Masaryk University
Faculty of Informatics

OData server endpoint for Infinispan

Diploma thesis

Tomáš Sýkora

Brno, 2014

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Sýkora

Advisor: Mgr. Marek Grác, Ph.D.

Acknowledgement

I would like to thank Mgr. Marek Grác, Ph.D., for official thesis advisory, precise answers for questions connected to official matter and review of thesis' text part; Ing. Martin Genčúr, for advice and leading from technical point of view, and for always being ready to help me; JDG QE team for fantastic every-day working atmosphere in place where I was able to get in touch with awesome technologies and for their support, including discussions on lunch and in the kitchen.

Additionally, I would like to thank Vitalii for many interesting brainstorming sessions and motivation; Michal for all answers around Infinispan servers, performance aspects and automation; Anna for a discussion about Infinispan queries; and Radim for precise answers for all of the tricky questions; Infinispan community developers, especially Adrian Nistor for discussion around Infinispan queries and providing very helpful insight, Sanne Grinovero for his advice and thorough information about performance aspects of Infinispan queries, and Manik Surtani for this very interesting thesis topic.

Many thanks to my close friends Jiří Sviták and Lukáš Žilka for suggestions and support; and to Zuzana Bzonková for help with English, which is still a great challenge for me.

Finally, I would not finish this thesis without infinite patience of my girlfriend Katka and endless support of my parents, thank you!

Abstract

We developed Infinispan OData server for storing, indexing and querying JSON documents maintained in Infinispan caches through Open Data Protocol standards and query language. Odata4j framework was improved for needs of Infinispan OData server. We provide performance benchmarks of Hot Rod, Memcached, REST and OData Infinispan servers in a clustered laboratory environment, with the use of PerfCake.

Shrnutí

Infinispan OData server byl vyvinut za účelem efektivního vkládání, indexace a dotazování nad JSON dokumenty uloženými v Infinispan cachi. Zároveň je použit dotazovací jazyk a přístup definovaný Open Data Protocol standardem. Byla provedena vylepšení v odata4j frameworku pro potřeby Infinispan OData serveru. Pro vzájemné výkonnostní porovnání Hot Rod, Memcached, REST a OData Infinispan serverů zapojených do klastru byl použit PerfCake a serverová laboratoř.

Keywords

Infinispan, OData, Open Data Protocol, NoSQL, odata4j, PerfCake, OData Jersey, JSON, Document store, Key-value store, Querying, Performance, Infinispan cakery

Contents

1 Introduction

Application and customer service providers face a continuously growing number of internet users. From architectural point of view, this fact brings web applications, that tend to become successful, to new challenges and common relational databases no longer fit current business needs of large companies. Solution architects, designers and developers are taking *NoSQL*[1] solutions into account more frequently, as they must react quickly to the growing market and their work is to satisfy the customer needs.

The thesis introduces one of the NoSQL solutions – *Infinispan*, which strives to become a standard solution for data caching. In order to alleviate a load on database machines, a cache, that is placed on the way between application and database layer, collects results of frequent queries into a fast accessible memory.

The community of Infinispan users continue growing. This caching solution, mainly focused on Java users, is used worldwide and in crucial and business-critical projects. As it is described in the section dedicated to Infinispan, this NoSQL store also supports *REST*[2] interface for client-server communication. The main idea behind this is to provide a uniform access for variety of clients independent of a programming language in use. Infinispan is in its core a key-value store, which serves as a very rapid database for storing objects specified by their unique keys.

As it is demonstrated later, Infinispan can also act as a document store when the specific configuration is enabled. Unfortunately, Infinispan REST server does not support querying functionality, which is essential for communication with embedded document store.

Together with the process of overall Infinispan spreading, there arose some users in the community who start requesting *JSON*[3] document store functionality and start asking for a possibility of querying over JSON fields. JSON as a format of document store entities is not supported yet (see figure 1.1), and this is one of the Infinispan's challenges – to make it more portable and ergonomic. Users already using JSON

1. NoSQL: Not only Structured Query Language
2. REST: Representational State Transfer
3. JSON: JavaScript Object Notation

are not willing to change their architecture, however, they are eager to use a greater variety of features that Infinispan offers.

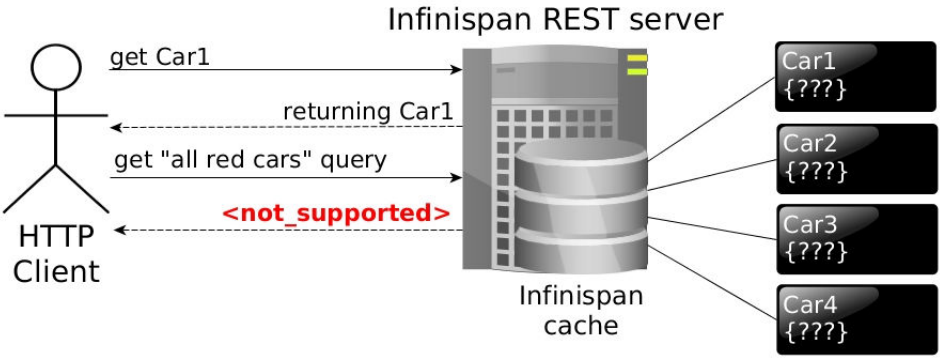

Figure 1.1: Infinispan REST server stores JSON documents simply as key-value entries without understanding an internal structure of JSON fields. Stored objects (values) are accessed purely via unique entry keys.

The thesis is aimed at providing access to an embedded Infinispan caches via *OData*[4] protocol. The leading idea and the main advantage is that users will be able to communicate with Infinispan caches using any client code which is able to follow OData standards and utilize OData query language.

NoSQL, in general, lacks overall portability and consequently OData was chosen as a protocol for supporting a standardized interface and accessing the document store functionality. There is already a REST server implemented in Infinispan, and OData server should add something on top of that module and provide state-of-the-art solution for wider utilization of Infinispan's document store capabilities, as well as indexing and data replication.

As a response to above-mentioned requests, we decided to implement standalone Infinispan OData server (see figure 1.2) in order to find out how community will react to this kind of solution, and gather feedback as soon as possible.

4. OData: Open Data Protocol

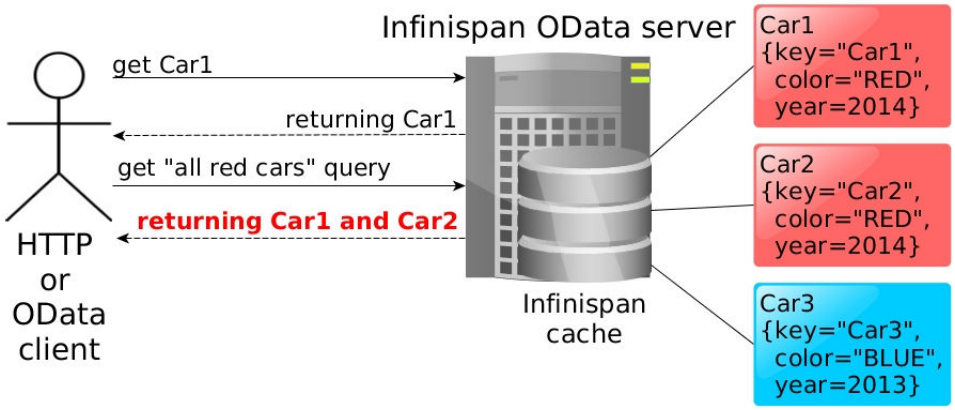

Figure 1.2: Infinispan OData server will be capable of understanding an internal structure of stored JSON documents. This hybrid solution will allow clients to query an Infinispan cache using a standardized OData query language, or classic key-value approach.

The principal intention of this thesis is to design and implement standalone Infinispan OData server. This client-server solution will make Infinispan's document store accessible, using standard *HTTP*[5] protocol, and will also provide a good starting point for community response in the matter of adapting to and implementing OData standards. Together with a description of a design and implementation, the thesis is concentrated on performance aspects of this new standalone server. Because of the fact that Infinispan is a data grid and caching solution, performance measurement and result summary are provided as well.

5. HTTP: Hypertext Transfer Protocol

2 NoSQL

This chapter summarizes fundamentals of two leading approaches for storing data and provides a theoretical background together with comparison of both RDBMS and NoSQL systems. Sections in this chapter are also dedicated to current business trends; to a question why and when to choose NoSQL; and to a classification of NoSQL stores.

2.1 RDBMS

Before elaboration of NoSQL phenomena it is firstly important to introduce Relational Database Management System (RDBMS) and provide a source for a needed technical background.

RDBMS is based on relational algebra and data model where relational calculus plays an essential role during the process of storing, describing, maintaining, and operating with a data [1]. The data is stored in rows and aggregated in constructs which are called *tables*, where *primary key* acts as unique identifier for an included tuple. For a data manipulation and querying functionality is usually used Structured Query Language (SQL).

It is recommended to get acquainted with important RDBMS-related terms for complete understanding of following sections in this chapter. *Fundamentals of relational database management systems* book [1] provides comprehensive overview and background for this purpose in chapters: *Overview of Database Management System*, *Entity-Relationship Model*, *Relational Model* and *Structured Query Language*.

2.2 NoSQL introduction

The primary purpose of this chapter is to provide introduction into the world of state-of-the-art phenomenon called NoSQL which currently makes the world go round.

NoSQL as "*Not Only SQL*" can be understood as a set of tools used for effective solving of data stores scaling and storing problems [2]. It does not matter whether issues are solved by use of typical SQL approach or by implementing NoSQL key-value store. What always matters is the right application of a particular solution which can deal with

specific problem in an effective way. Therefore, the whole philosophy of NoSQL is not about rejecting *RDBMS* systems, but rather about the right choice for running successful business.

Additionally, it is crucial to articulate that pure NoSQL approach is not an panacea for all kinds of problems and classic RDBMS solution can better fit application and business needs.

In order not to be stuck with only one opinion, [3] defines NoSQL exactly as: "*NoSQL is a set of concepts that allows the rapid and efficient processing of data sets with a focus on performance, reliability, and agility.*"

However, according to [4], NoSQL clearly covers only cases where classic RDBMS principles are not followed.

There is also a very nice overview of how NoSQL should be understood and how not. Trying to express these thoughts in different words would definitely harm the perfect meaning of those few points stated in [3]. For the purpose of this thesis and for the sake of brevity, characteristics are enlisted without descriptions:

What is NoSQL?

"*It's more than rows in tables; It's free of joins; It's schema-free; It works on many processors; It uses shared-nothing commodity computers; It supports linear scalability; It's innovative.*"

It's also important to say what NoSQL is not.

"*It's not about SQL language; It's not only open source; It's not only big data; It's not about cloud computing; It's not about a clever use of RAM and SSD; It's not an elite group of products.*"

It is also important to mention what started the era of the NoSQL phenomenon and look at wider chronological context. A book *Professional NoSQL* [4] contains a very good overview of use cases and historical background behind NoSQL movement. It mentions that Google started to fulfill and implement complex solution for their needs of processing enormous amount of data using parallel processing with support of easily scalable architecture. It was needed for applications like GMail, Google Finance, or Google Maps. Google also published parts of its research in set of interesting papers and spread the word about the whole concept, which was a driving engine of early forming open-source

community, and for instance developers of *Lucene*[1] were inspired by those papers. Later, *Hadoop*[2] imitated Google's distributed infrastructure, and this initiative started and developed solid ground for NoSQL movement. After Google, Amazon came out with their own solution: *Amazon Dynamo*[3] was presented in 2007. These two significant projects of leading companies literally catalyzed a number of new projects and users adopting NoSQL solutions.

That was a very short introduction into the world of NoSQL with plenty of new terms connected to it. The terms will be discussed progressively in subsections which provide a chapter dedicated to current trends and elaboration on the question why not only relational solutions are chosen among developers. After that, NoSQL stores are divided into four main categories and each of them will be shortly described. Finally, some hints how to choose the right NoSQL solution will be shown and will help to decide what fits well for which kind of business and application.

2.3 Current trends

Internet itself has changed significantly during the period of last twenty years. According to Couchbase paper [5], web applications have been changing rapidly as well, and the reason is that applications need to deal effectively with three basic facts: When business and consequently web applications encounter *increasing number of users*, they still need to *keep service response time in values of milliseconds*. More and more users means bigger and *fast growing amount of data* that can be gathered for further processing and analyzing. Applications providing complex functionality for users can be improved by special mechanisms, which usually depend on collection of both structured and semi-structured data.

With respect to above-mentioned three points, current applications need to be able to serve responses very quickly to satisfy the business needs. Traditional RDBMS architecture can provide good solution in

1. Apache Lucene, search engine, Home page:
<`http://lucene.apache.org/`>
2. Home page: <`http://hadoop.apache.org`>
3. Home page: <`http://aws.amazon.com/dynamodb/`>

the case of single and well-tuned machine usage, however, this weak point can easily turn into serious bottleneck during peak hours when system is experiencing growth in users accessing services concurrently.

Three major phenomenons causing progressive move to NoSQL solutions can be recognized: *big users, big data and cloud computing.*

- **Big Users** – With overall expansion of the internet and still growing number of users capable to access the network even from their mobile devices, web applications need to deal with approximately three billion users on-line. Also, it is necessary to consider special peak times, which last for a short or mid-term amount of time. These peak times can be experienced typically before Christmas or Valentine's day.

  RDBMS solution might not be prepared enough for dealing with those visitor pikes and, additionally, this approach as it is architecturally built is not prepared for scaling. Therefore, many companies have started to examine and later implement NoSQL stores to improve applications and to be able to react more flexibly by using scalable (more in sections 2.4 and **??**) database technologies.

- **Big Data** – Continuously more and more data appears worldwide because of growing number of on-line users, and this causes the arising of a possibility for capturing and gathering more interesting and business related data. Today's ratio of structured data is only 20 % and NoSQL performs better than classic RDBMS solutions when working with semi-structured data or structure-less data [5].

  Applications need to be adapted to this new kind of data so that they can process it and effectively return requested results. Blog posts, tweets, Facebook status updates or various log files can be mentioned as proper examples of data without any solid structure. Databases without easily changeable schema will encounter problems during the intention of storing data of vague structure, and that is why flexible solutions are being chosen for driving more responsive and consequently successful business.

- **Cloud Computing** – It was usual that business applications were being run on a single PC and were interacting with insignificant number of users. Two-tier architecture was chosen in cases where applications needed to be prepared for concurrent access of more users. This kind of solution is well known as a client-server architecture.

  However, current trend seems to be different and three-tier systems, with support of NoSQL databases, started to be implemented in situations where applications need to deal effectively with really huge amount of users.

  The main advantage is much easier scaling ability when running the system in the cloud architecture. During the growth of concurrent users accessing service, new machines may be plugged into the cloud system in order to provide better distribution of a load among more machines.

