besides being a triplestore which evaluates SPARQL queries, Tygrstore is also designed to serve as a foundation framework for rapid prototyping of new techniques in RDF storage. According to that, the general focus was on extensibility rather than a specific optimal behavior. Tygrstore is written in Cython [1] which permits fine grained performance optimization written in c but also enables the full benefits of Pythons dynamic nature.

Based on the TokyoTyGR triple store (used in [Basca and Bernstein, 2010]), Tygrstore implements the Hexastore model described in [Weiss et al., 2008] and [Weiss and Bernstein,]. In Section 1.1 the context of triple stores is briefly discussed as it is the base of the Motivation for this thesis. The detailed Architecture of Tygrstore is covered throughout Chapter 2. While Tygrstore can be working in the same way as TokyoTyGR, it can also easily be adapted to other storage facilities than Tokyo Cabinet. This flexibility is shown in Chapter 3 where the two main backend storage drivers are explained in detail. Further characteristics of Tygrstore are explored in the Evaluation in Chapter 4.

[1] http://cython.org/

1.1 Context

In the recent years, the semantic web became a vivid topic in research. Especially the community of graph databases was revitalized. The characteristics of RDF Schema as graph model and SPARQL as a query language unfolded new problems. Among these topics the scalability, processing speed and optimization of triple stores are still heavily discussed. RDF Storage research has concentrated on three main areas: join processing [Vidal et al., 2010], [Senn, 2010], [Kim et al., 2009], query optimization [Schmidt et al., 2010] and data-structures [Weiss and Bernstein,] , [Senn,], [Atre and Hendler,] and [Atre et al., 2010]. The combined question still is, on how these going to scale. Brewers CAP theorem [Gilbert and Lynch, 2002] clearly reveals that traditional RDBMS will not solve the problems at hand. Meanwhile the Lean Startup methodology[2] and the advance of the internet on mobile (and other platforms) originated in todays abundance of SaaS and PaaS offerings[3]. These systems in turn have also scaling needs beyond the possibilities of ACID conform RDBMS. This resulted in a new generation of DBMS, often subsumed as NoSQL stores. This typology usually includes:

- Key/Value Stores
- Distributed Key/Value Stores
- Graph Oriented Stores
- Column Oriented Stores
- Document Oriented Stores
- Map-Reduce based Systems
- Mixtures of the above

Each of these having at least a hand full of implementations[4].

So this business oriented community solves at least similar problems as the semantic web community. Where they differ is, that while the semantic web works towards a standardization of each components via the W3C, the interfaces in NoSQL world lack any there of. This leads to the many proprietary query language in existence and the fallback to (compared to SPARQL) relatively primitive data retrieval protocols (REST or JSON).

1.2 Motivation

These two communities are solving similar problems and each of these offer methods the other lacks and needs. In that sense, Tygrstore could provide a framework to help those two worlds

[2] `http://en.wikipedia.org/wiki/Lean_Startup`
[3] this includes social networks, google apps, social games etc
[4] a list can be found on `http://nosql-database.org/`

converge, while not changing any of the sides at its core. In this Thesis i show an architecture which allows us to leverage, the already highly optimized, production ready, NoSQL stores and supply them with an standardized way to access data therein. Furthermore it should be possible to rapidly prototype and evaluate new concepts in other areas, especially in query optimization and join processing.

2

Tygrstore Architecture

2.1 Architecture Overview

The core of Tygrstore consists of five main classes. The Sparql parser, the *QueryEngine*, the *Stringstore*, the *IndexManager* and the *Index*. Put together, these modules allow the execution of SPAQRL SELECT queries. Figure 2.1 shows the call hierarchy of these classes. TygrstoreServer and the shell script tygrstore.sh are simple wrapper scripts to make the *QueryEngine*s functionality available via HTTP, msgpack[1] and in a posix console. The Sparql parser is a separate Python package called cysparql. It contains Cython bindings to Rasqal 0.9.25 [2] as well as methods to save data. This package was kindly provided by Cosmin Basca.

2.1.1 Execution Model

After the query has been parsed, the Query Engine uses the Stringstore to convert the RDF Literals to their internal representation (database ID). As we will see, this database ID is usually a hash of the RDF Literal in n3 notion. The Query is also being optimized by looking up the selectivities of each variable used in the WHERE clause. Further query variables are resolved using recursion. Doing that, the *QueryEngine* operates on the Index Manager which decides on which of the available actual indices is being consulted. The *IndexManager* returns generators for all Triple Patterns containing the current variable. These generators yield database Ids which then are merge-joined by the *QueryEngine*. The resulting intersection defines the space in which the Variables solutions can possibly reside. For each of these possible solution a recursion step is done and the remaining variables are solved. This means that results are being generated in a constant manner and can be streamed from the client.

[1] `http://msgpack.org`
[2] `http://librdf.org/rasqal/`

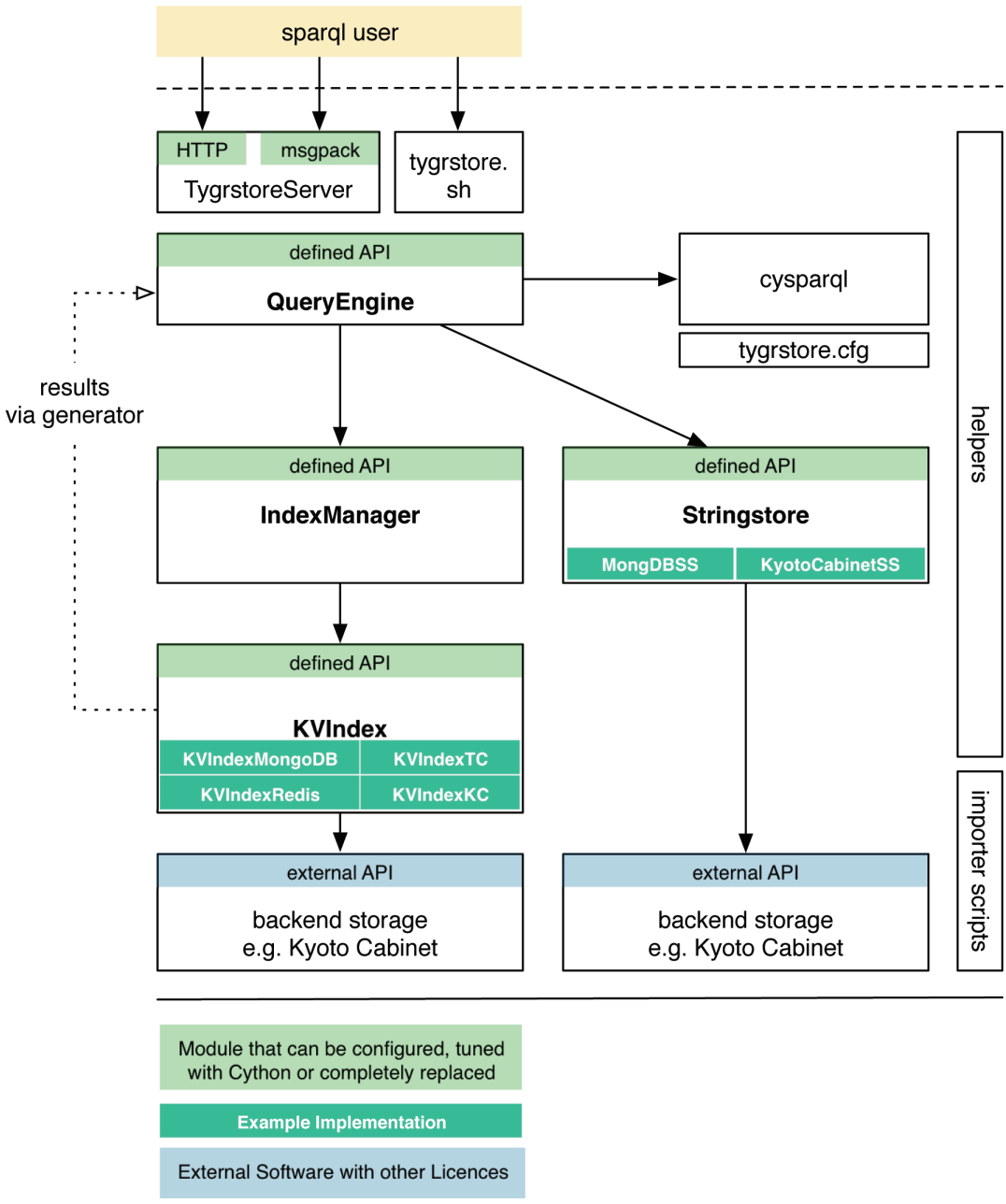

Figure 2.1: Tygrstore Architecture Overview.

2.2 Index Manager

The *IndexManager* instance provides methods to maintain and lookup indexes. It serves as the central dispatcher and single point of entry to the indexes. Thus the *QueryEngine* should never call on indexes directly but always trough *IndexManager*.

IndexManager API methods:

General Remark: ttriple means an instance of the Tygrstore Triple class. A natural is one character of the natural ordering, so usually 's' or 'p' or 'o'.

- `selectivity_for_triple(statement_as_tuple)` Where statement is a tuple filled with Database Keys or None in the order of `self.naturals`. It proxies the request to the correct Indexes and returns an integer with an estimation on how many triples match the pattern. If multiple Indexes can be used it returns the minimal value.
- `ids_for_triple(statement_as_tuple, searched_natural)` returns a generator which yields database keys matched by the triplepattern in `statement_as_tuple`. `searched_natural` is a single character denoting the position of the unbound variable which should be resolved. E.g. if the triplepattern (`statement_as_tuple`) is `('31337',None,'38317',None)` and the searched_natural is 'c' the generator will yield all contexts which have a triple in which the subject and the objects correspond to the above keys.
- `ids_for_ttriple(ttriple, var)` returns a generator which yields database keys matched by the triplepattern in ttriple. Var is a string with the name of the SPARQL variable but without the leading ? or $. `ids_for_triple` should be prefered over this method, since it is more generic.
- `add_to_all_indexes(statement_as_tuple)` used to add a statement to the store. This should only be used for small amounts of statements. Bulk insertion should be done via a specialized importer script which talks directly to the backend. See Section 2.8
- `close()` calls close() on all indexes
- `len()` returns the total number of statements in the store

IndexManager Internals:

The *IndexManager* instance keeps a list of all indexes as well as a mapping Dictionary to find suitable indexes. Initialized in the `__init__()` function of *IndexManager* this simple data-structure is the foundation of the *IndexManager*:

```
'''here we instanciate one index class for every permutation of the naturals string.
we also add a (tuple as) key for every sub-tuple (eg. s,  and s,p and s,p,o  for s,p,o,c
    ) with the index
```

```
as value. they go into unique_indexes and indexes
'''
def build_indexes(self):
    #build a list of tuples of all permutations of
    #e.g. 'spo' -> ('s','p','o'), ('s','o','p')  etc.
    for p in permutations(self.naturals):
        #the name (in the index known as internal_ordering). e.g. ('s','o','p') -> "sop"
        index_name = "".join(p)
        #instanciate a new index.
        #equal to e.g.: an_index = KVIndexTC(config_instance, name="spo")
        an_index = self.index_class(self.config, name=index_name)
        #maintain a list of all indexes in unique_indexes
        self.unique_indexes.append(an_index)
        #generate the keys for the mapping dict self.indexes and set the index as value
        #e.g. for 'spo'  -> ('s',), ('s','p'), ('s','p','o') as keys
        for i in range(1,len(p)+1):
            self.indexes[p[0:i]] = an_index
    #for (None), (None,None,...) set any index
    for none_tuple_length in range(1,len(p)+1):
        none_tuple = (None,) * none_tuple_length
        #if we have no given vars it does not matter which index we take
        self.indexes[none_tuple] = an_index
```

When the *QueryEngine* calls the `ids_for_ttriple` method, the correct index is internally evaluated with the `index_for_ttriple` method. This method puts together the already solved variables and determines which of the unsolved part in the triple-pattern we search for (natural). With that information at hand it can look up the index in the Dictionary.

```
'''return the coresponding index for a statement'''
@memoized
def index_for_ttriple(self, ttriple, var):
    #get all solved naturals in the triple.
    idx_name = tuple(i for i in self.compress(self.naturals, ttriple.ids_as_tuple()))
    # add the natural where the variable we are searching for is
    idx_name += (self.naturals[ttriple.variables_tuple.index(var)],)
    #look the index up in the dict
    return self.indexes[idx_name]

'''return the coresponding indexes for a statement'''
@memoized
def indexes_for_tuple(self, triple):
    #generator for all possible key combinations
    naturals = self.compress(self.naturals, triple)
    #generator for all permutations
    all_indexes = permutations(x for x in naturals)
    #list of all
    return [ self.indexes[i] for i in all_indexes  ]
```

Instead of caching this method with memoized it should be re-factored and the information about the naturals should be pushed to the *Triple* class. This was omitted as the Triple wrapper class is a later addition to the framework and its interface being still unstable.

All other methods are straight forward implementations of their respective task. Only `self.update_only_one` has a slight effect in that it signals that all indexes in `self.unique_indexes` are possibly the same instance of a *KVIndex*. The reasons for this existence are further described in Section 3.2.2.

2.3 Storage Backends

To store the actual indexes of the the triple store, a storage backend is needed. These Backends can be anything, from optimized in-memory or on-disk data-structures to network accessible Databases. The interface to implement such a backend is kept to a bare minimum. This allow easy and fast development of new Adapters. Described in more detail in 2.3.1, a functional read only index could be implemented by providing just 3 API methods. Namely the `ids_for_triple` ,`selectivty_for_triple` and `__init__`. None the less it is recommended to implement the whole API described below. Currently there are Adapters for Tokyo Cabinet, Kyoto Cabinet, Redis and MongoDB available. All of them providing unique features not found in the others. This can be exploited to provide special performance or scaling characteristics to Tygrstore. Changing the backend Storage is a matter of reloading the database.

Mixing different types of backends have not been tested and is currently not supported by the configuration subsystem. As the API provides unambiguous methods it should certainly be possible, since there is absolutely no (self)coupling between different index instances.

2.3.1 Storage Backends

All storage backends need to subclass the KVIndex class. This superclass provides the general housekeeping functionalities common to all types of backend storage. The order of the index should not be important in the actual implementation. It is determined or defined at creation time, where a decorator adjusts the parameters of the API calls to match the current index. This is the almost 'magic' input ordering decorator and its setup function:

```
def setup_reordering_decorators(self):
    self.reordering = []
    for i in self.internal_ordering:
        self.reordering.append(self.input_ordering.find(i))
    self.logger.info("reordering: %s" % str(self.reordering))
    self.ids_for_triple = self.input_reorder_wrapper(self.ids_for_triple)
    self.add_triple = self.input_reorder_wrapper(self.add_triple)
    self.selectivity_for_triple = self.input_reorder_wrapper(self.selectivity_for_triple
        )
```

```
def input_reorder_wrapper(self, original_func):
    def reorder(the_tuple, **kwargs):
        try:
            reordered_tuple = []
            for x in self.reordering:
                reordered_tuple.append(the_tuple[x])
            reordered_tuple = tuple(reordered_tuple)
            return original_func(reordered_tuple, **kwargs)
        except IndexError:
            self.logger.error("defect tuple!")
            self.logger.error( the_tuple  )
    return reorder
```

`setup_reordering_decorators` wraps the methods which receive triple-patterns with the `input_reorder_wrapper`. For each call made to these methods the incoming tuple is rearanged to match the order of the receiving index. This is a trick to make the code of the *KVIndex* implementations much more concise.

KVIndex API Methods

A KVIndex implementation has to provide the following methods to be able to work:

- `__init__`
- `open`
- `close`
- `add_triple`
- `selectivty_for_triple`
- `ids_for_triple`

  `__init__(config_file, name="spo")`

  `config_file` is an instance of ConfigParser.RawConfigParser and enables direct access to the tygrstore.cfg configuration file. Each backend type should use a separate section to read configurations which can not be shared. All other configuration shall be read from the [index] section. The name parameter is a string that denotes the ordering of the index and can be read from the property `self.internal_ordering`. This association is done in the base class KVIndex. Therefor it is necessary to initialize the Base class by calling `super(KlassName, self).__init__(self.config_file,name)` where KlassName is the name of the subclass in which the call is made. Also a call to the open method should be made to assure that the connection to the actual back end works.

`open()` returns: `INDEX_OPEN` all methods needed to initialize the backend(s) should be made here instead of the init function so backends can be closed and reopened at runtime. If the backends fails to open an Exception should be thrown. In case the Index uses different levels they should be stored in the list self.levels

`close()` returns `INDEX_CLOSED` this function should close the backend in a way that it can be safely opened by another instance and all network traffic to the backend is stopped.

`add_triple(tuple_of_ids)` this method should add a single triple to the index. The parameter is a tuple which contains Ids in the order described by `self.input_ordering` e.g. 'spo'

`__len__()` returns: an integer returns the total number of entries in the index

`selectivity_for_triple(tuple_of_ids)` returns: an integer returns the selectivity estimation of the `tuple_of_ids` triple pattern. In the best case it returns the effective number of triples matching the input pattern. If this is not possible another algorithm which mimics these numbers should be implemented or `self.selectivity_estimation` should be set to false.

`ids_for_triple(tuple_of_ids, num_records=-1, searched_key=-1)` returns: a generator which yields ids for the first key which is none. E.g. if `tuple_of_ids` is ('id1' , 'id2' , None, None) and we are in a spoc index then the generator should yield keys of type 'o' . If num_records is positive it should yield Sets of ids, where each set contains the amount of record this variable has. If `searched_key` is positive it should contain the index of the key which should be resolved, thus 2 for the above example or if keys of type 'c' are looked for it should be 3.

Following is a list of properties maintained by the base class:

- `self.updateable`: if this is set to False the store does not need to accept calls to the add_triple() function.
- `self.keylength`: the length of the database id in bytes
- `self.input_ordering`: the ordering in which requests come in. Usually 'spo' or 'spoc'
- `self.internal_ordering`: the ordering of the index. E.g. 'ops'

2.4 Sparql Parser

Sparql is parsed by Rasqal[3] via its cython bindings cysparql. Since these bindings are new and not finished I have created wrapper classes for easier prototyping and debug-

[3] `http://librdf.org`

ging. The helpers module provides three of these wrapper classes around cysparql: *Triple*,*BGP* and *ResultSet*. These exist to maintain and aggregate properties needed to resolve the query. While they could (and eventually should) be moved into the cysparql module, there are reasons to keep them. First of all they can be seen as a prototype about which functionality Tygrstore expects. Further they help to give a more detailed view when profiling since they can be run in pure python mode.

The most important class ist *ResultSet*, it provides the methods on which the *QueryEngine* algorithm operates. It has the following properties and methods:

- `triples` a list of all triples in the Basic Graph Pattern
- `variables` a list of all variables in those triples
- `unsolved_variables` a list of all variables that are unsolved in the current recursion step
- `solutions` a Dictionary with the variables as Key and id's as Values
- `triples_with_var(var)` a list of all triples which contain a certain variable
- `unsolved_triples()` all triples with unsolved variables
- `get_most_selective_var()` the next variable to solve
- `triples_with_var(var)` all triples containing the variable supplied
- `resolve(var, solution)` resolve a variable with an id
- `unresolve(var)` unresolve a variable

It is important to note, that this API will most certainly change, as the interplay between cysparql, *ResultSet* and *QueryEngine* is not yet optimal. Chapter 6 explains possible solutions for that.

2.5 String Store

The *Stringstore* Instance maintains the mappings from database IDs to all RDF literals (URIs and string literals) in the store. The base class Stringstore provides the function `get_new_id(any_string)` which returns a new ID. If a Hash-function is used then this function is just an alias to the hash-function. Else it uses the API methods of Stringstore which must be implemented.

Stringstore API Methods:

`id2s(an_id)`: returns: a string in n3 notation. This is used when the Database ID gets converted back to a result for a variable.

`s2id(a_string)`: returns: a database id. Used when the triple patterns are encoded to database ids. If `self.numeric_ids`: is false this function can be either an alias or a wrapper to `get_new_id(any_string)`. In case the String is not in the database the function should raise a `LookupError`. This can be exploited to abort the query if necessary triple patterns will not match at all.

`contains_string(a_string)`: returns: Boolean. Should return true if the RDF Term is already in the database

`add_string(a_string)`: returns: The new ID. This will add a String to the Database. It should use `get_new_id(any_string)` to get a new string.

`next_id(self)`: this is us used only if numeric_ids in [general] is set to true/yes in tygrstore.cfg. It should return the next free Database id which can be used for a new entry

`get_or_add_string(a_string)`: returns: an ID. This will add a String to the Database or return the ID if the String is already in the Store. This is provided by the base class but can be overridden for performance reasons.

2.6 *QueryEngine*

The *QueryEngine* orchestrates the process of resolving the provided SPARQL query string. It does so in three stages:

- Parsing the SPARQL Query and preparing its Data-structure
- Conversion between RDF Terms and Database keys and vice versa
- Recursive resolving of the Variables

QueryEngine API methods:

- `execute(query)` Query is a SPARQL query as string. The method returns a generator which yields hashes with the variables as keys and their solutions as values.