2.4 Why NoSQL?

A migration from typical RDBMS systems to NoSQL systems is experienced due to a couple of reasons. [5] mentions and elaborates on three of them, which seem to be most important, and provides possible answers to the question: *Why are developers choosing NoSQL solutions?*

- **Because of more flexible data model** – As was briefly introduced in 2.3, relational databases use non-flexibly defined schema, and this tends to be in contradiction with the requirement to react to and work with unstructured data.

  Moreover, programmers use OOP[4] techniques and they need to put extra effort into the "translation" of data from RDBMS system into the objects which they are using directly in the code logic. This mapping from tables, rows and columns into objects is often demanding. Having a possibility to get the whole object easily and directly from data store could alleviate necessary effort and make code more robust and less bug prone.

4. Object Oriented Programming

Additionally, requested information is gathered through many database tables using JOIN[5] statements. This can be problematic, in case of very big schema, for reading data and for writing data into the store as well.

That is the place where NoSQL databases offer a different solution. For example, document-oriented NoSQL store (see 2.6.4) is able to store whole JSON[6] documents without the need of changing their structure. Later, it is possible to get these big objects from database and use them directly in application logic.

Such a JSON document can aggregate data from multiple database tables (see figure 2.1) and it is not needed to use expensive JOINs anymore. Of course, this approach has its downside too – there will probably exist data redundancy as a particular information will be carried over a pile of documents.

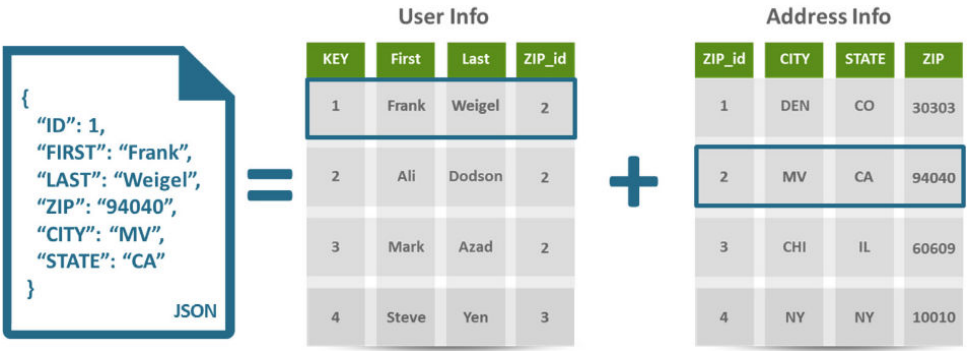

Figure 2.1: Aggregated information in JSON document [5].

- **Because of scalable solution possibilities** – Two different scaling approaches can be distinguished in dependence on the way how compute power is added into the system: *scale up* and *scale out* (see figure 2.2). During scaling-up process, compute power of a single machine is increased; this solution which is typically used for big database servers will sooner or later become a bottleneck. Hence, scaling-out approach is trying to deal

5. SQL JOIN clause, <http://www.w3schools.com/sql/sql_join.asp>
6. JavaScript Object Notation, <http://www.json.org/>

with this kind of problem by adding more mutually connected machines into the system[7].

NoSQL databases are designed and ready to serve as a large cluster of machines, which is capable to evenly distribute the load across the whole cluster and react dynamically to the increasing load.

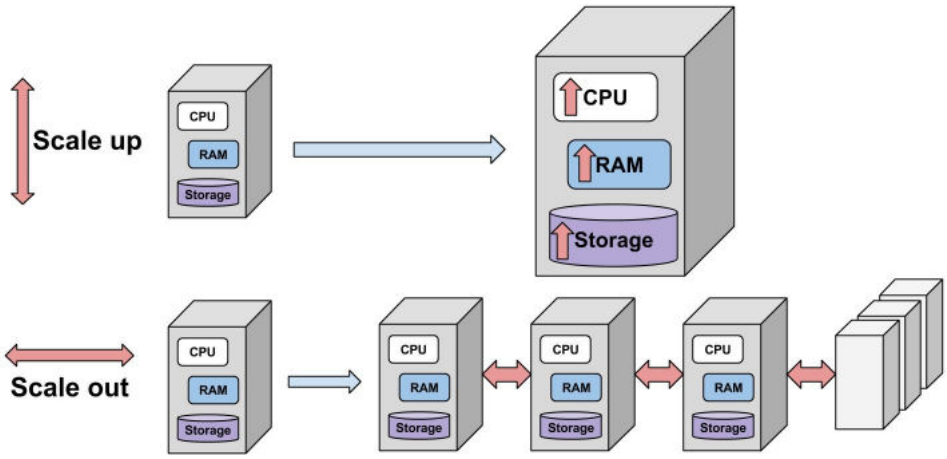

Figure 2.2: Vertical and horizontal scaling, schematic drawing.

When talking about clusters it is common to meet terms like *scalability*, *high availability* and *fault tolerance*. Chapter one in a book about Oracle Coherence [7] is dedicated to scalability and high availability, and chapter three mentions and explains fault tolerance.

- **In order to gratify needs of users** – Developers are also choosing NoSQL solutions because users expect fast and responsive web applications. This can be achieved more easily by using flexible database model which is well designed for scaling.

Besides *big data*, *flexible data model* and *performance*, [2] mentions some other reasons for taking NoSQL solutions into account during

7. Cloud system is a well-designed architecture for this type of scaling.

a decision phase of a project: *continuous availability*, *data location independence*, *modern transaction capabilities* and *overall a better architecture*.

Modern systems start adapting instruments capable of supporting continuous availability, which means that no down time is expected and the whole system will not go down even in the time of maintenance.

Data location independence starts to be important for global, worldwide business. Customers around the whole world expects fast access to the same pieces of data and this scenario is not easily reachable by using RDBMS databases.

Implementation of NoSQL advantages will have positive impact upon the overall architecture of applications and will prepare them for the smooth process of a scaling, so they can support continuous, or at least high availability and dynamic data distribution by using more flexible schema, which fits better for various types of data and use cases.

2.5 NoSQL and RDBMS comparison

RDBMS approach is designed to save space, and in this case, the data is maintained usually only in one row or column in a particular table. However, as a counterpart, this brings problems when it comes to getting the data from the database. It is often necessary to join pieces of information coming from a bundle of tables to get the final result. These operations may be very expensive and this issue arises also during the process of writing data into several tables.

It was shown in the figure 2.1 that JSON document-oriented store is able to solve this kind of problem while facing the other side – possible data redundancy.

Figure 2.3 is worth thousand words and clearly depicts architectural differences between NoSQL and RDBMS systems.

As it can be observed, NoSQL approach is dealing with complex functions in the middle tier which alleviates complexity of database tier. Without the need of providing this functionality, database tier can be fully and directly focused on storing the data.

In traditional RDBMS systems, database tier which is loaded with overweight functions can easily turn into a bottleneck and negatively impact performance and scalability possibilities of this architecture.

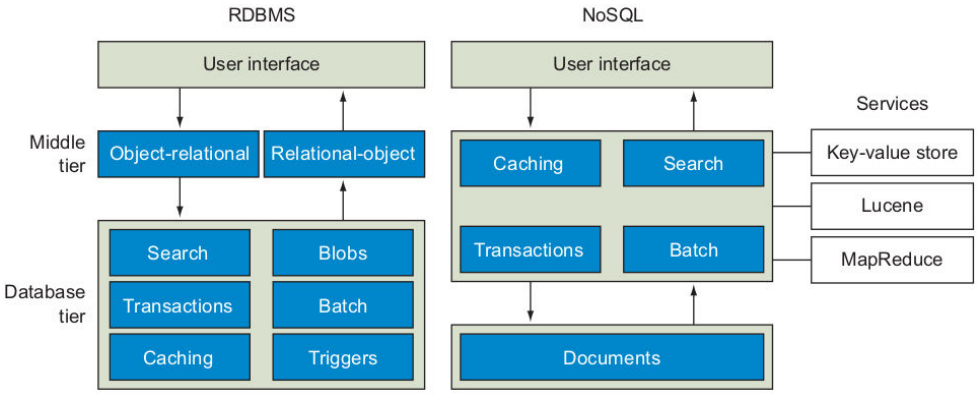

Figure 2.3: Architectural differences of RDBMS and NoSQL systems [3].

Generally, NoSQL is about to keeping the building components reusable and as simple as possible. From a developer's point of view, simpler components and application design lead directly to less source code to write, better understanding of code, easier testing of a new application and improved capability of portability to a new architecture.

Last point – *application portability* – is the place where RDBMS systems wins. There is an intention in NoSQL projects to develop SQL-like standards to make overall progress in the matter of portability.

RDBMS also fits better the system environment which is full of very strict transactions and where ACID[8] principle takes place. On the other hand, when rigid ACID is not needed, it is possible to utilize NoSQL mechanisms for handling data consistency (*eventual consistency* and *CAP theorem* [3]).

See page 19 and 20 in [3] for comprehensive summary of pros and cons of RDBMS and NoSQL solutions. *Professional NoSQL* book [4] mentions other common problems and restrictions that RDBMS systems need to deal with and suggests NoSQL as an alternative solution.

RDBMS database schema presumes that working set of data is structured without any big variability in uniformity, and therefore, any change in the schema is associated with painful data migration.

Such a database is also full of indexes and references between tables to make quick querying possible. Everything seems to be all right until the time of denormalization process arrives, and during which RDBMS

8. ACID – Atomicity, Consistency, Isolation and Durability [1].

systems try to support better scaling and tend to look more „*NoSQL-like*“. NoSQL is more flexible, scalable and ready to deal with various types of data. However, as a counterpart it loses transactional integrity and possibility of flexible indexing in an effective querying mechanism.

2.6 NoSQL stores classification

Generally, NoSQL stores are divided into four main categories according to their primary usage, approach to storing data entries, querying possibilities and some other features [3; 4; 6]. Following list is not complete, but should be sufficient for this introductory chapter: *key-value stores*, *graph stores*, *column stores* and *document stores*.

2.6.1 Key-value stores

One of the best ways how to introduce the functionality of the key-value NoSQL store is using an example of a typical lexical dictionary. A particular word can be deemed as a key and its translation as a value. Thus, every key has its mapped value and the keys are unique across the whole data set.

These stores are commonly used as a HashMap collection with $\mathrm{O}\big(1\big)$ average asymptotic computational complexity for data access [4]. It makes this architecture ideal for very fast data retrieval, regardless of the amount of stored entries. Additionally, the type of stored values does not have to be exactly specified. This fact together with overall simplicity[9][10] makes from the key-value stores very popular solution, easy to set up and able to spare developers' time.

Typical examples of the key-value stores: *Redis*[11], *EHCache*[12], *Infinispan*[13] *Berkley DB*[14].

---

9. Key-value stores provide easy-to-use interface supporting put, get, remove and update functionality which has a positive impact on application portability.
10. As a downside, key-value stores do not usually support querying over the values.
11. Home page: <`http://redis.io/`>
12. Home page: <`http://ehcache.org/`>
13. Home page: <`http://infinispan.org/`>
14. Product page:
<`http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html/`>

2.6.2 Graph stores

Graph databases may be successfully used when there is a need to store not only data itself but also additional information about data linkage. This kind of NoSQL stores works well when relationships between objects present in a graph structure should be analyzed. The graph stores are typically utilized for fast search of connection patterns present in the data gathered from social networks.

Whereas the key-value stores consist of only two fields (key and value), graph stores are based on usage of three fields (see figure 2.4). *Nodes*, *relationships* and *properties* create a graph with internal logical structure which is ready for deep analysis and queries [15].

Each node in a graph structure is usually closely connected to a few other nodes around which brings a disadvantage – graph stores do not scale out well.

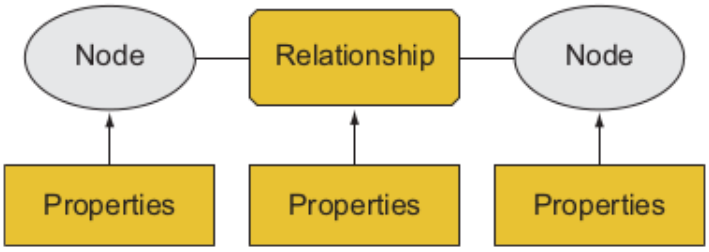

Figure 2.4: Schematic structure of data stored inside of the graph databases. Properties can describe nodes and also their mutual relationships [3].

Classic instances of graph stores:
Neo4j[16], *FlockDB*[17], *InfiniteGraph*[18].

2.6.3 Column stores

Column databases use column-oriented approach to storing data without any mandatory or fixed structure, which allows that names of

15. The result of a simple graph store query can, for example, show nearest neighbors of a particular node.
16. Home page: <http://neo4j.org/>
17. Home page: <https://github.com/twitter/flockdb/>
18. Home page: <http://www.objectivity.com/infinitegraph/>

columns do not have to be predefined. Additionally, no column is stored for *null* value, and thus it saves space. This is in contrast with classic RDBMS systems that store data in rows.

Column databases can serve as a good tool for updating stored documents, because only really needed pieces of data are actually loaded into the memory. It is possible to change value in the specific column without a need of reading the whole row in which the column is stored.

The way how exactly data is stored in this type of NoSQL databases is quite complex and well explained in [4], accompanied by many practical use cases.

Typical representatives of column stores:
HBase[19], *Cassandra*[20], *Cloudata*[21].

2.6.4 Document stores

In this last type of NoSQL stores, keys are usually very small and used only rarely. What really matters are values; better to say *documents*. In the meaning of the NoSQL document store, a stored *document* is a whole large object, usually in *XML*[22] or *JSON*[23] format, which internally includes a list of key-value pairs. These fields are indexed during the process of document storing and thus ready for later querying.

Documents with the same meaning are usually kept in one collection despite the fact that this approach lacks logical cause, it is possible to store documents with different structure and schema in one collection.

Document object can be quite huge and its structure complex, but *querying API*[24] remains simple, easy-to-use and well-prepared for the query filtering requested subset of documents. Key-value stores do not usually provide this functionality, and values are accessed strictly by using their unique keys. To obtain a value, it is needed to know the key. Document stores solve this issue and are able to return a collection of results based on the query over documents (values).

---

19. Home page: <`http://hbase.apache.org/`>
20. Home page: <`http://cassandra.apache.org/`>
21. Home page: <`http://www.cloudata.org/`>
22. Extensible Markup Language, <`http://www.w3.org/XML/`>
23. JavaScript Object Notation, <`http://www.json.org/`>
24. Application Programming Interface; providing querying functionality of a particular data store.

Characteristic instances of document stores:
MongoDB[25], *CouchDB*[26], *Terrastore*[27].

Hybrid implementations of NoSQL stores can also exist to deal with special use cases and these are beyond the scope of the thesis. Comprehensive list of NoSQL databases can be found at <`http://nosql-database.org/`>.