2.6.1 Query Engine Architecture

Currently the Engine only supports select Queries with one BGP (Basic Graph Pattern). Let us look at the steps from Section 2.6 in more detail:

The Query is parsed by instantiating a `cysparql.Query(query_as_string)` with the query in SPARQL notation. In the current Tygrstore version, only the top most Basic Graph Pattern is selected. Each contained RDF Statement in the WHERE clause is wrapped

within a `Triple` object. These are kept in a separate list. Each of these statements is also enriched with the corresponding database keys and a selectivity estimate. It does so by first using the *Stringstore* instances `get_ids_from_tuple` method. After that a valid tuple can be sent to `IndexManager`'s `selectivity_for_tuple` method.

Then an (empty) *ResultSet* is instantiated with the list of all statements and a list of all variables within these. Now the first recursive step of variable resolving is entered by a call to `evaluate(empty_result_set, firstvar)`. The first variable given by *ResultSet*s `get_most_selective_var()` method. The evaluate method then merge-joins all Statements in which the Variable being resolved is contained. For each of the Keys yielded by the join operation, the current variable is set as resolved. There after, a new recursion step is taken, to solve the next most selective variable. If the recursion returns, the variable is reset as unresolved again. The depth of the recursion is limited by the number of variables present in the BGP. If all variables are solved the property `solutions` of the *ResultSet* is yielded and the recursion depth decreases.

Using python generators, the merge join can actually merge multiple generators in a chain. `multi_merge_join` takes a list of generators (such as those provided by *IndexManager*'s `ids_for_ttriple` method) and merges them with the `merge_join_with_jump (left_generator, right_generator)` method. As the name suggests, this merge-join implementation provides the possibility to jump within the index by using the `send()` method of generators. The bigger the difference of the merging sets' size is, the higher the chance that this jump operation has a huge impact on the performance. Of course the jumping needs to be possible in the backend, if not, it can be emulated as shown in section 3.2.2.

The actual merge join operation uses three methods to build up the generator chain:

The first step is the `multi_merge_join` method. It just collects the different generators and passes them on.

```
def mergejoin_ids(self,triples_with_var, var):
    id_generators = []
    if len(triples_with_var) == 1:
        return self.index_manager.ids_for_ttriple(triples_with_var[0], var)
    for triple in triples_with_var:
        id_generators.insert(0,self.index_manager.ids_for_ttriple(triple, var)
            )
    #self.logger.debug( "joining %s generators" % len(id_generators))
    return self.multi_merge_join(id_generators)
```

For each triple pattern we add a generator (that generates matching results) a list. This list is sent to `multi_merge_join` which returns the generator for that keyspace:

```
def multi_merge_join(self, generators):
    #generators = list(generators)
    result = generators.pop()
```

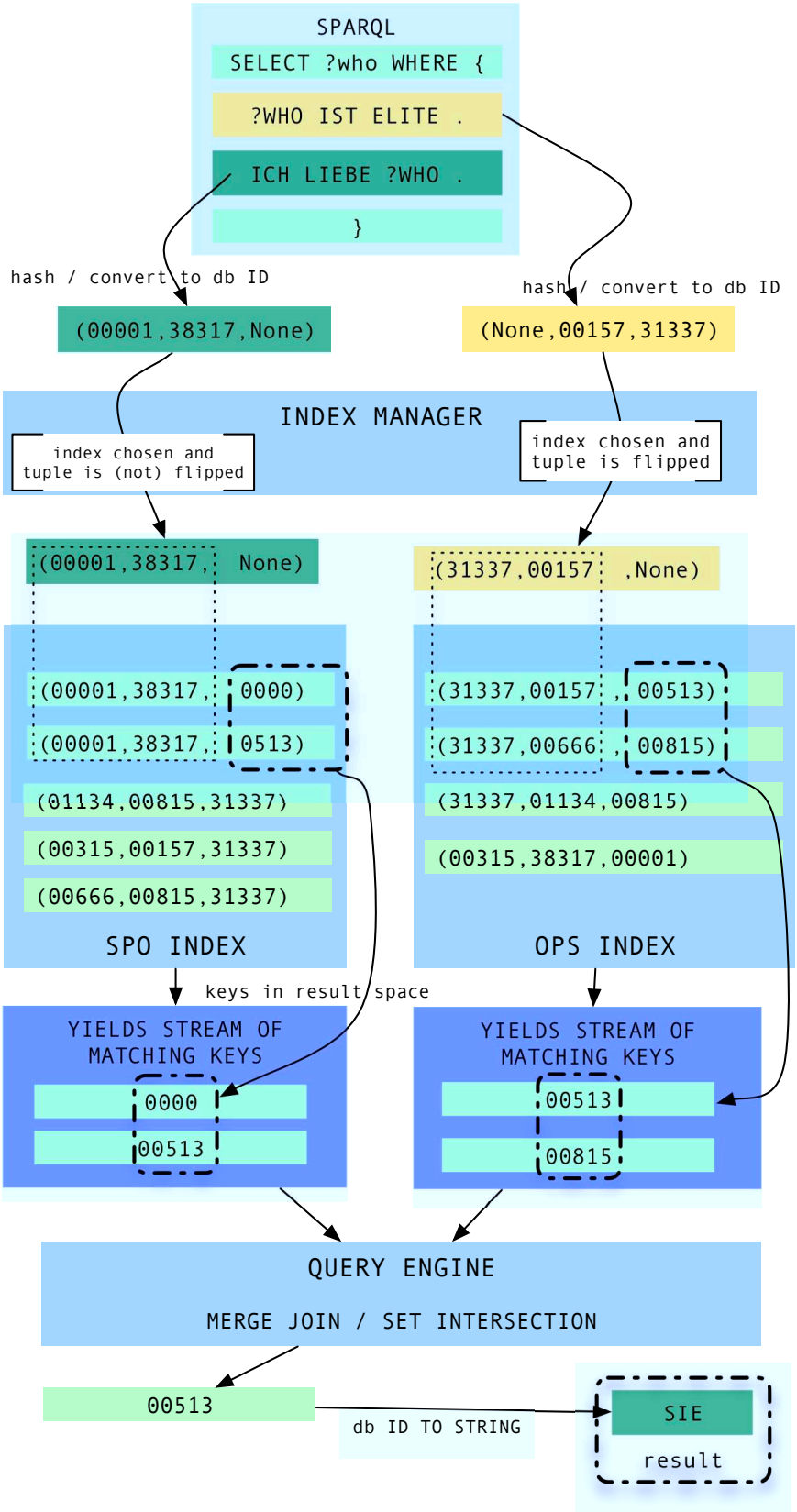

Figure 2.2: Tygrstore Query Processing.

```
    if self.jump_btree:
        while len(generators) > 0:
            result = self.merge_join_with_jump(result, generators.pop())
    else:
        while len(generators) > 0:
            result = self.merge_join(result, generators.pop())
    return result
```

`multi_merge_join` chains the different generators in that it always calls `merge_join_with_jump` for two generators. One of them being the generator returned from `merge_join_with_jump` . This is where the actual merging happens. It uses the standard *identity* comparator. The only special thing here is, that it sends the next possible lowest database id to the generator. Of course this approach needs the input to be sorted. The whole process is drawn by example in Figure 2.2

```
def merge_join_with_jump(self, left_generator, right_generator):
    left = left_generator.next()
    right = right_generator.next()
    while left_generator and right_generator:
        comparison = cmp(right, left)
        if comparison == 0:
            yield left
            left = left_generator.next()
            right = right_generator.next()
        elif comparison > 0:
            left = left_generator.send(right)
        else:
            right = right_generator.send(left)
```

2.7 Installation and Configuration

2.7.1 Dependencies

The following Libraries and Applications should be present

- Raptor $\geq$ 2.0
- Rasqal $\geq$ 0.9.24
- git (for cython, not needed for pure python mode)
- python = 2.6
- kyoto cabinet, tokyo cabinet or mongodb

The following Python packages should be present

- cysparql (provided)

- cython (a version which supports generators, see below)
- kyotocabinet[4]
- tc[5]
- pymongo[6]

Currently the released cython version 0.14 does not support generators. But a branch exists on github[7]. It can be installed by the following commands:

```
git clone https://github.com/vitek/cython.git
cd cython
sudo python setup.py install
```

after that, the libraries from the lib dir should be installed:

```
cd Tygrstore/libs/cysparql
sudo python setup.py install
```

If all prerequisites are met, Tygrstore can be used either in pure python mode or as compiled cython module. To switch between the modes the bash scripts tocython.sh and topython.sh can be used. It just renames the Tygrstore files from .pyx to .py and vice versa. To compile the pyx files the standard script setup.py can be used or the build.sh file. Assuming a database and the configuration files are present, Tygrstore can be used within python in the following way:

```
import ConfigParser
from kyoto_cabinet_stringstore import *
import index_manager as im
import query_engine
LOG_FILENAME = 'logs/test.log'
logging.basicConfig(filename=LOG_FILENAME,level=logging.DEBUG)
config = ConfigParser.RawConfigParser()
config.read("cfgs/benchmark1-kc-kc.cfg")
logging.debug("starting stringstore")
stringstore = KyotoCabinetStringstore(config)
logging.debug("starting index_manager")
iman = im.IndexManager(config)
logging.debug("starting QueryEngine")
qe = query_engine.QueryEngine(stringstore, iman, config)
for result in qe.execute(query):
    print str(result)
```

[4] http://fallabs.com/kyotocabinet/pythonlegacypkg/
[5] http://pypi.python.org/pypi/tc
[6] http://api.mongodb.org/python/1.9/index.html
[7] https://github.com/vitek/cython

2.8 Importer

To import RDF Statements into Tygrstore in a efficient way, it is best to save them into the used backend store directly and without any overhead. A set of scripts reside in the importer directory of the Project to accomplish this. They follow the Unix philosophy: each script does only one specific task, but they can be connected with pipes. To parse ntriples files a regular expression that just separates the properties of each Statement is feasible. To parse RdfXML and other formats, the Python module RDF is used. This module includes bindings to raptor. Since importing into a B+Tree can be achieved in constant time if the input is sorted, this should be preferred. The general approach is a three step operation:

1. parse the ntriples file with nt2hxid.py. This generates the output for one hxid index file.
2. sort the hxid files with gnusort
3. import the hxid file to the database with hxid2kc or hxid2mongo.

nt2hxid.py reads a ntriples file from stdin and outputs *one* hxid file. The hxid file contains the concatenated hexvalues of the hashed RDF Terms in a Statement. The script needs to know which hashing function should be used (md5/sha1), which order the output should have (spo,sop,osp...) and the name of the output file. You can also supply the number of total triples you want to insert,this will yield in status updates while the process goes on. While this data-format uses double the space of its binary representation, it can leverage the sorting speed of gnusort. Note: the regexp in the ntriples parser is incomplete, it does not recognize blank nodes correctly.

hxid2kc.py reads a sorted hxid file from stdin and writes into Kyoto Cabinet B+Trees. It generates all levels of the index and updates the selectivity estimates as it goes along. The needed parameters are: the bnum for the btree, the name of the index (e.g SPO will result in the files SPO0.kct, SPO1.kct, SPO2.kct) and the total number of keys (line-count of the input file) for status updates.

hxid2mongo.py does the same as *hxid2kc* but for mongodb. It needs the hostname, database and collection as parameters instead of the bnum and name.

nt2sskc.py imports ntriples file into the stringstore hash-table. Parameters are: the bnum, the name of the output file and the total keys to expect. This script implements a caching strategy. The maximal cache-size parameter denotes how many keys should be read until the keys are written to the btree and the cache is cleared.

There is another prototype of an importer script for Kyoto Cabinet. The script nt2kc.py writes the input stream to a number of small trees. These can then be joined into a final big file by the merge operation of the Kyoto Cabinet API. Overall this is slightly faster than the above approach but it does not produce reusable hxid files. I also tried to implement a fan-out approach with the python multiprocessing tools. The use of queues (i.e. the serialization) slowed down the process more than what was gained from multi-core computing power. Instead the above scripts could be slightly adapted to read/write to stdout and the processes can work via standard unix pipes and tee to named pipes. This will also work to read RDFXml with librdf's rapper.

3

Example Implementations

3.1 Kyoto Cabinet Backend

Kyoto Cabinet is the successor to Tokyo Cabinet. Both of these DBM Libraries offer different Types of data-structures. Most notably both support Hash-tables and B+Trees. For the main Tygrstore implementation Kyoto Cabinet was chosen, since it offers a cleaner API and is actively developed. It also features more data-structures such as in-memory versions of the Disk based B+Tree/Hash-table. The configuration and the library allows easy switching between the different data-structures, mainly by supplying an alternative filename extension.

3.1.1 Data-model

To save the RDF Statements on disk we use the File Tree Database, Kyoto Cabinet's on-disk B+Tree implementation. Each index gets as many trees as there are components in the Statement. Thus normal triples are stored in 3 trees, to which we will refer as levels. Figure 3.1 shows which information the different levels maintain in a 'spo' type index. The database Ids are saved as fixed length binary strings within the b+tree. Each level consists of one more concatenated key as its parent level. This leverages the possibility that we can jump to prefixes of keys. The lower levels also act as storage for the selectivity estimation. If we increment the occurrence of each key combination while filling the tree, we have the exact numbers for the query optimizer. The deepest level does not save any information in its *value* fields. That *value* can be used to store information associated with a triple - to preserve disk space, only one index can be chosen i.e.: 'spoc'."

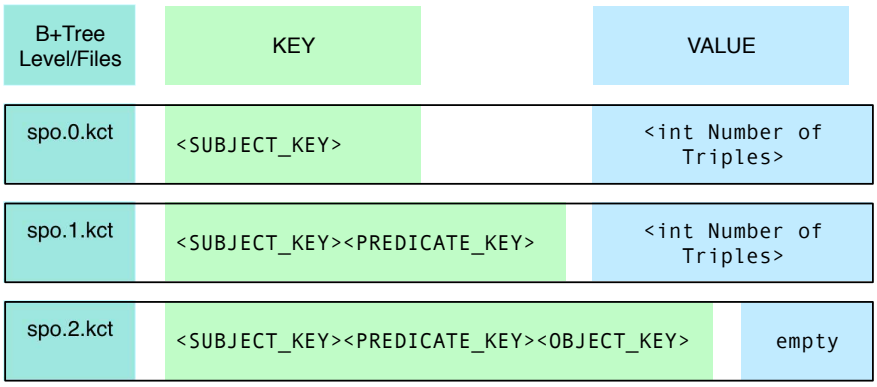

Figure 3.1: Index structure on different B+Trees..

3.1.2 KVIndexKC Implementation

The Kyoto Cabinet Index Backend resides in the *KVIndexKC* class. There is also a similar implementation for Tokyo Cabinet (*KVIndexTC*) which mainly differs in the names of the library calls. All parameters can be set within the [KC] (respectively [TC] for Tokyo Cabinet) section of tygrstore.cfg. The option `indexconfig` is the Kyoto Cabinet config string which is appended after the filename[1]. The bnum refers to the number of buckets and should roughly be 2-4x the actual number of statements who will be inserted into the tree.

[1] `http://fallabs.com/kyotocabinet/pythonlegacydoc/kyotocabinet.DB-class.html#open`

The actual generator for the keys requested by a triple pattern via `ids_for_triple` is the method `generator_for_searchstring_with_jump`. The parameter `search-string` is the actual prefix in the B+Tree and `loffset` and `roffset` define where the searched database ID lies within the tree (in bytes). It is actually just from the end of the search-string up to one key-length (length of a hash) to the right.

```
def generator_for_searchstring_with_jump(self,searchstring,loffset=0,roffset
    =16):
    #select the deepest level
    cur = self.levels[-1].cursor()
    #jump to the lowest possible key
    cur.jump(searchstring)
    while 1:
        try:
            next = cur.next()
            #check if we are still within the correct keyspace
            if next[0:loffset] == (searchstring):
                #yield the key and receive the next possible lowest key (
                    jumpto)
                jumpto = yield(next[loffset:roffset])
                if jumpto:
                    #advance the cursor
                    cur.jump("".join((searchstring, jumpto)))
            else:
                #keyspace is exhausted
                raise StopIteration
        except KeyError:
            self.logger.error("key error for: %s" % str(searchstring))
            cur = self.levels[-1].cursor()
            cur.jump(next)
```

The selectivity estimation is just a lookup in one of the levels (besides the topmost):

```
'''get the selectivity count'''
def selectivity_for_triple(self, triple):
    if triple == (None,) * len(triple):
        return len(self)
    else:
        key_prefix = "".join( filter(lambda x: x != None, triple) )

        long_big_endian = self.levels[len(triple_without_none)-1][key_prefix]
        return struct.unpack_from(">q", long_big_endian)[0]
```

Note the clumsy retrieval of the integer. This is actually a limitation of the Kyoto Cabinet library (Same goes for Tokyo Cabinet, there the function is called addint). The api foresees the `increment` method to retrieve ints. This has the severe sideffect that it needs the database to be opened with the DBOWRITER flag. Whenever the application crashes and the file is opened in DBOWRITER mode the b+tree is being repaired automatically by the Kyotocabinet Library. Unfortunately this copies the whole file,

while blocking all access to the b+tree. The above procedure converts the binary string (which contains the 8byte signed integer in big endian) to a python long.

3.2 MongoDB Backend

3.2.1 About MongoDB

MongoDB [2] is a relatively new, schema-free document oriented database which gained a strong community in the recent months. It is Open Source[3] but backed up by a VC financed Startup named 10gen. It is designed as a scalable distributed database with extensive query capabilities. By reducing transactional semantics it gains performance and horizontal scalability in comparison to tradition RDBMS. Queries are being written in Javascript and evaluated in the database with the built in Spidermonkey Javascript engine. Besides document matching and altering, MongoDB also supports aggregation functions such as MapReduce or grouping.

MongoDB works over the network and provides libraries for the most common programming languages.

MongoDB is already widely used in large scale production systems[4]. Among many others the New York Times [5], CERN [Kuznetsov et al., 2010], and Springer[6] use it. The most impressive documented setup is currently at wordnik. They host over 12 Billion records (3TB data) within their mongodb cluster [7] [8].

Two important features make it specially suitable as a Tygrstore backend: Cursors for query results and compound secondary indexes. The Cursor allows us to seamlessly stream the result set and also propagates operations such as distinctness and sorting directly back to the store. The indices internally are implemented as B-Trees [9]. Thus MongoDB gives us a scalable btree and even does the maintenance of the indices for us.

A mongod server can host multiple databases. Each databases can contain any number of so called Collections (Think tables in RDBMS world), within these you can store Documents. These Documents are stored in BSON[10] format and usually processed as JSON within the client library. It supports the most common datatypes but allows also custom datatypes, which are stored as binary strings. For our purpose the types ObjectID and Binary are relevant. The former is a 12 byte GUID (4 byte timestamp, 3 byte machine id, 2byte process id and 3 byte counter) and the later can save binary

[2] https://mongodb.org
[3] GNU AFFERO GENERAL PUBLIC LICENSE Version 3.0
[4] http://www.mongodb.org/display/DOCS/Production+Deployments
[5] http://hackshackers.com/2010/07/28/a-behind-the-scenes-look-at-the-new-york-times-moment-in-time-pr
[6] http://realtime.springer.com/
[7] http://blog.wordnik.com/b-is-for-billion
[8] http://blog.wordnik.com/12-months-with-mongodb
[9] http://www.mongodb.org/display/DOCS/Indexes
[10] http://www.bsonspec.org/

data of arbitrary length. For Binary there are also subtypes, we use the `MD5_SUBTYPE` for md5 keys, this uses exactely 16 bytes within the db.

The Mongo wire protocol[11] directly contains the data in the same format as it is written on disk. It is especially designed to minimze overhead. The on-disk Fileformat consists of a .ns namespace file and a sequence of files prefixed with the database name. Their size is doubled on each new file created (starting with 256MiB). Each new document is stored after the next, but a padding factor leaves empty space, so that documents can have growth without the need to move its physical location.

Limitations

Currently it is not possible to jump with a cursor. This means, that we either have to throw away intermediate results or create a new cursor to model jumping. The later can be done by adding a conditional operator for the space we are searching. Effectively setting a minimal value for the searched keyspace. While this works in practice, it brings a certain amount of overhead, which slows down the merge join operation considerably.

3.2.2 *KVIndexMongo* Backend

The MongoDB Index Backend is implemented in the Class *KVIndexMongo*. The main difference to the Tokyo/Kyoto store is, that indices are handled by the database and the selectivity estimation is also retrieved from there. While this simplifies maintenance, it needed some tweaking of the initial architecture. But it also proved the architecture decisions to be right, as the api changes where minimal.They would have been even fully avoidable, although with some performance hits. The details are discussed within the following description of the triple store datamodel used by *KVIndexMongo* implementation

Datamodel

Each triple is stored as one Document, with a node id (`_id`) and one key for all naturals (e.g. s,p,o or spoc). As values for the Keys we define a class name in tygrstore.cfg. Currently supported classes are bson.ObjectID, bson.binary.Binary and long. Note however that the ObjectID datatype only makes sense if you use the provided MongoDB Stringstore [see 3.2.3]. You can also define custom classes which serialize to binary. To plug into our MongoDB adapter, a custom Type would need an instance

[11] `http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol`

property called `binary` which provides the key as string. Also the standard comparison method `__cmp__(self, other)`. Using *Binary* as key type, an example document would look like the following in JSON notation:

```
{
 '_id': ObjectId('13370000000000000038317'),
 's': Binary('13371b5a421405f522000815',2),
 'p': Binary('13371b5a421405f522000815',2),
 'o': Binary('13371b5a421405f522000815',2)
}
```

Each mongodb document needs an unique _id key of any datatype. It needs to be unique within the collection and it is also used for sharding by default. Another aproach would have been to store the whole triple as a custom datatype and use mongodb as sharded btree only. This would have resulted in the same aproach used by the Kyoto/Tokyo implementation. This has not been followed because i wanted to push the maintenance of the compound indexes and the selectivity estimation down to the database. As we now have no need to define the indices ourselves, there *would* be no need that the *IndexManager* holds all 6 (for a fully indexed triple store) *KVIndex* instances. Reordering the triples before entering an *KVIndex* instance is made unnecessary by the `searched_natural` parameter.