2.7 Choosing the right NoSQL solution

Both NoSQL and RDBMS databases have its pros and cons which need to be carefully considered with respect to desired functionality and system architecture. In cases when NoSQL is decided as the suitable solution, there exists a huge amount of products on the market and it is definitely not an easy task to choose the right one.

Pattern name	Description	Typical uses
Key-value store	A simple way to associate a large data file with a simple text string	Dictionary, image store, document/file store, query cache, lookup tables
Graph store	A way to store nodes and arcs of a graph	Social network queries, friend-of-friends queries, inference, rules system, and pattern matching
Column family (Bigtable) store	A way to store sparse matrix data using a row and a column as the key	Web crawling, large sparsely populated tables, highly-adaptable systems, systems that have high variance
Document store	A way to store tree-structured hierarchical information in a single unit	Any data that has a natural container structure including office documents, sales orders, invoices, product descriptions, forms, and web pages; popular in publishing, document exchange, and document search

Figure 2.5: Summary table for NoSQL store types with short description and typical usage for each type [3].

25. Home page: <`http://www.mongodb.org/`>
26. Home page: <`http://couchdb.apache.org`>
27. Home page: <`https://code.google.com/p/terrastore/`>

Table in figure 2.5 provides a brief summary of typical use cases where NoSQL stores are used, divided according to above-mentioned NoSQL stores classification.

It is out of scope of this thesis to provide comprehensive overview of various NoSQL products and to compare them. However, chapter fourteen in *Professional NoSQL* book [4] provides very helpful guidance in a matter of choosing the right NoSQL solution.

The thesis is concentrated on a particular NoSQL key-value and document store solution – Infinispan – which is introduced in next chapter.

3 Infinispan

The exact quote from Infinispan project landing page[1] offers a compact specification of this NoSQL technology [9]:
„*Infinispan is an extremely scalable, highly available key/value NoSQL datastore and distributed data grid platform - 100% open source, and written in Java. The purpose of Infinispan is to expose a data structure that is highly concurrent, designed ground-up to make the most of modern multiprocessor/ multi-core architectures while at the same time providing distributed cache capabilities. At its core Infinispan exposes a Cache interface which extends java.util.Map. It is also optionally backed by a peer-to-peer network architecture to distribute state efficiently around a data grid.*“

Infinispan replicates and evenly distributes data entries between nodes in a cluster as well as brings a possibility for configuration of persistent stores, so the data is not hold only in the memory but also persisted on a physical disk or in an underlying database. Moreover, it come up with mechanisms for maintaining memory usage by using eviction and expiration [8].

Infinispan, as a key-value store from basis, supports also functionality for document store behavior. Capabilities of querying over values are more elaborated in section 3.5. Its schema-less principle is very flexible and old-stored document can be very easily replaced by a new version of the same document or a completely different file. Therefore, with arrival of a new document structure, there is no need for changing the schema. This is one of the tremendous advantages of NoSQL stores.

Infinispan is able to serve as a *cache* or *data grid platform*. Besides others, Infinispan is strongly connected to terms like scalability, high availability and failure tolerance.

3.1 Interacting with Infinispan

InVM[2] running mode is the first possibility how users can interact with Infinispan caches [10], where both Infinispan and an application

1. Home page: <`http://infinispan.org/`>
2. InVM – inside of Java Virtual Machine.

are running in the same virtual machine. This mode is also called as *embedded* or *library mode*, because Infinispan libraries are directly embedded and used in application as *jar files*[3]. It is possible to meet all three terms in practice and they are used with the same meaning.

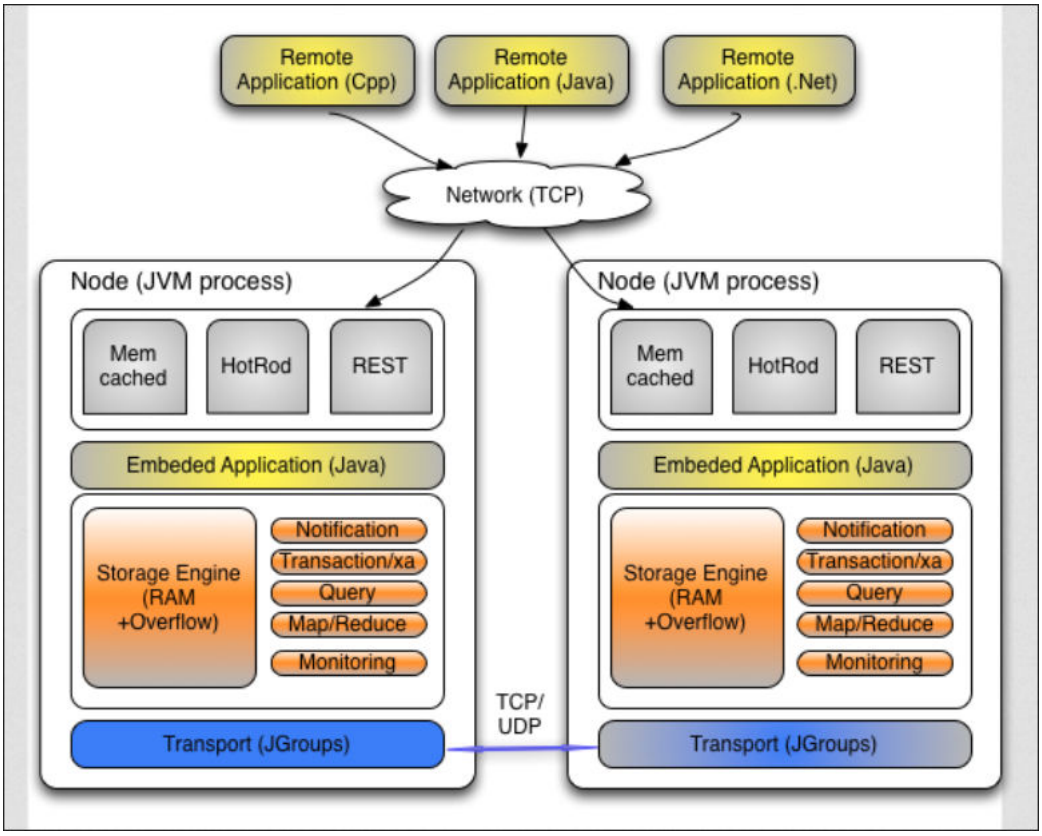

Figure 3.1: Schematic figure of Infinispan common usage and system architecture [11].

The yellow oval in the figure 3.1 represents an application using Infinispan in the *library mode*. Infinispan itself with its functions is colored orange.

Transport layer (blue color) is elaborated in next section 3.2, before short introduction of clustering modes.

3. Java ARchive file, for example: infinispan-core-6.0.0.Final.jar.

Yellow colored remote clients together with gray colored protocol names (*Memcached, Hot Rod, REST*) represent the other possible communication approach with Infinispan, and it is a *client-server mode*, which is described thoroughly in section 3.3.

3.2 Clustering modes

Infinispan instances use JGroups, reliable multicast system [16], to ensure an inter-node communication. Infinispan nodes are able to detect each other on the network and automatically create a cluster. JGroups also takes care about entries replication.

Infinispan caches can be configured to run in four basic clustering modes, which affects behavior of an entry distribution across nodes in the cluster: *distribution mode*, *replication mode*, *invalidation mode*, and *local mode*. See section *6. Clustering modes* in *Infinispan User Guide* [12] for more details.

3.3 Client-server access

The focus of this thesis is put on a client-server mode, where clients are accessing the data in caches via special protocols, while Infinispan itself runs inside of the server virtual machine and utilizes its *JVM heap size*[4] to store entries.

Three main Infinispan server modules are supported: *Memcached*, *Hot Rod* and *REST* module. There is also an experimental *Websocket module*[5] implemented, but this solution is not widely used and for now is beyond the scope of the thesis.

Memcached server module

This module is implementation of the popular *Memcached protocol* [13], enhanced by Infinispan functionality for entry replication. Figure 3.2 depicts a scenario of a crash and consequent

4. Java virtual machine heap size: <http://docs.oracle.com/cd/E15523_01/web.1111/e13814/jvm_tuning.htm#i1141344>.

5. Infinispan Websocket server module documentation: <http://infinispan.org/docs/6.0.x/user_guide/user_guide.html#_infinispan_websocket_server>.

addition (join) of a node into the cluster of Infinispan Memcached servers. The data have survived on running nodes and will be replicated back to the joined node. In case of original Memcached servers, data would be lost. Note that clients need to update the list of servers manually.

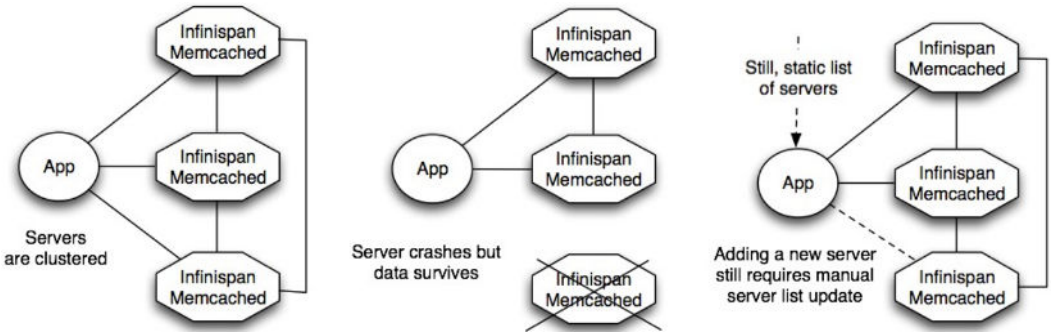

Figure 3.2: Node crash scenario in the Infinispan Memcached cluster [14].

Infinispan Memcached server does not support smart routing. It means that clients[6] may place a request to a node which is not an entry owner. In that case, data needs to be fetched over the network from another node (the true owner), so that can be returned back to the client (see figure 3.3). This operation costs time and affects the overall performance of a protocol in clustered environment.

How to use this module and other details is described in the Infinispan documentation [12].

6. List of Memcached clients: <`http://code.google.com/p/memcached/wiki/Clients`>.

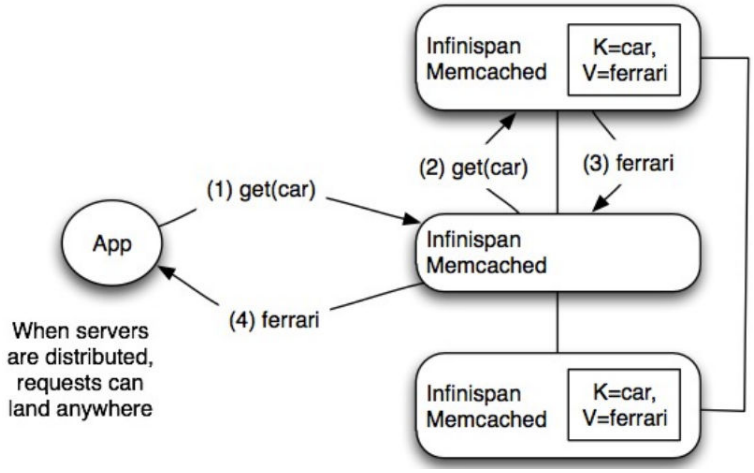

Figure 3.3: Scenario of a „not smart“ client get request [14].

Hot Rod server module

This server module implements Hot Rod[7], Infinispan-specific binary protocol [12], which is usually used in applications where speed really matters. Hot Rod clients are able to detect joined nodes automatically into the cluster and update server list accordingly (see figure 3.4).

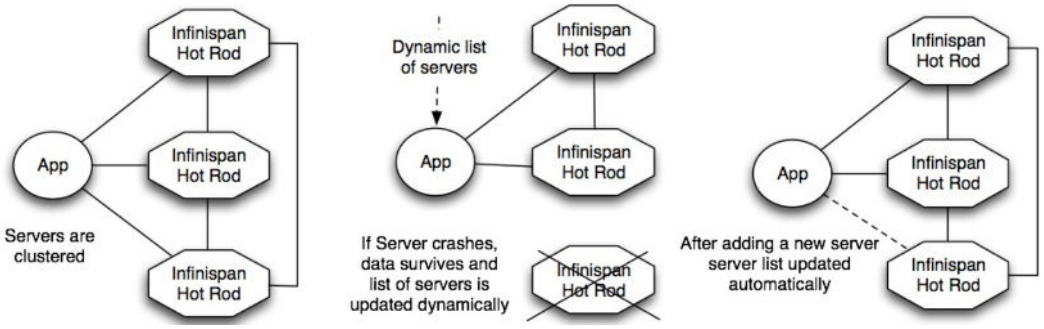

Figure 3.4: Node crash scenario in the Infinispan Hot Rod cluster [14].

7. Hot Rod protocol documentation: <http://infinispan.org/docs/6.0.x/user_guide/user_guide.html#_hot_rod_protocol>.

Infinispan Hot Rod protocol supports smart routing (see figure 3.5). The true *entry owner* is determined according to the entry hash[8], and clients exactly know which node to ask for the data. This mechanism ensures that there are no unnecessary entries fetched between nodes in the cluster over the network.

How to use this module and other details is described in the documentation[9].

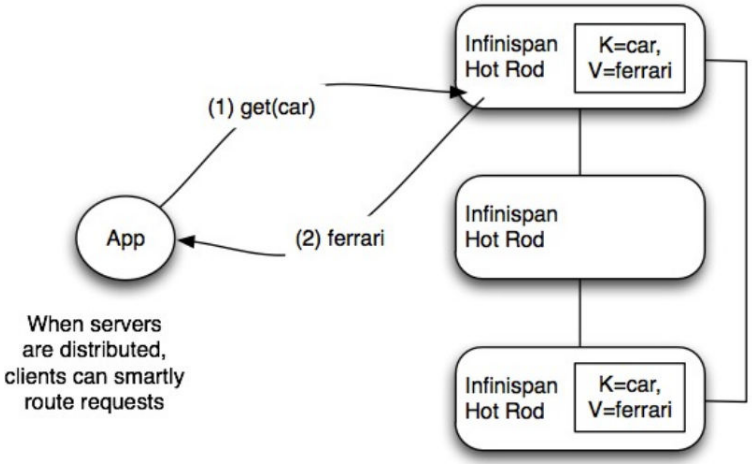

Figure 3.5: Scenario of a „smart“ client get request [14].

REST server module

The whole next section (3.4) dedicates special attention to the Infinispan REST server, because this module is crucial for the thesis' goals.

At the end of this section, table 3.1 summarizes properties of three aforementioned protocols.

8. HashCode() method of stored object is usually used to determine the location of the entry in the cluster.
9. Infinispan Hot Rod server documentation: <`http://infinispan.org/docs/6.0.x/user_guide/user_guide.html#_using_hot_rod_server`>.

Protocol	Type	Client libraries	Clustered	Smart
Hot Rod	Binary	Java, Python, C++	Yes	Yes
Memcached	Text	Plenty	Yes	No
REST	Text	Standard HTTP clients	Yes	No

Table 3.1: Summary of main protocols used for interaction with Infinispan [14].

3.4 Infinispan REST server module

This *JAX-RS*[10] implementation provides *RESTful*[11] HTTP access to Infinispan caches [12]. Infinispan REST server module is shipped as a `WAR`[12] file, which needs to be deployed to an application server. After successful deployment, Infinispan caches are started and are readily accessible for clients who would use a specific host and port (typically `localhost` for local testing purposes and `port 8080`). It is also possible to deploy more instances in order to create a cluster.

This module was implemented because of an overall idea to make Infinispan more platform independent. The main advantage of the Infinispan REST server is the support for diversity of common HTTP clients. Any application using HTTP client can interact with this server, and thus get direct access to Infinispan functionality.

Looking into the source code[13], Infinispan REST server is very lightweight. The whole core of this module is created basically by one class written in *Scala*[14]. Developers decided for the Scala because of an optimal performance impact and in order to try out new technology.

This straightforward approach has its downsides as well. REST server module does not support smart routing and clients need to be

10. Java API for RESTful services: <`https://jax-rs-spec.java.net/`>.
11. RESTful services: <`http://www.oracle.com/technetwork/articles/javase/index-137171.html`>.
12. Web application archive, <`http://docs.oracle.com/cd/E19316-01/820-3748/aduvz/index.html`>.
13. Infinispan REST server, source code: <`https://github.com/infinispan/infinispan/blob/master/server/rest/src/main/scala/org/infinispan/rest/Server.scala`>.
14. Scala programming language: <`http://www.scala-lang.org/`>.

manually aware about any change in the cluster topology (leaving and joining nodes) to know which node they should ask for entries.

The main intention was to provide easy-to-use and fast HTTP based access to Infinispan caches. Because of that reason, REST server offers just basic key-value store functionality and is not ready for working with document store capabilities (see next section 3.5), which is the main challenge of this thesis.

Configuration requisites and possibilities can be found in the documentation[15]. Some practical examples, for demonstration purposes, are listed below:

Putting entry into the cache:

`HTTP PUT method`

`http://localhost:8080/infinispan/rest/default/key1`

`infinispan/rest` is a path of the service.

`default` is a cache name.

`key1` is an unique key of a particular entry.

Entry `value` is set as a payload in the `body` of `HTTP POST` request. Also, `content-type` header have to be set properly, for example `"application/json"`.

Putting entry with usage of `HTTP PUT method` will cause update of any data (value) under the same key. `HTTP POST method` ensures different behavior; in case of conflict (key already exists in the cache) `HTTP CONFLICT status` is returned and entry will remain untouched.

Special HTTP headers:

It is also possible to set up special HTTP headers for change in the entry storage behavior. `content-type` header is mandatory, others are optional:

15. Infinispan REST server configuration guide: (<`http://infinispan.org/docs/6.0.x/user_guide/user_guide.html#_configuration_6`>.

`performAsync` (true) sets asynchronous behavior; request returns immediately and does not wait for data replication in the cluster.

Following two headers are associated with the *expiration*[16] Infinispan functionality:

`timeToLiveSeconds` specifies maximal lifetime of entries. Default value is `-1`, which means that entry never expires.

`maxIdleTimeSeconds` again, with default value `-1`, means that entry never expires. If entry is not get or updated for specified amount of time, it will be expired from the cached after that.

Getting entry from the cache:

`HTTP GET method`

`http://localhost:8080/infinispan/rest/default/key1`

`Value` for a particular key is returned in the `body` of `HTTP GET` response.

`HTTP HEAD method` called on the same address returns no content and this can be used for checking header fields.

`HTTP GET method`

`http://localhost:8080/infinispan/rest/default`

Note that no key is specified for now. This request for the whole `default` cache returns the list of all stored keys. Four possible values for `Accept` header can be passed to set up a format of the response: `application/xml`, `application/json`, `text/html` and `text/plain`.

16. Infinispan expiration feature: <`http://infinispan.org/docs/6.0.x/user_guide/user_guide.html#_expiration`>.

Removing entries from the cache:

`HTTP DELETE method`

`http://localhost:8080/infinispan/rest/default/key1`

This HTTP request will remove entry from the cache; specified by a key.

`http://localhost:8080/infinispan/rest/default/`

With specification of a cache (without the key), this request will clear all entries from the `default` cache.

3.5 Infinispan queries

This chapter shortly introduces querying functionality, which makes the Infinispan not only a key-value store, but also adds features of a document store. Therefore, Infinispan can be classified as a hybrid NoSQL store implementation.

In a typical, rigid key-value store, clients need to know the key to obtain the specific value[17]. But what if someone needs to filter for instance all users named John? Queries works well exactly for such a use case and allow the clients to select and filter entries according to the content, which is stored inside of their values. Key-value stores usually response to `get` request by returning just one particular value. The situation is different while using document stores. The result of a query can be a set or collection of more values meeting query criteria.

Infinispan supports two approaches of accessing querying functionality: *queries in library mode* and *remote queries over Hot Rod*, both based on *Apache Lucene*[18] and *Hibernate search*[19], using these technologies for indexing and consequent searching in cached documents [12].

Two possible uses (*library* and *remote queries*) of document store functionality will be shortly introduced below:

17. Hash functions, hash wheels and distribution algorithms take care of it.
18. Apache Lucene home page: <`http://lucene.apache.org/`>.
19. Hibernate search home page: <`http://hibernate.org/search/`>.

Queries in the library mode

This functionality can be used only by Java client code and for indexing pure Java objects (POJOs[20]). At the first place, it is needed to prepare a class for indexing. See listing 3.1 for a simple example of `Book` class [12].

Listing 3.1: Book class ready for indexed object

```
// Instances of class annotated by @Indexed will be indexed
@Indexed
public class Book {
    // @Field annotation will pick the field for indexing
    @Field String title;
    @Field String description;
    ...
}
```

If Infinispan cache is configured with `indexing enabled`, instances of class `Book` will be automatically indexed using specified `fields` during the process of putting entries into the cache. Next listing (3.2) shows the way how is possible to obtain query results [12].

Listing 3.2: Query call with results

```
// To obtain search manager from cache
SearchManager searchManager =
    org.infinispan.query.Search.getSearchManager(cache);
// queryBuilder provides easy-to-use fluent API
QueryBuilder queryBuilder =
    searchManager.buildQueryBuilderForClass(Book.class).get();

org.apache.lucene.search.Query luceneQuery =
    queryBuilder.phrase()
                .onField("description")
                .andField("title")
                .sentence("book about Infinispan")
                .createQuery();
```

20. Plain Old Java Object

```
// The query API accepts any Lucene Query
// and results can be restricted to specific class
CacheQuery query = searchManager.getQuery(luceneQuery,
    Book.class);

// To obtain results:
List objectList = query.list();

for (Object book : objectList) {
     System.out.println(book);
}
```

Since version 6.0, Infinispan comes with new query DSL (Domain Specific Language). Searching and indexing logic still remains coupled with *Apache Lucene* and *Hibernate search*, but the new API for queries is different and independent. Users do not need to construct low level *Apache Lucene* queries any more (as was used in listing (3.2). Listing 3.3 demonstrates usage of new query DSL [12].

Listing 3.3: Usage of new query DSL

```
// Location of new API
import org.infinispan.query.dsl.*;

// Again, it is needed to obtain searchManager
SearchManager searchManager =
    org.infinispan.query.Search.getSearchManager(cache);

// Use DSL query factory for the Query object construction
QueryFactory qf = searchManager.getQueryFactory();

// Search books with a title which contains the word
// "Infinispan"
org.infinispan.query.dsl.Query query = qf.from(Book.class)
     .having("title").like("%Infinispan%")
     .toBuilder().build();

// To obtain results:
List<Book> list = query.list();
```

This was just quick example. Infinispan query DSL offers much more: *filtering operators*, *filtering based on attributes of embedded entities*, *boolean conditions*, *nested conditions*, *projections*, *sorting* and *pagination*. More information about aforementioned options and other code samples can be found in the documentation[21].

After short introduction into Infinispan's querying feature available in *library mode*, this section also acquaints readers with the other option: *remote queries over the Hot Rod protocol*.

Remote queries

Functionality of remote queries over the Hot Rod protocol is present in the Infinispan since the project version `6.0.0.Beta1`. Adrian Nistor publicly announced this new feature and introduced details in his blog post [15].

Queries in the library mode suffers from one disadvantage – non-JVM clients are not able to use JAVA API of Apache Lucene. This problem was solved by new DSL (Domain Specific Language). There exist implementations of the Hot Rod client in some of the non-Java languages[22]. and DSL is implementable as well. This approach opens up the functionality of remote queries over the Hot Rod.

Other important fact is that structure of stored data need to be known to both client and server, which need to understand common encoding format. *Google's Protocol Buffers*[23] (*Protobuf* in short) has been chosen as a common format for encoding entities.

For successful usage, it is required to define entity structure using `.proto` file. See very short example [12] of such a descriptor file below:

21. Infinispan's query DSL documentation: <`http://infinispan.org/docs/6.0.x/user_guide/user_guide.html#_infinispan_s_query_dsl`>.
22. Hot Rod client implementations:
<`https://github.com/infinispan/cpp-client`>,
<`https://github.com/infinispan/python-client`>,
<`https://github.com/infinispan/ruby-client`>,
<`https://github.com/infinispan/dotnet-client`>.
23. Google's Protocol Buffers: <`http://code.google.com/p/protobuf/`>.

```
package book_sample;

message Book {
  required string title = 1;
  required string description = 2;
}
```

Then, `.proto` file needs to be compiled into `.protobin` binary descriptor[24] and the output registered by `ProtoStreamMarshaller` instance of `RemoteCacheManager`, as shown in listing 3.4.

`ProtoStreamMarshaller` uses help of *ProtoStream*[25] library for encoding of entities. In order to provide better explanation of this quite complex topic, there are used code snippets from [15], which are modified to resemble `Book` example from *queries in the library mode* section above.

Listing 3.4: Necessary remote queries registrations

```
import
org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
...
ConfigurationBuilder clientBuilder =
    new ConfigurationBuilder();
clientBuilder.addServer().host("127.0.0.1")
    .port(11234)
    .marshaller(new ProtoStreamMarshaller());

    ProtoStreamMarshaller.getSerializationContext(
    remoteCacheManager).registerProtofile(
    "book-schema.protobin");
```

24. The process of creation `.protobin` binary descriptor: <https://developers.google.com/protocol-buffers/docs/techniques?hl=ro#self-description>.
25. ProtoStream sub-project: <https://github.com/infinispan/protostream>.

As a last step, it is important to register marshaller[26] for the `Book entity` (see listing 3.5).

Listing 3.5: Entity marshaller registrations

```
ProtoStreamMarshaller.getSerializationContext(
    remoteCacheManager).registerMarshaller(
    Book.class, new BookMarshaller());
```

For illustration, `UserMarshaller.java` as an example of entity marshaller class can be found in Infinispan's *Protostream* sub-project[27].

Finally, set up procedures are done and clients can start querying the cache over the Hot Rod protocol in a similar way as shown in listing 3.6.

Listing 3.6: Querying the remote cache

```
import org.infinispan.client.hotrod.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;
...

remoteCache.put(1, new Book("Infinispan", "Great book"));

QueryFactory qf = Search.getQueryFactory(remoteCache);

Query query = qf.from(Book.class)
    .having("title").eq("Infinispan")
    .toBuilder().build();

List list = query.list();
assertEquals(1, list.size());
assertEquals("Infinispan", list.get(0).getTitle());
assertEquals("Great book", list.get(0).getDescription());
```

26. Marshalling: <http://en.wikipedia.org/wiki/Marshalling_%28computer_science%29>.
27. UserMarshaller.java code example: <https://github.com/infinispan/protostream/blob/1.0.0.Alpha6/sample-domain-implementation/src/main/java/org/infinispan/protostream/sampledomain/marshallers/UserMarshaller.java>.

The whole process of *Protobuf* format adoption and necessary registrations is quite complex and users are forced to use *Google's Protocol Buffers* as an encoding format for their data in order to make remote queries work properly. As very robust and compact, this approach still does not allow users to directly use queries for documents in common and popular JSON format.

The main challenge of this thesis is to develop alternative, standalone Infinispan server, which will allow clients to store JSON documents into Infinispan caches and process queries over these values and instead of Hot Rod and DSL, using OData standards (for more information about Open Data Protocol, see next chapter 4).

4 Open Data Protocol

This chapter provides fundamental knowledge about Open Data protocol (OData) and tries to answer the question why should be this solution tied together with Infinispan.

OData frees stored information from current data silos and brings the data for variety of possible consumers [17]. To achieve that, OData utilize common web technologies: HTTP, REST, JSON and Atom Publishing Protocol[1]. Data can be used only when it is understood properly by clients and OData prepares it for uniform access logic which uses abstract data model (see figure 4.1).

OData is licensed under the Open Specification Promise[2] which ensures that any clients can freely communicate with applications using OData protocol.

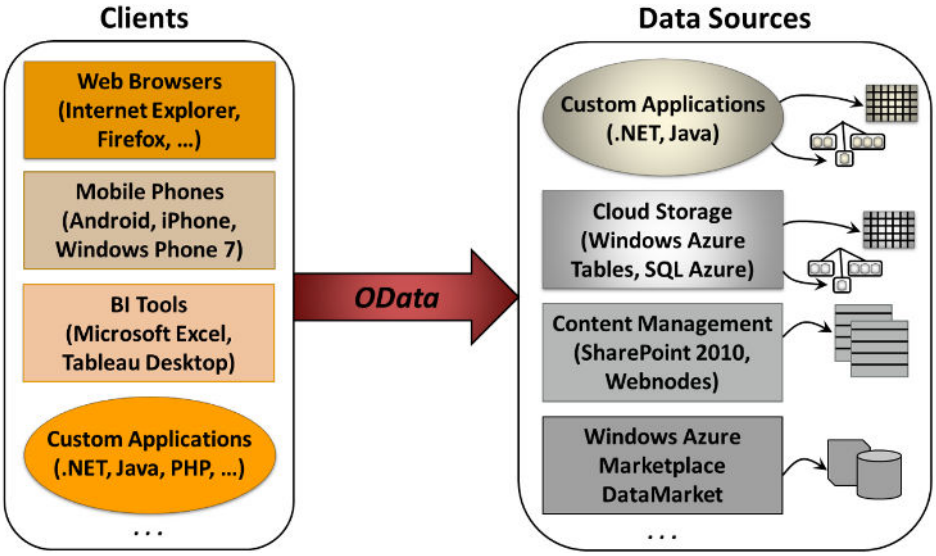

Figure 4.1: Using OData, diverse types of data sources become available to various clients, applications and devices [18].