Indexing

To archieve fast lookups we want 6 way indexing as used with the Kyoto Cabinet backend. This needs to be done manually by indexing the collection by compound keys. If our triples are stored in a collection named 'spo' in the 'tygrstore' database the following commands start the creation of the indexes in a mongo shell:

```
use tygrstore
db.spo.ensureIndex({s:1, p:1, o:1});
db.spo.ensureIndex({s:1, o:1, p:1});
db.spo.ensureIndex({o:1, p:1, s:1});
db.spo.ensureIndex({o:1, s:1, p:1});
db.spo.ensureIndex({p:1, o:1, o:1});
db.spo.ensureIndex({p:1, s:1, s:1});
```

Note that these operations are blocking, unless the optional parameter `{background: true}` is suplied. The creation of an index for 130mio triples takes about 20 minutes.

API Changes

Maintaining the Indexes directly within the database required some adaption of the way the Kyoto/Toyko adapters worked. Thus the following *KVIndex* api changes where introduced:

- The property `update_only_one` (configured via tygrstore.cfg) can be set to true. If this is the case, a call to `add_to_all_indexes(triple)` will insert the triple only into the first index. Note also, that the *IndexManager* still maintains 6 different `KVIndexMongoDB` instances within `unique_indexes`. This can be exploited to provide paralellism to different mongo routing servers. A configuration option to define hosts/ports per server would need to be introduced for that.
- The superclass `KVIndex` is initialized with a new optional parameter `reorder=False` which disables the reordering decorator.
- The function `ids_for_triple` got an optional parameter named `searched_natural`. This is a single character (e.g. 's' or 'p' or 'o') which describes the key being searching for. This was done because that information was easily accessible in the *IndexManager* class but would have been cumbersome to reconstruct on the *KVIndex* level. It would have been possible trough the use of the `internal_ordering` property and enabled reordering.

Implementation

The actual implementation of *KVIndexMongo* then was very straight forward. Still the internals shall briefly be discussed to hint to the (current) limitations:

To save or search a MongoDB document the find function needs to be provided with a json object including the relevant properties. The *IndexManager* (with disabled reordering) delivers us a tuple of the form `( 'keyS', 'keyP', 'keyO')` each of these keys can be None. To generate the needed JSON structure the function `to_mongo_hash` is provided:

```
def to_mongo_hash(self, triple):
    mongo_hash = {}
    for i in range(len(self.input_ordering)):
        if triple[i]:
            mongo_hash[self.input_ordering[i]] = self.id_class(triple[i])
    return mongo_hash
```

`self.id_class` here contains a reference to the *Class* of which type the keys are (e.g. *Binary, ObjectId*). Thus each key of the input tuple is wraped in an object needed for MongoDB.

The second noteworthy function is `generator_for_searchstring_with_jump`. It is the generator which is returned by `ids_for_triple`.

```
def generator_for_searchstring_with_jump(self, mongo_hash,searched_natural,
    num_records=-1):
    cursor = self.collection.find(mongo_hash, {searched_natural:1})
    #we can inform mongodb with a hint about the compund index it should
        choose
```

```
#cursor.hint( [ (self.internal_ordering[0],1), (self.internal_ordering
    [1],1), (self.internal_ordering[2],1)] )
cursor.sort(searched_natural)

nextid = None
result = None
while 1:
    mongo_doc = cursor.next()
    result = mongo_doc[searched_natural]
    #primitive jump emulation!
    if nextid and nextid > result.binary:
        mongo_hash.update({searched_natural:{"$gte":Binary(nextid, bson.
            binary.MD5_SUBTYPE)}})
        #jump emulation by creating a new cursor. slow!
        cursor = self.collection.find(mongo_hash, {searched_natural:1})
        #cursor.hint( [ (self.internal_ordering[0],1), (self.
            internal_ordering[1],1), (self.internal_ordering[2],1)] )
        cursor.sort(searched_natural)
    if type(result) == long: nextid = yield(result)
    nextid = yield( result.binary )
```

The sort operation on the cursor allows mongodb to use the correct index, it is only costly if such an index is not present. Since the cursor does not suport forward jumping this is emulated. This is a severe preformance limitation. The solution is to start a new query with the $gt (greater than) parameter set for the key[12].

3.2.3 MongoDB Stringstore

The class *MongoDBStringstore* implements a *Stringstore* model to save the id to string conversion and vice versa. Each RDF Term is saved as a Document like the following:

```
{
 '_id': ObjectId('13371b5a421405f522038317'),
 'n3': '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
 }
```

or with custom ids

```
{
 '_id': Binary('13371b5a421405f522000815',2),
 'n3': '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
}
```

[12] a ticket in the mongdb bugtracker has been opened

MongoDB Scaling

MongoDB can be set up as a cluster. Each such cluster consists of 3 types of nodes: shard servers (mongod) , config servers (mongod) and routing processes (mongos). Figure 3.2 visualizes such a setup. A shard can in fact consist of multiple replicated servers to quarantee availability. Each shard saves chunks of a sharded collection. A chunk is defined as all elements within a range of the sharding key. This information and other central metadata is kept in the config servers. Routing servers act as a gateway to split requests to the shards and merge their results. From the client there is no distinction wheter he is talking to a mongod server or to a routing server. All this is absolutely hidden from the client, he sees no difference what so ever between a sharded setup and just one single mongod instance.

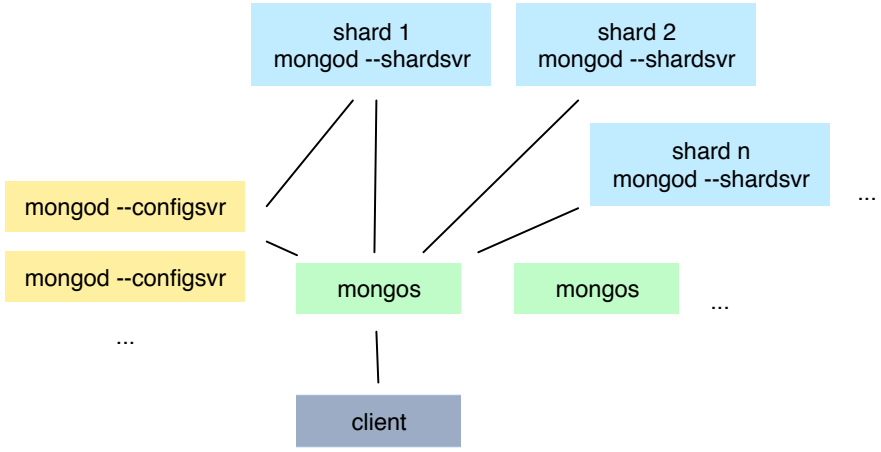

Figure 3.2: MongoDB sharding concept.

4

Evaluation

In this chapter we will explore performance characteristics of Tygrstore. First Tygrstore is benchmarked against ttlite (a TokyoTygr version with cython bindings) which is nearly entirely written in c. in section 4.1 Tygrstore with both backends is profiled to get more insight into where time is lost.

4.0.4 Dataset

As primary benchmark the Lehigh University Benchmark (LUBM) [Guo et al., 2005] has been chosen. Two Datasets have been prepared and converted into the ntriples[1] format. Generating data for 1000 University resulted in 138'318'414 Statements and 100 Universities generated 1'273'248 Statements. The current performance limitations of the importer scripts would have made larger datasets a time consuming task to import.

4.0.5 Hardware

The hardware platform was a standard PC with an Intel Core2Quad Q8400@2.66GHz cpu and 8GB of main memory. The operating System was Archlinux [2] running a standard 2.6.37-ARCH Kernel. If not noted otherwise the data was stored on a RAID 0 disk array of two Corsair CSSD-F120GB2 120GB SSD Drives.

4.0.6 Software

Kyoto Cabinet 1.2.34 Mongodb: v1.9.0-pre from git a6105a2855132288324 Python Modules: kyotocabinet-python-legacy-1.13, tc 0.7.2, pymongo 1.9+

[1] http://www.w3.org/2001/sw/RDFCore/ntriples/
[2] http://www.archlinux.org/

4.0.7 Benchmarking

The first Benchmark Setup does a list of LUBM Queries each of them six times, for a maximum of 1, 10, 100 and 1000, 10'000 and 50'000 results. The detailed Results can be found on page 48. Tygrstore is compared against ttlite (cython bindings to TokyoTygr). This peer was chosen because Tygrstores Query Engine was constructed in a similar manner to TokyoTygr. So we can obtain relation to what the added flexibility (more dynamic code) and modularization (pluggable backend) cost us.

Each query has been made twice and in advance to the first query the disk cache was cleared with `echo 3 > /proc/sys/vm/drop_caches`.

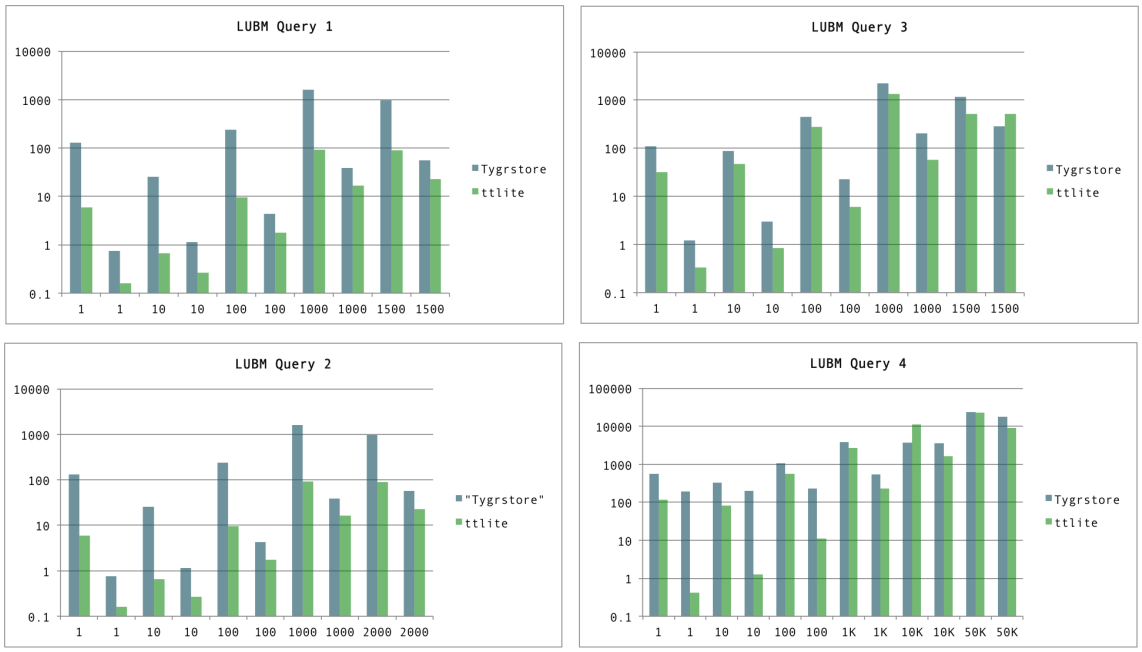

Figure 4.1: Warm and Cold runs in milliseconds

We see that ttlite is still much faster with small queries. As the result set gets bigger, Tygrstore runs up and is even faster sometimes in Query 3 and 4. This is probably due to the fact, that the current parsed query has not an optimal structure but the python generators and the comparison of ids is quite efficient. As that data-structure is only heavily used in the beginning it has less effect the longer the query takes.