1. AtomPub: <http://www.ietf.org/rfc/rfc4287.txt>.
2. Open Specification Promise: <http://www.microsoft.com/openspecifications/en/us/programs/osp/default.aspx>.

Two basic communication sides are specific for the OData: *producers* and *consumers*.

- *OData producers* are usually server-side services which provides data for consummation by consumers.
- *OData consumers* are clients which are able to connect to the OData service provided by producers and manipulate exposed data.

Although OData home page provides access to thorough OData protocol specification, this is not the best source of information for someone who need to get acquainted with the OData very quickly.

White paper written by David Chappell [18] provides an excellent overview and focuses on fundamental basics of the OData initiative. The paper is full of descriptive images and examples, which help readers to better understand more complex matter.

It is not easy to summarize stated information even more, however, this chapter tries to provide a shortcut of the most important facts with the help of David Chappell's paper.

Figure 4.2 illustrates four major components that all put together create OData environment architecture:

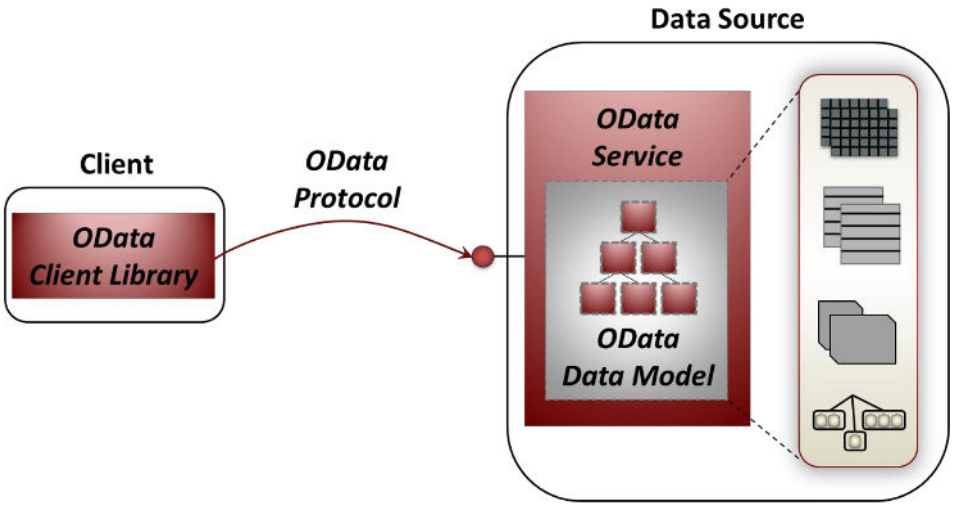

Figure 4.2: Four fundamental components of OData: data model, protocol, client libraries and service [18].

OData data model

Entity Data Model (EDM) is an OData abstract data model for describing logical structure independently of the way how data is stored physically. EDM is able to describe *entities* together with their mutual bidirectional *associations* (one-to-one or many-to-one) and it is up to a service provider how the internal data is mapped to the EDM.

OData protocol

REST, HTTP, OData EDM and OData query language (which is introduced later in section 4.2) are fundamental building blocks of the OData protocol.

Consumers communicate with producers using standard HTTP and follows REST conventions [18]: `GET` method for accessing the data, `PUT` method for complete updating already existing entities, `MERGE` method also for updating, but only for replacing selected properties, `POST` method is used for a creation of a new entity, and finally, `DELETE` method for an entity deletion.

When clients want to see or manipulate the data, they send an HTTP request to a specific URI. Producers provide a `metadata` document to let consumers learn about the service, its interface and EDM schema. Metadata document is accessible from a service's root URI (see example below), where clients can find EDM schema of a particular OData service, described in *conceptual schema definition language* (CSDL).

```
http://host:8887/ODataInfinispanEndpoint.svc/$metadata
```

EDM schema describes the data structure in an abstract way, but does not specify how the data is transferred over the network. For this reason, OData uses Atom or JSON format to serialize data into a form understandable for consumers. For better imagination, figure 4.3 depicts how the process of a translation from relational model to Atom works.

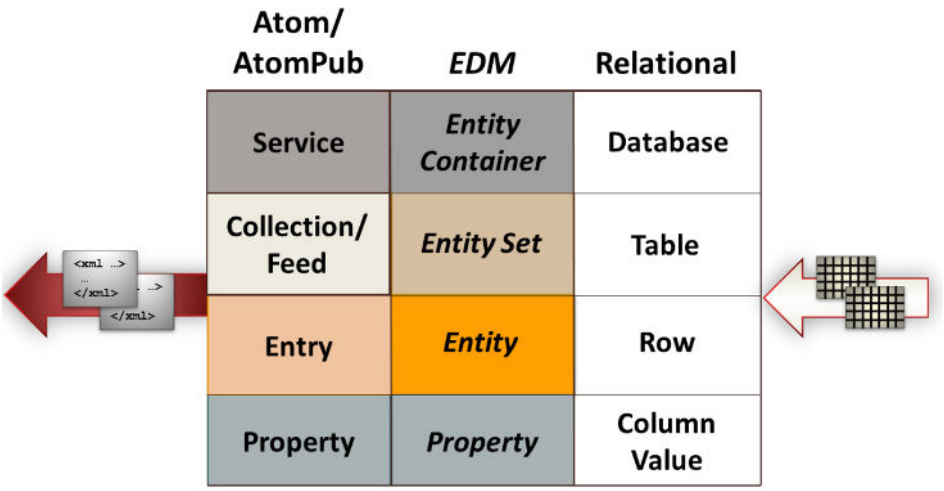

Figure 4.3: Data from RDBMS system is mapped to abstract EDM and then serialized into Atom (or JSON) which is understandable by consumers [18].

David Chappell also provides simple practical example for better understanding how the process of producer-consumer communication can look like (see figure 4.4).

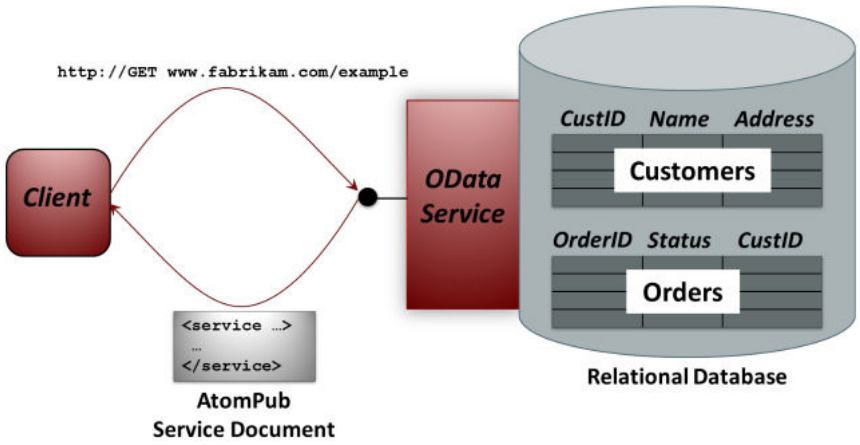

Figure 4.4: Example of OData service bound to underlying relational database [18].

OData service is bound to underlying relational database and exposes stored data for wide consumer usage. As it was mentioned earlier, consumers can ask for `metadata` document and in this case, they will obtain schema similar to:

```
<service ...>
...
    <collection href=„Customers">
        <atom:title>Customers</atom:title>
    </collection>
    <collection href=„Orders">
         <atom:title>Orders</atom:title>
    </collection>
...
</service>
```

Or if consumers want to deal with a JSON format:

```
{ „d" : {
    „EntitySets": [„Customers", „Orders"]
} }
```

Since now, consumers know, that OData service exposes two entity sets; *Customers* and *Orders*. It is possible to obtain the data from a particular database row by issuing HTTP `GET` request, using specific URI requesting retrieval of the order with ID 5630:

`http://www.fabrikam.com/example/Orders(5630)` returns:

```
{ „d" : {
    „results": {
        „OrderID": 5630,
        „Status": „Placed",
        „CustID": 8499734 }
} }
```

OData client libraries

There exist a pile of OData client libraries[3] in an OData ecosystem, and therefore, developers does not need to create client code from scratch and they can choose from already prepared solutions for: *JavaScript*, *Java*, *.NET framework*, *Silverlight*, *PHP*, *Ruby*, *Android*, *iOS*, or *Windows Phone 7*.

OData service

Special tools and frameworks does not provide support only for a client side, but most of them can be also used for a help with creation of a server side producers' code. Developers can utilize this features for starting the service, for data serialization into Atom or JSON format, and for parsing URIs containing advanced OData queries.

4.1 Why OData protocol?

This section discuss principal advantage which can OData bring into applications' architectural infrastructure.

In the matter of a client communication, databases and data silos offers usually diverse API's which need to be documented and developers sometimes have to put demanding care into adaptation of these interfaces. OData focuses its effort on a creation of an uniform access pattern for various sources of data, to unlock them and prepare for broad consummation.

To solve this task, OData uses the HTTP RESTful standard, an abstract OData Entity Data Model, and well known ATOM and JSON formats as a format for data interchange between clients and servers (see figure 4.5).

3. OData libraries: <`http://msopentech.com/odataorg/libraries/`>.

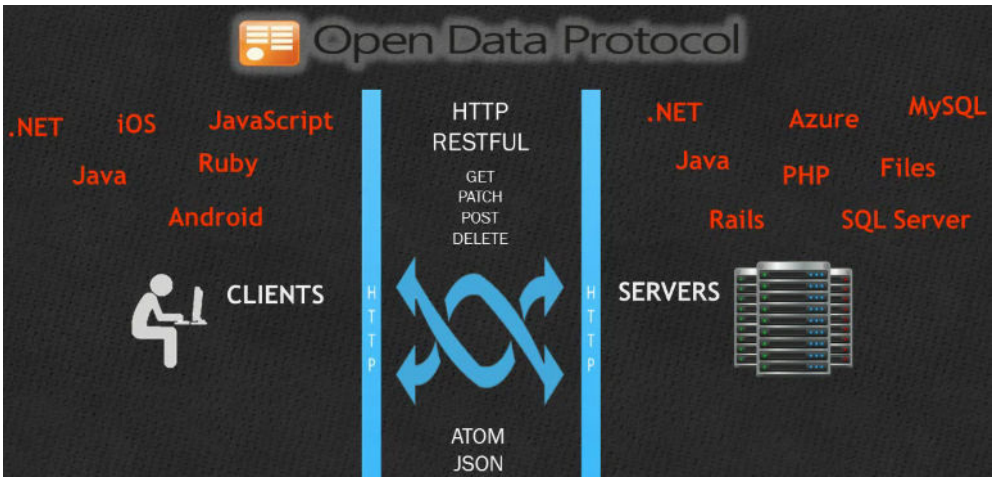

Figure 4.5: Illustration screenshot from OData introduction video [17].

4.2 OData query language

OData also implements advanced query language which allows clients to filter and order results of their `GET` requests [18]. OData queries are prefixed by „`$`“ sign inside of a request URI. Possible (not a complete list) system query options are:

`$top=n`: Returns the first `n` entities.

`$skip=n`: Skips the first `n` entities.

`$format=n`: Specifies data format for returning (JSON or Atom).

`$filter=<expression>`: Can be used for filtering entities which match an `expression`. An *expression* consists of *logical*, *arithmetic*, or *grouping built-in filter operators*[4].

`$orderby=<expression>`: Orders results, in dependence on one or more selected properties, in ascending or descending order.

`$select=<expression>`: Returns only a subset of entity properties.

`$orderby`, `$filter` and `$select` options depend on OData expressions. An example of a `$filter` option usage may look like [17]:

`http://services.odata.org/OData/OData.svc/Products?$filter=Category eq 'shoes' and Price lt 100.00` where `eq` operator stands for *equal* and `lt` operator stands for *less than*.

4. OData expressions: <http://www.odata.org/documentation/odata-v3-documentation/odata-core/#102311_Built-in_Filter_Operations>.

4.3 Actions, functions and service operations

From the thesis' point of view, this section introduces another important OData extension: *actions*, *functions* and *service operations*.

According to *OData V2* documentation [20]: *"OData services can expose Service Operations, which are simple, service-specific functions that accept input parameters and return entries or complex/primitive values."*

According to *OData V3* documentation, section dedicated to extensions [21]: *"Actions and functions extend the set of operations that can be performed on or with a service or resource."*

These three constructs could provide an ideal way of communication with an underlying Infinispan cache and they are different one from each other.

Actions are operations that can have side effects and they may be bound to its first parameter. On the other hand, functions can be bound to its first parameter as well, but, functions are not allowed to have any side effects. Service operations are distinguishable from actions and functions by obligation to specify also an `HTTPMethod` annotation attribute on the corresponding data service `FunctionImport` element. Service operations can have side effects and they does not need to be bound to any parameter.

4.4 OData and Infinispan motivation

Generally, NoSQL solutions lack overall portability as they are different one from each other and almost each database of this kind provide an unique interface for accessing stored data.

Infinispan is not an exception and its library mode usage is mainly focused on Java clients. However, Infinispan did a step further to solve this problem by introducing its server modules. A few implementations of an Infinispan Hot Rod client were developed to support a few different programming languages and there is also an Infinispan REST server which provides an HTTP access to key-value store capabilities. The problem is that users are not able to effectively use Infinispan for operating with JSON documents.

OData offers solution for this situation as it provides an abstract EDM for describing data model, and therefore, with new endpoint, all OData clients will be able to understand Infinispan OData service and communicate with caches, store JSON documents into the caches and manipulate the data using OData standard. This will make Infinispan more open to the OData ecosystem and hopefully will expand number of potential users. Any client will be able to interact with the Infinispan in uniform way.

Additionally, OData advanced query language perfectly fits Infinispan needs, because it can be translated into Apache Lucene query and indexed JSON documents can be queried to obtain requested results from an underlying cache. It opens querying functionality over the JSON values and makes from the Infinispan a hybrid NoSQL store implementation mixing the best from key-value approach and strong capabilities of document store for complex querying.

5 Infinispan OData server design

Infinispan OData server should work as a standalone server opening up the document store functionality to the vast amount of possible consumers (clients). JSON was decided to be used as an universal format for data contained in documents which will be stored into Infinispan caches.

JSON is lightweight, already widely used format and requested by Infinispan users. Additionally, the processing of this format is faster than the processing of XML (as shown in a comparison case study of JSON and XML data interchange formats [22]).

This chapter introduces Infinispan OData server requirements, survey of possible solutions and projects around OData which fits Java programming language, and depicts functionality of the whole architecture using suitable UML diagrams.

5.1 Requirements

This section provides a list of technical requirements for Infinispan OData server together with use case UML diagram.

General requirements:

– standalone HTTP server

– can be started as a Java process with a possibility to specify Xmx and Xms parameters for Java heap size

– serves as a communication gate for accessing embedded Infinispan document store

Interface requirements:

– follows OData V3 standard

– supports basic querying over JSON documents and its fields

– retains key-value store access approach also for JSON documents[1]

– supports OData filters and advanced OData query language[2]

1. When entry is requested by its key, exactly one JSON document is returned.
2. Using this approach, a collection of JSON documents can be returned in dependence on a filter query.

– supports CRUD operations: create, read, update and delete for JSON documents

Infinispan related requirements:

– can be started with specified path to Infinispan configuration file

– is able to create a cluster and utilize Infinispan data-grid capabilities (replication, distribution)

See figure 5.1 for UML use case diagram.

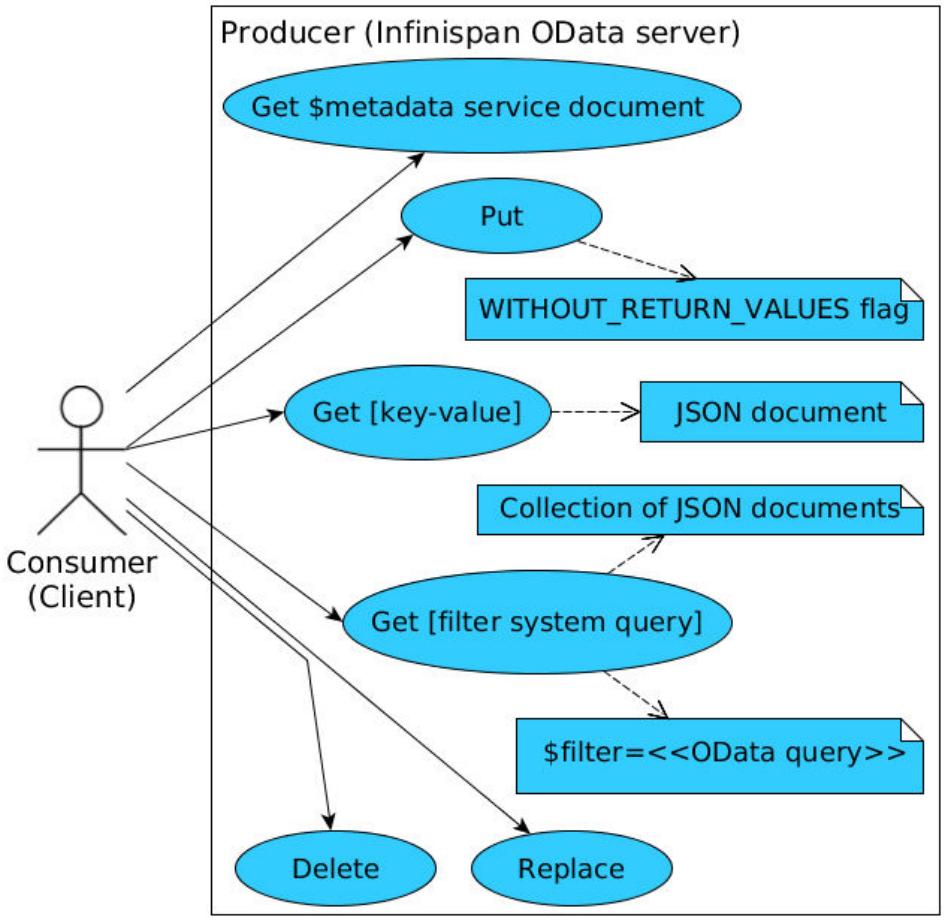

Figure 5.1: Use case UML diagram for Infinispan OData server.

5.2 Solution investigation

Before the process of designing, there were investigated possible frameworks, tools, examples and projects which may help with an Infinispan OData server implementation. The following list offers the results of that survey:

odata4j toolkit

odata4j – An OData toolkit for Java is a framework that implements the OData standards for both client-side (consumers) and server-side (producers) [19], which is licensed under the *Apache License 2.0*[3]. The Infinispan project primarily use Java and odata4j framework will help with adaptation to OData standards. For more information see the project home page at <`http://code.google.com/p/odata4j/`>.

OData Jersey server

Besides other features, odata4j framework also provides package `org.odata4j.jersey.producer.server` where is located a helpful class `ODataJerseyServer.java`. This class provides access to the OData server which is an implementation of *Jersey JAX-RS*[4] and Sun's HTTP server. OData Jersey server will be used as a server-side container for encapsulation of the embedded *Infinispan cache manager*[5], which will live inside the server and will utilize its Java virtual machine heap space.

InMemoryProducer class

In one of the odata4j examples, `InMemoryProducerExample` class instantiates `InMemoryProducer` for demonstration purposes where Java entities can be registered inside of an OData producer. This particular implementation of the `ODataProducer` interface provides an ideal baseline and template for the `InfinispanProducer` (more in subsec-

3. Apache License 2.0: <`http://www.apache.org/licenses/LICENSE-2.0`>.
4. Jersey: <`https://jersey.java.net/`>.
5. Cache manager is the Infinispan-specific entity which acts as a container for more Infinispan caches.

tion 6.4.1) which will be another implementation of the `ODataProducer` interface.

An EDM schema will be significantly different, but specific blocks of a code can be reused and instead of various POJOs, Infinispan caches will be registered as *entity sets*[6]. `InfinispanProducer` producer will be similar to an application using embedded Infinispan libraries where caches live inside of a server's Java virtual machine process.

After all necessary operations, a producer class is usually registered as a static instance to the OData Jersey server via `DefaultODataProducerProvider`.

OData actions, functions and service operations

A question how to expose supported cache operations for data manipulation emerged. Methods defined in `ODataProducer` interface determine specific return types, such as `EntityResponse`, `EntitiesResponse` or `void` return type in some cases. It would be better to have more space for Infinispan-specific needs which can appear now or later and usage of those methods is quite restrictive. On the other hand, `callFunction()` interface method return type is a `BaseResponse`, which seems to be a perfect interface to extend and implement for dealing with an alternative payload.

OData *functions and actions* mechanism is ideal for a mimicking of an Infinispan `org.infinispan.commons.api.BasicCache` API and possibly more extensions in the future.

Additionally, service operations may have a parameter defined and also work with `$filter` and other system query options. It will be possible to decide whether clients want to use key-value approach (`key` parameter definition) or document store capabilities (`$filter` option definition).

This alternative was investigated after some problems with the first server prototype which had significantly long response time and return type of a classic producer methods did not fit our needs for carrying over a payload for clients. Subsection 6.5 dedicates more attention to performance impact of this approach.

---

6. Entity Data Model (EDM) entity set.

Unfortunately, *functions and actions* are not supported in the latest released version (0.7) of the odata4j framework, which is still quite young. There exists an issue requesting that feature: <`http://code.google.com/p/odata4j/issues/detail?id=224` – *Implement OData actions and functions*. A big „*thank you*“ belongs to Samuel Vetsch who implemented this feature under 0.8.0-SNAPSHOT version – a code is located at: <`https://bitbucket.org/svetsch/odata4j-actions`>. It will be forked and modified for needs of the Infinispan.

FieldBridge from Hibernate search

As required, Infinispan OData server will support filtering of stored JSON documents, but Infinispan does not provide any mechanism for handling queries over JSON format. An annotation `Indexed` used for a POJO class, as was discussed in section 3.5, does not fit this use case because JSON documents can not be easily translated into the general POJO classes during an application's run-time. Users would like to store various types, or instances of various classes described by JSON schema. However, these classes are not known before.

This task can be resolved by using a `FieldBridge` mechanism from Hibernate search[7] which acts as a bridge between a Java property and a Lucene *Document*[8] [23].

Needed Java properties can be obtained using Jackson[9] libraries that are able to extract these properties from JSON string and prepare them for an addition into the Lucene Document. With the `FieldBridge`, it is possible to decide which fields and properties will be added.

Infinispan itself will take care about indices in dependence on an indexing configuration.

The question, how to index fields from JSON documents, is more elaborated in chapter *Implementation*, subsection 6.4.2, together with code listings for a better demonstration of an implemented solution.

---

7. Hibernate search: <`http://hibernate.org/search/`>.
8. Lucene Document class: <`http://lucene.apache.org/core/2_9_4/api/all/org/apache/lucene/document/Document.html`>.
9. Jackson, JSON processor: <`http://jackson.codehaus.org/`>.

Clustering issues

An embedded Infinispan cache manager instance will be run inside of a server and Infinispan-specific clustering mechanism should work without any problems, as it uses JGroups protocol for inter-node communication. Infinispan will take care about data distribution out-of-the-box.

5.3 EDM schema structure

Infinispan communication interface needs to be mapped to an interface of provided OData service, which brings a question how to apply an OData EDM schema structure here. EDM schema will expose a minimal subset of information needed by consumers to understand server's supported services.

OData Infinispan server will provide an access to a hybrid key-value and document store without any rigidly defined schema, where documents can be different one from each other. Therefore, there is no room for any closed EDM schema in the design. The schema will be as simple as possible to preserve overall simplicity of the key-value store approach together with an addition of querying possibility over JSON values.

Another idea how to deal with this problem was also considered; to change the EDM schema during the process of storing entries and try to generate new adjusted `$metadata` document in dependence on stored JSON objects. However, this solution would be resource exhausting, and thus, not suitable.

Because the fact that structure of JSON documents is not known before, there will be no description for `Properties` in EDM schema. This step will ensure, that consumers can freely store any JSON document object in any cache.

The most important elements are `EntitySet` and `FunctionImport`. One `EntitySet` for each Infinispan cache, with the same name and `FunctionImport` for every CRUD[10] operation and other possibly exposed operations.

10. CRUD: create, read, update, and delete.

A running Infinispan instance can be considered as a database. Then, an `EntityContainer` element represents `DefaultCacheManager`, `EntitySet` represents an element for mapping a started Infinispan cache, and stored JSON document is represented as an `Entity`.

Generally, the structure of storing the data is a matter of a database design. An Infinispan cache can be treated as a particular table and then, when it is needed to store different objects (for instance *Persons* and *Cars*), one *named cache* will be used only for storing *Person* JSON documents and the other one only for *Car* JSON documents.

This approach will automatically help to distinguish between different types of objects. Both *Persons* and *Cars* can have identical attribute, for instance *age*, and if cars and persons are mixed together in one cache, a query „*return all where age > 20*“ would return both *Cars* and *Persons*. However, if those objects are stored separately in different caches, client will firstly choose what cache to query, secondly, will build a query, and lastly, will obtain only the results of a specified type.

5.4 Basic component communication logic

After possible solution investigation is easier to imagine architecture of essential components and how they are connected together. Figure 5.2 depicts communication of basic components during `put` operation and figure 5.3 during `get` operation.

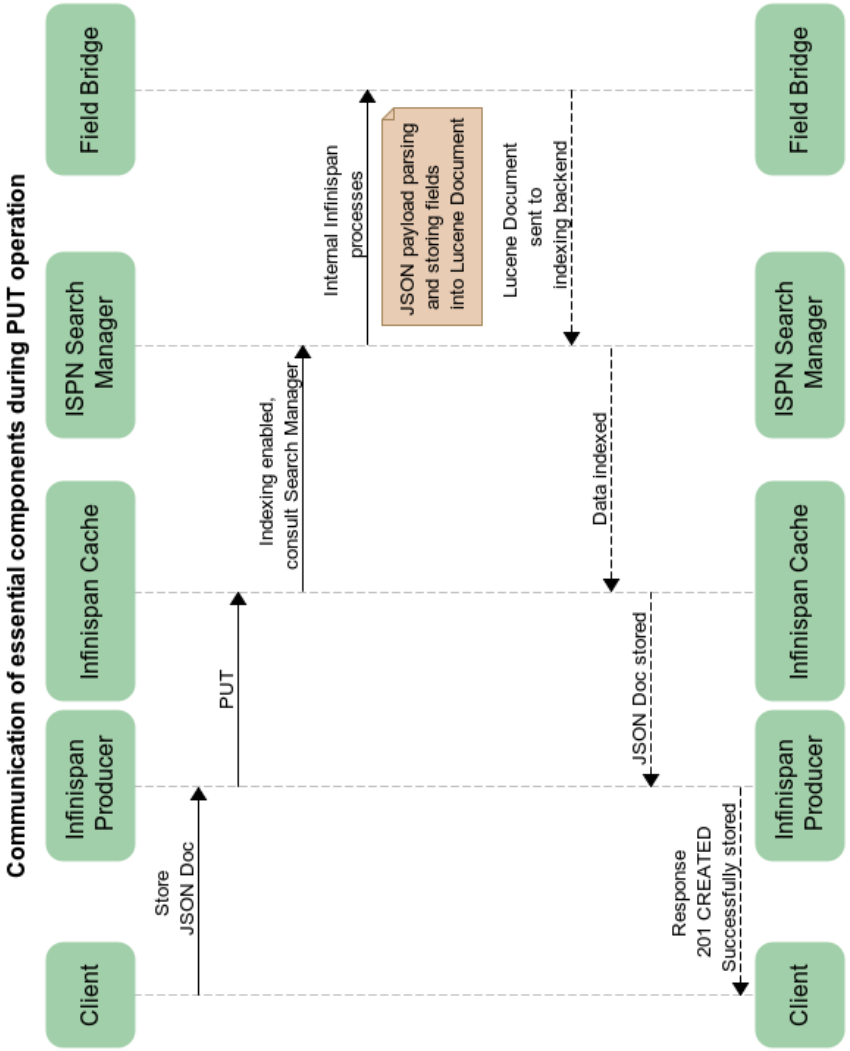

Figure 5.2: Communication of basic components during `put` operation.