Some optimizations have already been added by reformulating certain code but the biggest impact was caused by caching the `index_for_ttriple` method from the *Index-Manager* class. For that the `@memoized` decorator has been added to the method. This resulted in performance gains up to 75%. The minimum gain measured was 16%, the details can be read from Table 4.1. Still some optimizations are not done yet or should

only be done for specific reasons when prototyping. Chapter 6 explains some of them.

Tygrstore vs. ttlite

Query	Number of Results	Tygrstore	Tygrstore with @memoized	gain	ttlite
lq1	10	1.28	0.91	41%	0.40
lq1	100	6.47	3.9	66%	1.25
lq1	1000	59.11	33.78	75%	11.30
lq2	10	4.90	3.19	54%	1.77002
lq2	100	42.41	25.14	69%	14.89
lq2	1000	399.73	245.30	63%	158.55
lq3	10	3.76	2.72	38%	2.70
lq3	100	30.62	18.41	66%	13.32
lq3	1000	282.32	175.54	61%	88.12
lq4	10	199.79	172.41	16%	4.71
lq4	100	238.45	198.02	20%	25.75
lq4	1000	620.10	466.78	33%	146.68

Table 4.1: Queries LQ1 - LQ4 on the LUBM Dataset with 100 Universities. Query Time in milliseconds

Figure 4.2 shows the results from Query 4 on the 138 Million triple dataset. We see that the result rate per time is constantly around 2000 to 2500 results per second. Only for less than 1000 results this is not the case, probably due to the setup costs.

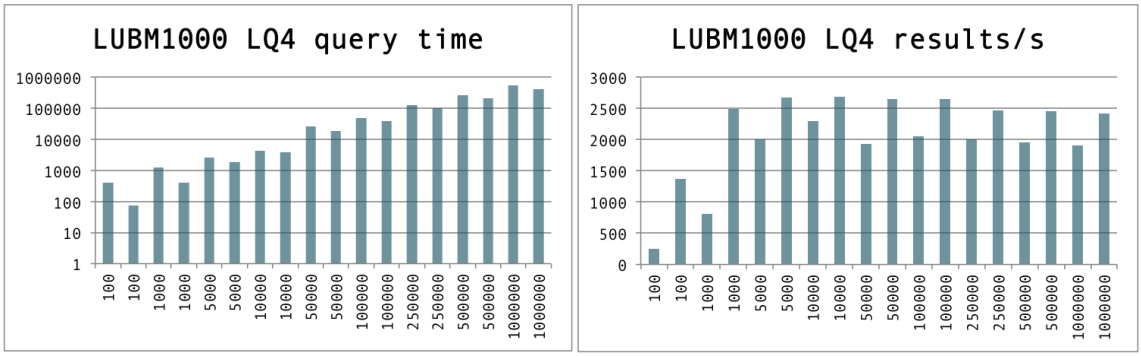

Figure 4.2: Query Time and requests/s on the 138 Mio. Dataset (Warm/Cold)

4.1 Profiling

The mixed numbers from the benchmarks suggest that there would be some work needed to get on par with ttlite. With the following profiling it should be explored which the driving factors behind the lack of performance are. Let us start the Story a bit earlier tough:

The first attempt to Profile Tygrstore brought really strange Results. Running Query LQ5 on the 100 Universities Benchmark trough cprofile resulted in the profile shown in Table 4.4.

12285534 function calls (9551702 primitive calls) in 17.104 CPU seconds

ncalls	tottime	%	cumtime	p/c	filename :lineno(function)
3248172	9.314	54.57%	11.188	0.000	indexkc.py:109(generator_for_searchstring_with_jump)
1528290/28122	3.383	19.77%	15.780	0.001	query_engine.py:171(merge_join_with_jump)
608494	1.651	9.65%	1.651	0.000	`method 'jump'of 'kyotocabinet.Cursor'objects`
3191901	0.755	4.41%	0.755	0.000	`cmp`
1677670/458118	0.474	2.77%	14.958	0.000	`method 'send'of 'generator'objects`
28137	0.184	1.07%	0.365	0.000	index_manager.py:80(index_for_ttriple)
608500	0.150	0.87%	0.150	0.000	`method 'join'of 'str'objects`
14120/8	0.136	0.79%	17.101	2.138	query_engine.py:74(evaluate)
28143	0.097	0.56%	0.243	0.000	index.py:38(reorder)

Table 4.2: Profile of Query LQ5

selectivity	subject	predicate	object
411501	?student	ub:advisor	?advisor
72302	?advisor	ub:worksFor	?department
32043	?department	ub:subOrganizationOf	?university
2148830	?student	ub:name	?name
1120834	?student	ub:telephone	?tel
7	?student	ub:takesCourse	GraduateCourse33

Table 4.3: Selectivities of Query LQ5

15 seconds and 608'494 jumps in the btree to generate just 7 results, that was bad. After some time in the debugger it was clear, that path of the most selective variable was not followed. Table 4.3 shows the selectivities and the triples. The *QueryEngine* chose to resolve university as the second variable, because it the selectivity estimation was not updated. After changing that, the order in which the variables where resolved was correct. But query time only went down to about 14 Seconds. The first result set was delivered after some milliseconds and then there were 10 seconds without results.

The reason was subtle: the order of the merge join generator chain does matter! After the first result set was generated a new generator was *appended* to the chain, but there where still unexhausted generators before that. So then the chaining was changed so that new generators where inserted into the front of the list.

Table 4.4 shows the same cprofile benchmark with the now corrected *QueryEngine*. This looks like a reasonable result.

6789 function calls (6703 primitive calls) in 76 CPU milliseconds

ncalls	tottime	% cumtime	filename :lineno(function)
95	036	036	`method 'increment'of 'kyotocabinet.DB'objects`
88	028	028	`method 'jump'of 'kyotocabinet.Cursor'objects`
8	001	076	`query_engine.py:26(execute)`
179	001	029	`indexkc.py:110(generator_for_searchstring_with_jump)`
78/8	001	072	`query_engine.py:75(evaluate)`
95	001	001	`index_manager.py:67(index_for_tuple)`
148	001	037	`index.py:38(reorder)`

Table 4.4: Profile of Query LQ5

So let us analyze a slower Query, to get a more detailed view on the scene. For that Query LQ4 was chosen, since it is the slowest. The detailed Profile is shown in table 4.5. It took 1044 milliseconds in pure python cprofile mode. Compiled with cython the same code took 540ms. This is nearly 50% slower than same code dynamically interpreted.

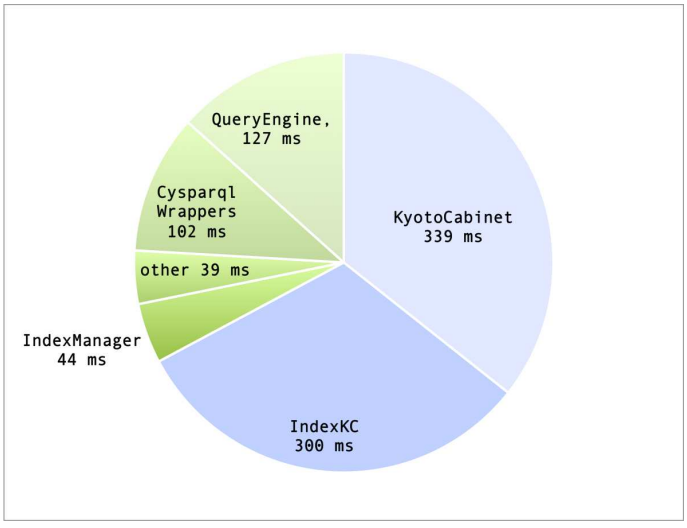

Figure 4.3: Distribution of calls milliseconds

About 1/3 (340 ms) of overall time is lost in the Kyoto Cabinet driver. The same percentage applied to the compiled version, is less than it took ttlite for the very same

564954 function calls (560292 primitive calls) in 1044 milliseconds

ncalls	tottime	cumtime	filename:lineno(function)
52843	245	499	indexkc.py:110(generator_for_searchstring_with_jump)
46521	206	206	method 'jump' of 'kyotocabinet.Cursor' objects
3576	103	103	method 'increment' of 'kyotocabinet.DB' objects
5663/1001	053	1.005	query_engine.py:79(evaluate)
3691	048	572	query_engine.py:187(merge_join_with_jump)
96402	036	036	cmp
6901	026	180	index.py:38(reorder)
9048	023	208	helpers.py:15(__call__)
43196	019	447	method 'send' of 'generator' objects
4000	018	018	method 'get' of 'kyotocabinet.DB' objects
3576	017	032	index_manager.py:67(index_for_tuple)
50097	016	016	method 'join' of 'str' objects
3576	014	129	indexkc.py:92(selectivity_for_triple)
10632	014	021	helpers.py:129(unresolve)
3325	013	056	index_manager.py:101(ids_for_ttriple)
3325	012	012	method 'cursor' of 'kyotocabinet.DB' objects
6901	012	018	filter
4691	010	016	helpers.py:123(resolve)

Table 4.5: Profile of Query LQ4, 1000 Results, Pure Python

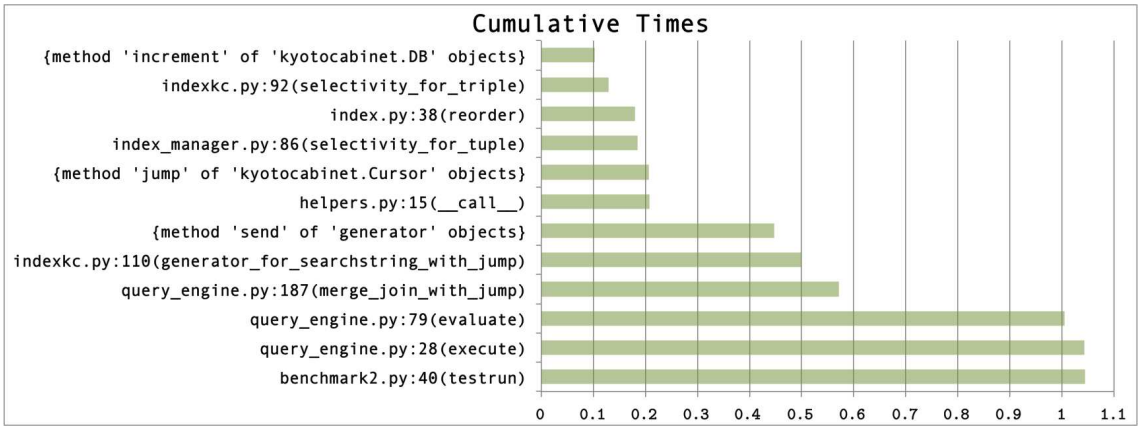

Figure 4.4: Time spent in and below a method in seconds

query. This suggests, that the current cpython kyoto cabinet driver is a limiting factor. Here we also see, that the cost of the reordering decorator is less than 3%. With optimized bindings also the `generator_for_searchstring_with_jump` function would be optimized. All non Kyoto Cabinet adapter method calls add up to 210 milliseconds. Subtracting about 50% performance hit by profiling and pure python mode we are around 15% below the time of ttlite. Albeit proof of this has to be made, it shows roughly the dimensions of the loss by the kyotocabinet python module.