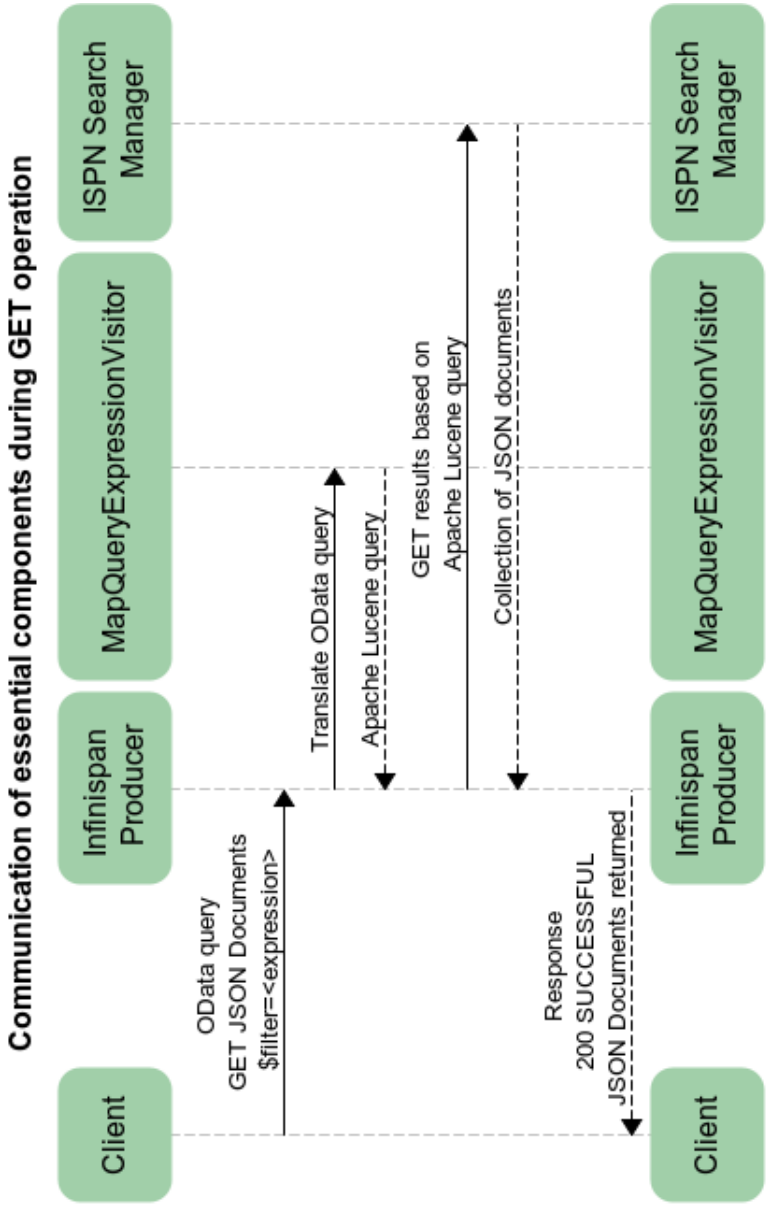

Figure 5.3: Communication of basic components during `get` operation.

6 Implementation

The thesis turns around three projects: *Infinispan OData server*, *Infinispan cakery* and *odata4j-actions*. Chapter *Implementation* describes the most important development aspects of the Infinispan OData server project and modifications in odata4j libraries. The Infinispan cakery project is more elaborated in next chapter (7.2) because it is bound up with a performance matter.

6.1 Source code and version control

The following list provides references to the crucial project code locations and repositories.

```
Infinispan OData server (Git):
   <https://github.com/tsykora/infinispan-odata-server/>

Infinispan cakery project (Git):
   <https://github.com/tsykora/infinispan-cakery/>

Modified odata4j, version 0.8.0-SNAPSHOT (Mercurial):
   <https://bitbucket.org/sykynx/odata4j-actions/>
```

6.2 Tools

It is hard to imagine the process of writing a code without an integrated development environment. IntelliJ IDEA serves perfectly for that purpose and also provides very useful embedded debugger, which is frequently used for faster identification of bugs. Another big advantage is that a plugin for JProfiler[1] can be installed into the IDEA, and thus, it is possible to run a server instance during development, try out necessary performance scenario, monitor methods and memory usage, and identify performance bottlenecks soon.

---

1. JProfiler: <http://www.ej-technologies.com/company/profile.html>.

Projects Infinispan OData server and Infinispan cakery are developed using JBossAS community code style. For the process of building and dependency management is used Apache Maven [24].

6.3 Building and running the server

`README.md` file located in the parent Infinispan OData server project directory provides easy-to-follow guide how to compile and start the server together with information about all necessary dependencies.

The second part of the `README.md` file contains a number of usage examples which demonstrates how to communicate with a server using `curl`[2] tool.

6.4 Implementation highlights

It is not an intention of *Implementation highlights* section to provide thorough description of implemented classes. The principal purpose is to rather introduce a model of the most important building blocks and present overall architecture, so further implementation details can be found easily in a source code itself.

6.4.1 (Infinispan) InMemoryProducer

A class `InfinispanProducer` can be considered as a heart of the whole Infinispan OData server. This class implements `ODataProducer` interface and all important code logic is done here. As mentioned earlier (5.2), this lightweight producer is registered via `DefaultODataProducerProvider` as a static instance to the OData Jersey server[3].

`InfinispanProducer` is based on in-memory producer example from odata4j, exactly `InMemoryProducer` class written mainly by John Spurlock, Tony Rozga and others. During the process of development it was decided that provided entity data model support is too heavy and complex for Infinispan OData server needs, and thus, significant part of

2. curl tool: <`http://curl.haxx.se/`>.
3. Jersey server of version 1.8 is used in modified odata4j 0.8.0-SNAPSHOT.

a code is not used. However, basic constructs for registering entity sets are reused for mapping of Infinispan caches.

Hosting Infinispan's default cache manager

During a construction of an `InfinispanProducer` instance is started embedded Infinispan `DefaultCacheManager` with a specific XML Infinispan configuration file. A path where that configuration file is located can be passed during a server start-up process.

`DefaultCacheManager` also starts all specified underlying Infinispan caches to be ready for an interaction.

EDM and Infinispan caches

Started Infinispan caches are registered as entity sets during a creation of new EDM schema. Then, `metadata` document can be accessed by clients to see list of configured caches.

Service operations – function imports

Together with an information about entity sets, a service `metadata` document also exposes a set of service operations that are used for interacting with the service. Reasons for this solution are already explained in design chapter (5.2).

More specifically, OData `FunctionImport` elements define open operations which can be used by OData consumers. These imports are specified during the process of generating of a new EDM schema (method `generateEdm(...)` and consequently `addFunctions(...)` method).

Every Infinispan cache (an `EntitySet` in EDM schema) has assigned four service operations: *put*, *get*, *replace*, and *remove* in order to support desired *CRUD* cache interface.

6.4.2 CachedValue, JsonValueWrapper and FieldBridge

JSON data coming from clients are not decomposed during the process of storing into an Infinispan cache. However, there is a necessity for some mechanism which allows to query those JSON documents according to their fields.

Infinispan `AdvancedCache` provides support for putting any object as a key, and the same applies for a value. Instances of a class `CachedValue` are stored as the values and keys are simply `Strings`, similar to Infinispan REST server module.

`CachedValue` object encapsulates a `JsonValueWrapper` object that wraps JSON string coming from a client side. This whole JSON document is stored into an Infinispan cache (into a document store) and indexed during the process of storing. Of course, Infinispan cache has to be properly configured with `<indexing enabled=„true" ...>`.

An usage of `@FieldBridge` annotation in `CachedValue` class is depicted in listing 6.1. Hibernate search determines what class to use for extended indexing according to annotation:
`@FieldBridge(impl = JsonValueWrapperFieldBridge.class)`.

Listing 6.1: CachedValue class encapsulating JsonValueWrapper

```
@Indexed
public class CachedValue implements Serializable {
    @Field(analyze = Analyze.YES, store = Store.NO, norms =
        Norms.NO, termVector = TermVector.NO)
    @FieldBridge(impl = JsonValueWrapperFieldBridge.class)
    JsonValueWrapper json;

    public CachedValue(String json) {
        this.json = new JsonValueWrapper(json);
    } ...
}
```

`JsonValueWrapperFieldBridge` performs demanding work of parsing stored JSON document („field by field" using Jackson libraries) and adding fields together with their values into *Lucene Document* (see listing 6.2). Infinispan configuration for indexing determines the place where Lucene Document is physically stored.

Above-mentioned logic is connected directly to the process of storing the JSON documents into the Infinispan caches. If server is started with an Infinispan configuration specifying disabled indexing, `@FieldBridge` and the whole process of indexing is ignored.

Listing 6.2: Main part of a JsonValueWrapperFieldBridge

```
@Override
public void set(String name, Object value, Document document,
    LuceneOptions luceneOptions) {
        ...
    // ObjectMapper -- org.codehause.jackon.map package
    ObjectMapper mapper = new ObjectMapper();

    Map<String, Object> entryAsMap =
    (Map<String, Object>) mapper.readValue(json, Object.class);

    for (String field : entryAsMap.keySet()) {
        // extracted field from JSON document is added into
            Lucene Document
        luceneOptions.addFieldToDocument(field,
            entryAsMap.get(field).toString(), document);
    } ...
}
```

6.4.3 MapQueryExpressionVisitor

OData defines its own advanced query language which needs to be translated into an Apache Lucene query in order to be used by Infinispan's `SearchManager` for querying indexed objects in the caches. This task is solved by `MapQueryExpressionVisitor` class that uses *Visitor* design pattern [25] and implements `ExpressionVisitor` interface from odata4j framework.

OData queries from requested URI are parsed and processed by odata4j framework into a `CommonExpression` implementations. Then, `InfinispanProducer` will obtain these instances as `QueryInfo.filter` objects during an OData function (service operation) call (see listing 6.3).

In `InfinispanProducer`, `QueryInfo.filter` object is passed to `MapQueryExpressionVisitor`'s `visit` method and the class parses provided OData query and creates Lucene query accordingly.

Listing 6.3: Processed QueryInfo object passed to the MapQueryExpressionVisitor instance

```
@Override
public BaseResponse callFunction(ODataContext context,
    EdmFunctionImport function, Map<String,
    OFunctionParameter> params, QueryInfo queryInfo) {
    ...
    mapQueryExpressionVisitor.visit(queryInfo.filter);
    ...
}
```

Listing 6.4 provides a closer look to the process of mapping common OData query to Lucene query, which will be later used to query Infinispan cache. `AndExpression` usually has two sides; the left side and the right side. Each of them is formed by next `Expression` and `visit()` method is recursively called to process the sub-expression to gradually build Apache Lucene query.

Listing 6.4: Translation of OData „AND“ expression to Apache Lucene query

```
@Override
public void visit(AndExpression expr) {
    BooleanQuery booleanQuery = new BooleanQuery();
    visit(expr.getLHS()); // left side, for instance "eq" query
    booleanQuery.add(this.tmpQuery, BooleanClause.Occur.MUST);
    visit(expr.getRHS()); // right side
    booleanQuery.add(this.tmpQuery, BooleanClause.Occur.MUST);

    // tmpQuery is returned for Infinispan SearchManager to
       query the cache for the results
    this.tmpQuery = booleanQuery;
}
```

Full list of supported system query options and filter operators is provided in appendix B.

6.5 Performance improvements in odata4j

During the process of development, a performance of Infinispan OData server is monitored by JProfiler tool with support of Infinispan cakery[4].

The first server prototype revealed that response time reaches a value of 40 milliseconds with using a common Apache HTTP client to send and get requests (see listing 6.5). Infinispan as a caching solution can not afford such a long response time and a workaround for this problem had to be found.

Listing 6.5: Common handling of a GET request with HttpClient

```
HttpClient httpClient = new DefaultHttpClient();
String get = serviceUri + "" + cacheName + "_get?$filter=" +
    query;
HttpGet httpGet = new HttpGet(get);
httpGet.setHeader("Accept", "application/json;
    charset=UTF-8");
HttpResponse httpGetResponse = httpClient.execute(httpGet);
EntityUtils.consume(httpGetResponse.getEntity());
```

Method `EntityUtils.consume()` (`org.apache.http.util` package) consumes obtained HTTP response and closes a content stream. All underlying system resources associated with the stream are released. The problem is that this operation takes more than 40 milliseconds using prepared odata4j specific `SimpleResponse` (or any other).

`FunctionResource` class (`org.odata4j.producer.resources`) processes responses from producer's `callFunction()` method. Usually, for instance in case of `SimpleResponse`, this class takes care also about serialization of the payload into standardized JSON response. The process of standardization is driven by OData format writers where was found a bottleneck.

4. Infinispan cakery is a tool developed, in scope of this thesis, for benchmarking of all Infinispan servers; and is introduced in section 7.2.

Firstly, the process of parsing a producer's response and following serialization took time and secondly, most importantly, these writers do not call `flush()` method, which causes problems. Infinispan-specific modifications in odata4j framework (version 0.8.0-SNAPSHOT) solves this problem.

It was needed to develop a new implementation of `BaseResponse` interface, `InfinispanResponse` in this case, to make the whole process more flexible and to allow to pass a payload from `InfinipanProducer` directly to the `FunctionResource` handling class. Since the whole JSON objects are stored inside of underlying Infinispan caches, there is no need for additional serialization of response there and the whole object can be returned as a response.

Then, it is possible to explicitly set up a `StreamingOutput`, write a JSON payload into it, flush it and close it, as nothing else is expected to be sent, and this response is immediately returned to the client. Apache HTTP client is able to consume this kind of response in 2-3 milliseconds in average; in dependence on a server load and testing environment. Listing 6.6 depicts the main part of an implemented workaround.

6.6 Mapping of cache operations

This section provides information about communication with underlying Infinispan key-value and document store from common OData consumer's point of view. Table 6.1 summarizes how CRUD operations of an Infinispan cache are mapped to the exposed OData service operations.

Infinispan cache operation	**OData service operation appendix**	**HTTP method**
put	_put	POST
get	_get	GET
replace	_replace	PUT
remove	_remove	DELETE

Table 6.1: Mapping of an Infinispan cache CRUD API to the OData service operations with specified HTTP methods.

Listing 6.6: Flushed and closed StreamingOutput on the server side

```
// response coming from InfinispanProducer
final InfinispanResponse ir = (InfinispanResponse) response;

StreamingOutput stream = new StreamingOutput() {
    @Override
    public void write(OutputStream os) throws IOException,
        WebApplicationException {
        Writer writer = new BufferedWriter(new
            OutputStreamWriter(os));
        // JSON document obtained from a cache
        writer.write(ir.getValue());
        writer.flush();
        writer.close();
    }
};

return Response.ok(stream, "application/json;charset=UTF-8")
    .status(ir.getStatus()).build();
```

Unfortunately, it is not possible to define two service operations with the same name and different only in assigned HTTP method. That is the main reason for using appendices for every service operation (`FunctionImport`).

For instance, service operations for `default` cache are called: `default_put`, `default_get`, `default_remove`, and `default_replace`.

It is also important to mention here, that a stored JSON document can be accessed by two approaches: *key-value store approach* (using service operations with a parameter `key`) or *document store approach* (using service operations with `$filter` specified).

The process of accessing an underlying Infinispan cache for the data is driven by internal logic of `InfinispanProducer class` according to input parameters for *get* requests.

Practical usage examples can be found in `README.md` file located in the main Infinispan OData serve project directory.

6.7 Functional test suite

JUnit[5] functional test suite is also a part of the Infinispan OData server project. Apart from tests for basic endpoint functionality (CRUD operations), tests are also ensuring that OData standards are followed.

Especially that: HTTP response codes are properly returned together with properly set HTTP headers where needed; OData queries are properly mapped to Apache Lucene queries and the server is returning the right results according to system query options and various operators; an underlying Infinispan is working and communicating properly; etc.

The test suite is located in `proj_home/src/test/java` project folder.