MongoDB Backend Profile

The MongoDB backend adapter was even much slower than the Kyoto Cabinet one. The profile in table 4.6 shows a pretty clear picture. Note that a dashed lines represents omitted lines. Significant time is spent on Serialization of the Json hashes. This is absolutely unnecessary in theory as an optimal backend for Tygrstore could work with binary data. Figure 4.5 shows a rough overview over Table 4.6. To achieve decent query times a new mongodb adapter would be needed. I did not run much more Benchmarks with the MongoDB adapter since the results where too skewed by the serialization. Besides the aggregates are not being cached, thus looking up selectivities took also very long. A setup with 4 shards, one config server and one routing server has been set up and queries did run as expected.

28058548 function calls (28053889 primitive calls) in 46.382 seconds

ncalls	tottime	cumtime	per call	filename:lineno(function)
60068	13.282	13.282	0.000	method 'recv' of '_socket.socket' objects
30029	8.035	22.879	0.001	bson._cbson.decode_all
2541899	4.140	6.614	0.000	binary.py:83(__new__)
2503005	3.502	6.342	0.000	objectid.py:140(__validate)
10570444	3.345	3.345	0.000	isinstance
2503005	2.018	8.360	0.000	objectid.py:54(__init__)
2541899	1.656	1.656	0.000	built-in method __new__ of type object at 0x7fb55131d9a0
75755	1.382	41.850	0.001	indexmongo.py:118(generator_for_searchstring_with_jump)
30029	0.326	37.978	0.001	cursor.py:511(__send_message)
77667	0.318	41.008	0.001	cursor.py:597(next)
31653	0.198	40.397	0.001	cursor.py:549(_refresh)
30029	0.171	13.681	0.000	connection.py:662(__receive_message_on_socket)
5660/1001	0.170	44.108	0.044	query_engine.py:79(evaluate)
60058	0.158	13.470	0.000	connection.py:648(__receive_data_on_socket)
30029	0.117	14.168	0.000	connection.py:676(__send_and_receive)
3688	0.107	41.578	0.011	query_engine.py:187(merge_join_with_jump)
30029	0.084	14.424	0.000	connection.py:685(_send_message_with_response)
66120	0.042	37.840	0.001	method 'send' of 'generator' objects
1001	0.004	46.380	0.046	query_engine.py:28(execute)
1	0.002	46.382	46.382	benchmark2mongo.py:40(testrun)

Table 4.6: Profile of Query LQ4, 1000 Results, MongoDB Backend, Pure Python

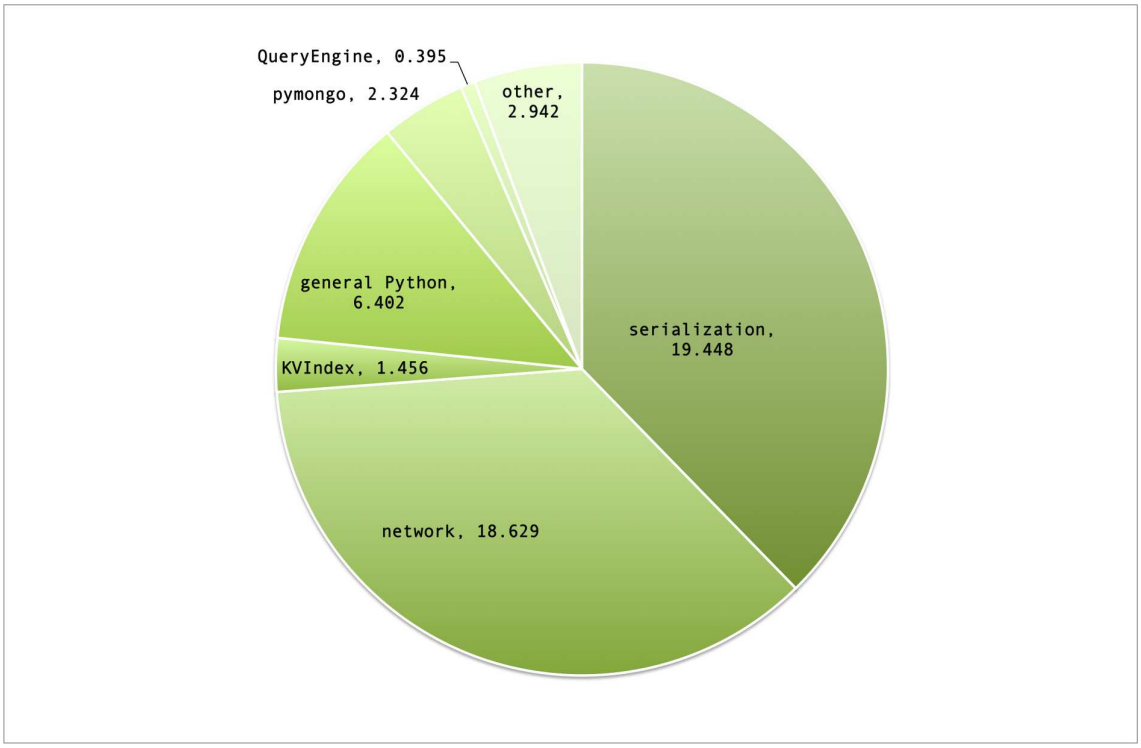

Figure 4.5: MongoDB Adapter Times spent in second

5

Limitations

5.1 Architectural Limitations

The Limits of the proposed architecture lays within the layering of the different modules. While this gives us the needed flexibility to have pluggable backend stores it abstracts the later. It further forces some decision to be made by the *IndexManager* which could be pushed down to the store. We saw that with the MongoDB implementation. There we actually have to fake the existence of multiple indexes. While in this case it has no side effect, it is strictly spoken not necessary. Choosing generators to stream the results limits the performance as it will not be possible to rewrite or optimize this streaming. But python/cython generators have very effective implementations so this should be the last concern, given the beauty and ease of use they give to the architecture.

5.2 Functional Limitations

Most of the missing features will be discussed in section 6.1 where also possible solutions will be presented. In general, the *QueryEngine* has the most severe limitations over the other components. Currently only one Basic Graph Patterns can be resolved. Albeit the Query is correctly parsed by cysparql, it is not yet transformed in a clear, concise data-structure. Since this data-structure (and a more universal algebra to solve it), will cost performance, it should be pluggable and the very basic approach in existence should still be possible. Any more advanced SPARQL operations such as filtering or aggregates will have some impact on the architecture and the internal API's.

Another function that is missing is the awareness of Datatypes/RDF Types. Tygrstore currently stores the Statements as strings in their n3 notation only.

Loading databases is currently slow and the regex based parser can not detect RDF Types.

5.3 Limits caused by Evolution

The biggest limit in the current implementation is probably the fact, that the author was new to Python. As Python coders have their very specific way of implementing patterns, some of them may have been violated. The domain of Triple stores was also new and the exact interaction of the different modules was unclear at the beginning. Despite several near complete rewrites some artifacts of previous iterations persist in the codebase

6

Future Work

As we have seen in the previous chapters there is still a lot room for improvements. The current Tygrstore implementation can be seen as a MVP (Minimal viable Product). Or in other words a proof that the architecture can work without unsolvable limitations.

In this Chapter some ideas of future enhancements within and around Tygrstore are shown. The next section shows where Tygrstore internally could be optimized. This can be regarded as an unordered Roadmap for further Development. Section 6.2 expands the view and shows some broader solutions and ideas in which Tygrstore could be a part of.

6.1 Internal Enhancements

There are mainly three areas where Tygrstore could be improved internally: performance optimizations, feature implementations and backend adapters.

Performance Optimizations

A lot of performance is lost in the current backend implementations. The cpython Kyoto Cabinet driver has some overhead. Thus rewriting it and using cython bindings would probably speed up the jump operations considerably. This would also lead the way to a custom key type with custom comparison function. Database IDs consisting of bigger hash functions could be factorized. If a hash function with uniform distribution is used, the key could be decomposed into integers which fit into registers. Then also a hash-table implementation with pluggable hashing function should be used for the Stringstore, so hashes will not get computed twice.

On the other side the SPARQL Processing and the Variable updating mechanism is not optimal. It could go further away from cysparql by enhancing the python classes for query processing. This is slower, but would allow faster prototyping of Query Optimization strategies. On the other hand the query processing could be pushed down to cysparql and a more defined interface will emerge, where other SPARQL parsing libraries could be plugged in.

Then there are different layers where concurrency could be added. Different graph patterns and especially Unions could be resolved in parallel. The Hash joins or variants there of could used when different Variable paths are explored in parallel.

Importing, conversion and exporting currently is too slow and involves too many tasks. This could be optimized by writing a custom cython adaptor to raptor (the provided python bindings are too slow) to reformat and sort the triples directly there and achieve maximum throughput.

Missing features

Feature-wise there is is also much work to be done to support more of SPARQL. E.g. an specialized Full-text engine such as Apache Lucene [1] could be used to support regex filtering. Maintaining knowledge about the RDF Type would be needed for that.

Regarding the MongoDB Backend, an extensive study on the true scalability of MongoDB for triple stores should be made. After that the jumping (and even the merge joining) could be implemented directly within a MongoDB node. Also selectivity estimation should get faster, either by maintaining a sharded Key-value store or by implementing a heuristic function for *QueryEngine*. After that a cython driver for the simple bson based MongoDB protocol should be written to minimize network overhead. This would also give the possibility to work directly on binary bson data and all serialization could be kept aside.

As for the Server, various caching techniques could be implemented. Especially keeping unexhausted generators or storing variable offsets for SPARQL queries with an OFFSET parameter would reduce overhead of such an operation. Adding a websocket based client with live streams should be fairly trivial, but would bring the idea of an 'infinite stream' out to the client side.

Last but not least there should be more backend adapters. Especially Riak [2] (it uses Erlangs GB Tree implementation) could be an interesting candidate.

[1] `http://lucene.apache.org/`
[2] `http://wiki.basho.com/An-Introduction-to-Riak.html`

Then the inter-mixing of different *KVIndex* implementations within one *IndexManager* could result in new performance characteristics. Alternatively a Hybrid *KVIndex* implementation could be done e.g. maintaing Level 0 and 1 within Kyoto Cabinet and Level 2 within MongoDB.

6.2 Ecosystem

As the real word use cases for RDF will often need to rely on data already stored in existing databases, there should be a way to minimize the overhead. The obvious way would be to just export the data into a triple store according to some Ontology. If the non RDF Storage system is a Document oriented store, the Ontology lies already partly within the document structure. Hence (generic) algorithms that generate *ad-hoc* RDF Statements on top of Documents are conceivable. Possibly with a map/reduce approach the RDF data would adapt to the flexible structure of Documents.

Next, the map/reduce functionality of MongoDB could be exploited on the data model described in Section 3.2. Work in this area has been done in [Myung et al., 2010].

As more and more computing power is on the client side it would make sense to distribute the most costly operations to where data is actually consumed. With todays fast Ecma Script interpreters and WebCL[3] (OpenCL for the Browser) being drafted, it could be possible to move the query engine to the client side. Tygrstore would then only serve to maintain and deliver the generators and abstract the backend storage . Together with a security model that narrows or widens the 'open world', this would give true, standardized data accessibility. Even better that it cuts costs for operators while acting as a driver for client-side hardware sales. It also would simplify the creation of mashups enormously. To accomplish this, a SPARQL parser in Javascript, a port of the *QueryEngine* and a WebSocket server for the *IndexManager* would be needed.

[3] `http://www.khronos.org/news/press/releases/khronos-releases-final-webgl-1.0-specification`

7

Conclusions

In this Thesis i have shown that a flexible architecture for RDF Storage is actually possible and imposes only modest concessions in regards to performance. Further it was sketched a way on which it is possible to horizontally scale a triplestore given that architecture. To become a truly usable solution some work is needed. That be refinements of the API and especially faster backend adapters. There is no clean sparql parser interface and no universal algebra / plan to solve a query.

I was very astonished about how simple the solution was in the end. Being able to resolve Basic Graph Patterns in less than a millisecond with my own code was inconceivable for me until i did it. Python with cython was a very good fit for this project since it is one of the few dynamic languages able to seamlessly go from highly expressive, duck typed code to routines written in c. This opens up the possibility to improve code in regards to performance when necessary. But before that happens, prototyping can happen very comfortably. On top of that it is fast and painless to interfere with already written c code.

I Truly hope that the Tygrstore framework can serve as a base for some further research. Personally i am wondering, how an elegant but still performant algebra for the *QueryEngine* looks like.

A

Appendix

A.1 Glossary

- Statement: A RDF Statement as defined by [Brickley, 2004] but including the Context for Named Graph relations.
- Triple: Same as Statement if not highlighted otherwise.
- Tuple: A python tuple
- Natural: A component of a Statement, thus a Subject, Predicate, Object or Context. Or one character out of the natural ordering.

A.2 Detailed Benchmark Results

Tygrstore: LUBM Benchmark Details

q	ms	no. results	time/result	result/second	QueryEngine setup time	warm/cold
lq1	1.794100	1	1.794100	557.382591	0.512123	c
lq1	0.988007	1	0.988007	1012.138996	0.324965	w
lq1	1.790047	10	0.179005	5586.446457	0.331879	c
lq1	1.276970	10	0.127697	7831.038088	0.332832	w
lq1	11.554956	100	0.115550	8654.294852	0.303030	c
lq1	6.438017	100	0.064380	15532.733400	0.352859	w
lq1	100.844145	1000	0.100844	9916.292134	0.336170	c
lq1	58.902979	1000	0.058903	16977.070069	0.355005	w
lq1	108.805895	1514	0.071867	13914.687270	0.342846	c
lq1	86.724997	1514	0.057282	17457.481233	0.398874	w
lq2	11.681080	1	11.681080	85.608523	1.173019	c
lq2	1.261950	1	1.261950	792.424712	0.489950	w
lq2	31.748056	10	3.174806	314.979912	0.488043	c
lq2	4.919052	10	0.491905	2032.911981	0.515223	w
lq2	313.961983	100	3.139620	318.509901	0.496149	c
lq2	42.371035	100	0.423710	2360.102860	0.550032	w
lq2	2923.185825	1000	2.923186	342.092518	0.507116	c
lq2	402.168989	1000	0.402169	2486.516929	0.521183	w
lq2	3397.074938	2007	1.692613	590.802392	0.504971	c
lq2	803.925037	2007	0.400561	2496.501423	0.524044	w
lq3	11.338949	1	11.338949	88.191594	1.721144	c
lq3	1.219988	1	1.219988	819.680281	0.451088	w
lq3	29.592991	10	2.959299	337.917855	0.482082	c
lq3	3.762007	10	0.376201	2658.155777	0.509977	w
lq3	189.826965	100	1.898270	526.795547	0.521183	c
lq3	30.470848	100	0.304708	3281.825295	0.519037	w
lq3	1144.536018	1000	1.144536	873.716496	0.502110	c
lq3	281.628847	1000	0.281629	3550.772622	0.513077	w
lq3	747.119904	1434	0.521004	1919.370630	0.504017	c
lq3	399.667025	1434	0.278708	3587.986778	0.505924	w
lq4	443.204165	1	443.204165	2.256296	0.691175	c
lq4	197.706938	1	197.706938	5.057991	0.476837	w
lq4	290.827036	10	29.082704	34.384699	0.537157	c
lq4	200.902939	10	20.090294	49.775280	0.522137	w
lq4	704.112053	100	7.041121	142.022849	0.513077	c
lq4	239.194870	100	2.391949	418.069167	0.526905	w
lq4	2263.238907	1000	2.263239	441.844649	0.545025	c
lq4	624.146938	1000	0.624147	1602.186823	0.526905	w
lq4	4455.750942	10000	0.445575	2244.290610	0.514030	c
lq4	4439.900160	10000	0.443990	2252.302899	0.516891	w
lq4	47754.329920	100000	0.477543	2094.050951	0.513077	c
lq4	42808.356047	100000	0.428084	2335.992531	0.524998	w
lq5	10.285854	1	10.285854	97.220898	1.472950	c
lq5	1.781940	1	1.781940	561.185978	0.658989	w
lq5	24.019003	7	3.431286	291.435911	0.658989	c
lq5	6.473064	7	0.924723	1081.404346	0.673056	w

Table A.1: Queries LQ1 - LQ4 on the LUBM Dataset with 100 Universities

A.2.1 Results

A.3 LUBM Queries

```
queries = {}
prefix = """
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
"""

queries["lq1"] = '''
SELECT ?department WHERE
{
        ?researchGroups ub:subOrganizationOf ?department .
        ?department ub:name '"Department1"' . } LIMIT 100'''

queries["lq2"] = '''
SELECT ?mail ?phone ?doctor WHERE
{
        ?professor ub:name '"FullProfessor1"' .
        ?professor ub:emailAddress ?mail .
        ?professor ub:telephone ?phone .
        ?professor ub:doctoralDegreeFrom ?doctor .
         }'''

queries["lq3"] = '''
SELECT ?studentName ?courseName WHERE {
     ?student ub:takesCourse ?course .
     ?course ub:name ?courseName .
     ?student  ub:name ?studentName .
     ?student ub:memberOf <http://www.Department1.University0.edu> . }'''

queries["lq4"] = '''
SELECT ?publication ?author ?department ?university
WHERE {
        ?publication  ub:name '"Publication0"' .
        ?publication  ub:publicationAuthor ?author .
        ?author ub:worksFor ?department .
        ?department ub:subOrganizationOf ?university .
} LIMIT 100 '''


queries["lq5"] = '''
SELECT ?university ?name ?tel WHERE {
        ?student ub:advisor ?advisor .
        ?advisor ub:worksFor ?department .
        ?department ub:subOrganizationOf ?university .
        ?student ub:name ?name .
```

```
        ?student ub:telephone ?tel .
        ?student ub:takesCourse <http://www.Department1.University0.edu/
            GraduateCourse33> .
}LIMIT 100'''
```

List of Figures

Bibliography

[Atre et al., 2010] Atre, M., Chaoji, V., Zaki, M., and Hendler, J. (2010). Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the 19th international conference on World wide web*, pages 41–50. ACM.

[Atre and Hendler,] Atre, M. and Hendler, J. BitMat: A Main Memory Bit-matrix of RDF Triples. In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, page 33.

[Basca and Bernstein, 2010] Basca, C. and Bernstein, A. (2010). Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In *Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems*.

[Brickley, 2004] Brickley, D. (2004). RDF vocabulary description language 1.0: RDF schema. *http://www.w3.org/tr/rdf-schema/*.

[Gilbert and Lynch, 2002] Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):59.

[Guo et al., 2005] Guo, Y., Pan, Z., and Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182.

[Kim et al., 2009] Kim, C., Kaldewey, T., Lee, V., Sedlar, E., Nguyen, A., Satish, N., Chhugani, J., Di Blas, A., and Dubey, P. (2009). Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389.

[Kuznetsov et al., 2010] Kuznetsov, V., Evans, D., and Metson, S. (2010). The CMS data aggregation system. *Procedia Computer Science*, 1(1):1529–1537.

[Myung et al., 2010] Myung, J., Yeon, J., and Lee, S. (2010). SPARQL basic graph pattern processing with iterative MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, pages 1–6. ACM.

[Schmidt et al., 2010] Schmidt, M., Meier, M., and Lausen, G. (2010). Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*, pages 4–33. ACM.

[Senn,] Senn, J. String-Based Semantic Web Data Management Using Ternary B-Trees.

[Senn, 2010] Senn, J. (2010). Parallel Join Processing on Graphics Processors for the Resource Description Framework. *ARCS 2010*.

[Vidal et al., 2010] Vidal, M., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., and Polleres, A. (2010). Efficiently Joining Group Patterns in SPARQL Queries. *The Semantic Web: Research and Applications*, pages 228–242.

[Weiss and Bernstein,] Weiss, C. and Bernstein, A. On-disk storage techniques for Semantic Web data-Are B-Trees always the optimal solution? In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, page 49.

[Weiss et al., 2008] Weiss, C., Karras, P., and Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019.