Functionality of the Infinispan OData server is tested via Apache HTTP client. Generally, these tests are from client point of view and ensures that clients can obtain requested JSON documents according to their queries and that OData service interface is working properly.

5. JUnit: <`http://junit.org/`>.

7 Performance testing

Performance testing chapter firstly introduces tools, which are used during the process of benchmarking, secondly, shortly describes automation and testing environment, then, elaborates performance testing plan and lastly, provides review of measured performance statistics across different testing scenarios.

7.1 PerfCake tool

PerfCake is a lightweight testing framework developed by Pavel Macík and Martin Večeřa from Red Hat's JBoss Middleware division [26]. This tool has been chosen as a framework for benchmarking of Infinispan servers because of a number of reasons.

PerfCake is very easy-to-use and can be integrated into a project simply via maven dependency management. Then, it is possible to create Java application which acts as a load generator and can be build and run by Maven. This fact has positive impact on testing automation where scripts takes care about the process of building and running load generating application.

The run of PerfCake is driven by a `scenario` XML file where are defined `generators`, `senders`, `reporters` and other configuration elements. A full list of elements is described in project documentation [26].

Already prepared `senders`[1] can be used for testing. In cases where provided senders are not suitable, it is possible to create own sender; using Java programming language. Additionally, PerfCake fits thesis' needs as it is licensed under Apache License 2.0. Finally, a possibility of close contact with developers is also an advantage.

7.2 Infinispan cakery

Infinispan cakery is a project for stress testing and benchmarking Infinispan servers, run by Maven and using the latest released PerfCake libraries (version 1.0). Additionally, after small modifications, it can

1. PerfCake sender is an element which is responsible for sending messages to the tested application.

be simply used for measuring a performance of Infinispan embedded code scenarios as well (not exclusively server modules). Source code is publicly available at GitHub[2] and can be reused and modified for any further performance testing of the Infinispan.

This project was found in order to be frequently used together with JProfiler during the time of Infinispan OData server development in order to reveal performance bottlenecks and to verify effectiveness of implemented optimizations. And most importantly, Infinispan cakery was used for final benchmarking and stress testing of Infinispan servers, as well as new Infinispan OData server.

PerfCake default (HTTP) `Senders` are suitable only for REST and OData server module but for the complete process of benchmarking it is necessary to use Infinispan-specific Hot Rod and Memcached client[3] as well.

Therefore, Infinispan cakery extends PerfCake senders by a new set of senders for Hot Rod, Memcached, REST and OData server to obtain better control over the code and test logic (generation of JSON documents, reporting of Infinispan-specific problems). These senders extend `AbstractSender` class and a block of code which is desired to be measured needs to be located inside `doSend()` method.

Then, a particular sender is specified in scenario XML file using: `<sender class="IspnCakeryHotRodSender">` together with other necessary elements: `generator` and `reporters`.

A list of reporters used by Infinispan cakery can be found bellow:

- `WarmUpReporter` – is used for warming a tested system up, stabilization of the process of garbage collection and filtering out initial non-stable metrics statistics.

- `ResponseTimeReporter` – measures and reports average response time value during the whole run[4].

2. Infinispan cakery: <https://github.com/tsykora/infinispan-cakery>.
3. Memcached client: <https://github.com/infinispan/infinispan/blob/master/server/integration/testsuite/src/test/java/org/infinispan/server/test/client/memcached/MemcachedClient.java>.
4. Hint: If it is desired to see a response time of every single response, this configuration can be used: `<period type="iteration" value="1"/>`.

- `WindowResponseTimeReporter` – functionality of this reporter is the same as functionality of common `ResponseTimeReporter`, but average response time is measured only for specified number (a sliding window) of latest responses[5].
- `AverageThroughputReporter` – measures and reports average number of operations per second that tested system is able to handle. Average value is calculated from the whole run results.
- `MemoryUsageReporter` – PerfCake also provides an agent for monitoring memory usage, which can be run on target system in order to gather memory usage statistics.

All reporters are set up to report metrics using a console and also store them into a `.csv` file; with a period of one second.

For instance, usual `ResponseTimeReporter` configuration inside of a scenario XML file:

```
<reporter class="ResponseTimeReporter">
    <destination class="ConsoleDestination">
        <period type="time" value="1000"/>
    </destination>
    <destination class="CSVDestination">
        <period type="time" value="1000"/>
        <property name="path" value="response-time.csv"/>
    </destination>
</reporter>
```

How to run Infinispan cakery and other technical details are described in `README.md` file located in the main project folder.

7.3 Automation and testing environment

Automated continuous integration (CI) Jenkins[6] system is used for making testing easier and repeatable. Jenkins jobs are set up by us-

5. Infinispan cakery uses 500 as a value of measured window by WindowResponseTimeReporter.
6. Jenkins: <http://jenkins-ci.org/>.

ing automated scripts with the possibility of gathering and archiving measured performance results.

There is also configured a *killer job* which cleans machines in case of any failure in CI. It looks for identifiers of processes started by performance benchmarking job and kills those processes, so machines are clean and ready for other tests in laboratory.

Tests are run in a JBoss Data Grid Quality Engineering team performance laboratory, where is located 8 machines – 4 machines are used for load generation with running an Infinispan cakery instance and 4 machines for a creation of an Infinispan 4-node cluster.

A schema of the process of load generation is depicted in figure 7.1. Instance of running Infinispan cakery acts as a load generator, utilizes performance of a machine and loads one of four servers connected into Infinispan 4-node cluster. Performance tests against Infinispan Hot Rod, Memcached and REST servers are run under the same scenario.

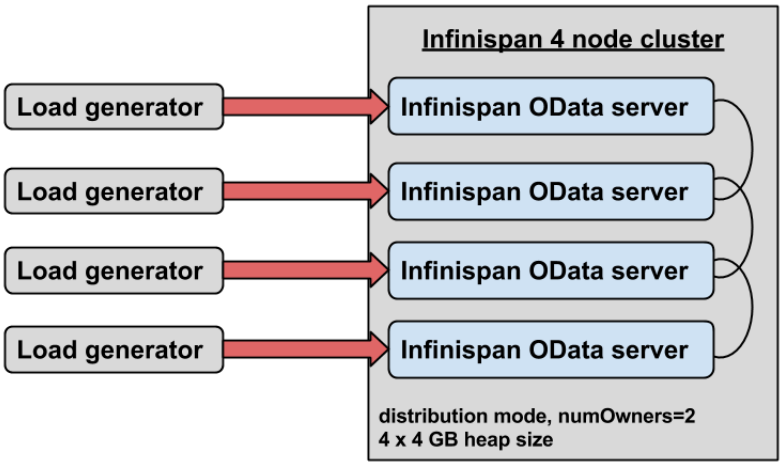

Figure 7.1: Utilization of 8 machines in Red Hat's JDG QE performance laboratory: Running Infinispan cakery instance acts as a load generator. The same schema is used for Infinispan Hot Rod, Memcached and REST server.

Standalone stations are used instead of any virtual instances, because data-grid is usually run this way in production environment. Eight machines in total, each standalone machine with this hardware and software configuration:

```
OS name: linux, version: 2.6.32-431.1.2.el6.x86_64
OS arch: amd64, family: unix

Java: OpenJDK 1.7.0.45.x86_64

CPU: 8 cores, 2000.335 MHz, AMD Opteron 6128
CPU L2 cache size: 512 KB
RAM: 64 GB
Network: 1000 Mbps full duplex
```

7.3.1 Libraries and versions

Table 7.1 provides a list of components which are connected to following performance testing results.

Tool / Library	**Version**
Infinispan OData server	1.0-SNAPSHOT
Infinispan cakery	1.0-SNAPSHOT
Infinispan core	6.0.0.Final
Infinispan query	6.0.0.Final
Infinispan Hot Rod client	6.0.0.Final
Infinispan servers	6.0.0.Final
Apache HTTP client	4.3.1
PerfCake	1.0
odata4j	0.8.0-SNAPSHOT

Table 7.1: Versions of components connected to performance testing of Infinispan servers.

7.4 Performance testing plan

This section firstly provides information about a testing plan that is used for performance benchmarking of Infinispan server modules and Infinispan OData server. Secondly, results of all four servers basic comparison are presented, and then, followed by results of advanced com-

parison of Infinispan REST and Infinispan OData servers. Lastly, two different approaches for accessing stored JSON documents in Infinispan OData server are compared (key-value and query approach) from performance point of view.

7.4.1 General testing plan division

It is important to choose an approach where are not compared apples and oranges, and therefore, the process of performance testing is divided into three main categories:

- **Key-value access: all servers, basic** – This scenario compares all four Infinispan servers to find out differences between three various client approaches. These access approaches are distinct one from each other. Clients communicate with Infinispan Hot Rod server using a Hot Rod client; with Infinispan Memcached server using a Memcached client; and for a communication with both Infinispan REST and OData server is used common Apache HTTP client. Scenario mainly aims at speed of serves.

- **Key-value access: OData vs REST server, advanced** – Infinispan cakery uses Infinispan Hot Rod and Memcached client for communication with these two respective server modules, however, for REST and OData, it uses common HTTP client. Therefore, this scenario applies only to benchmarking REST server against its true competitor – OData server. Infinispan REST server is implemented like one lightweight class written in Scala language. The principal intention of this scenario is to find out how both REST and OData server will perform during increasing load.

- **OData server: key-value access vs OData query access** – Infinispan OData server provides two approaches for accessing stored JSON documents: a key-value approach and OData query language approach. This scenario aims at revealing performance differences between direct key-value access and more complex OData query access.

7.4.2 Consistent benchmark settings

Consistent settings which are the same for all runs and scenarios and also for all server modules are listed below. All Infinispan servers are started with these settings during performance testing:

- **JVM heap size** – Servers are started with `Xms=4096m` and `Xmx=4096m` options. Options are intentionally set to the same values as that is recommended by JBoss Data Grid QE team (JDG QE) for smooth running of an Infinispan cluster. These settings help to avoid problems in cases when actual size of a heap needs to be re-set during heavy load. It is also recommended by JDG QE that size of stored entries should not exceed 50 % of specified heap size.

- **Infinispan related settings** – A cluster is created by four servers where Infinispan caches are run in distribution mode with `numOwners=2` settings, which causes that each stored entry is replicated to one other Infinispan node. `TCP` is used as a transport protocol as it fits better small clusters (under 100 nodes in size) running in distribution mode (see section 1.4.2 in [12]). Property `-Djava.net.preferIPv4Stack=true` is set up together with `-Djgroups.tcp.addres=$inet_addr` for smooth node discovery.

- **LargePages** – For better performance of high throughput and memory-intensive applications is recommended to enable *Java support for large memory pages* [27]. This is achieved by enabling JVM option: `-XX:+UseLargePages`.

- **No delay between requests** – PerfCake load threads send requests at maximum speed. Red Hat's JBoss Division QE teams are usually using 100 milliseconds delay between requests to virtually simulate real load of application usage. This value has been chosen as a trade-off between real request delay and number of real clients communicating with an application. In the benchmarking of Infinispan server modules, such a setting has no real effect, because the intention is not try to simulate real load, but compare server modules with each other under heavy load.

- **Filling of caches, entry size** – At the beginning of every run, 30000 entries is stored into the cache. Infinispan cakery generates entries approximately of `20,3 kB` in `value` size. Then, server is loaded with huge number of `get` requests.

  Special scenarios with put-get ratio (1:1, and 1:9 for heavy-read systems) were also considered but these are out of the thesis' scope, because the intention, again, is not try to simulate real system load but rather benchmark server modules.

  However, when this kind of testing will be requested in the future, it will not be a problem to mimic special put-get ratios by extending Infinispan cakery senders and scenarios.

7.4.3 Smoke testing

A quick smoke test is run first to see whether testing environment is set up properly, all logs and metrics are flawlessly gathered, memory usage is monitored, and Infinispan cluster is successfully formed.

Benchmark automation in Jenkins tremendously helps with running performance tests with different settings for each run. That is enough to change a few test properties (duration of a run, number of load threads, target Infinispan server, etc.) in a configuration of Jenkins job and simply rerun the test. To make two changes in a configuration is a faster approach than running a full long test which could fail later.

7.5 Comparison of all four Infinispan servers

OData server is started with settings where indexing is disabled in order to have the same setup as other three Infinispan server modules. Key-value access approach and consistent settings described in testing plan are used for all servers.

Performance test parameters:

```
number of load threads: 5 (5 per node, 20 per cluster)
number of entries: .... 30000
warm-up period: ....... 60 seconds
main load period: ..... 10 minutes
request delay: ........ 0 milliseconds (full load)
```

Table 7.2 depicts performance statistics of all four Infinispan servers reached during a run with above-mentioned scenario parameters. Values apply for the whole Infinispan 4-node cluster.

Server type	**Average response time**	**Average throughput**	**Max memory usage**
Hot Rod	0.07 ms	285714 ops/s	1.984 GB
Memcached	1.70 ms	11765 ops/s	2.338 GB
REST	3.42 ms	5848 ops/s	2.057 GB
OData	2.59 ms	7722 ops/s	2.817 GB

Table 7.2: Performance results of all four Infinispan servers.

Infinispan Hot Rod server which uses Infinispan-specific Hot Rod binary protocol clearly provides the best performance capabilities. Memcached, OData a REST protocol response times are higher; all three belong into a category of text protocols. Chapter 8 provides more room for a discussion of results.

The main purpose of this basic scenario is to present the fact that performance of HTTP-based access differs from Hot Rod and Memcached. The next scenario is aimed at a comparison of two true competitors – REST and OData.

As will be elaborated in next section, Infinispan REST server exceeds average server response time threshold (100 ms) under the load of 400 load threads. For illustration, table C.1 and chart in figure C.1 shows statistics of Infinispan Hot Rod and REST servers exposed to the same load. Fine-tuned Java binary Hot Rod client provides not only much better operational throughput but also memory utilization.

7.6 OData and REST server comparison

In this scenario, OData server is also started with disabled indexing as it does not make sense to compare different Infinispan cache settings because Infinispan REST server does not support Infinispan queries. Therefore, data are accessed using key-value approach in both cases.

Infinispan cakery uses Apache HTTP client version 4.3.1 for connecting to Infinispan REST and OData server.

The testing strategy is as follows – settings for number of entries and test duration remains the same, but number of load threads, loading the whole cluster, is increasing. Thus, servers are put under more and more load in every successive run. The threshold for average response time is set to 100 milliseconds.

Performance test parameters:

```
number of load threads: varies (20, 40, 60,...per cluster)
number of entries: .... 30000
warm-up period: ....... 60 seconds
main load period: ..... 5 minutes
request delay: ........ 0 milliseconds (full load)
```

Infinispan OData server performance results are depicted in table 7.3. Every table row represents a particular run of Jenkins performance testing job with specified number of loading threads and other parameters. Gathered metrics: average server response time, average throughput, and maximal used memory during a test run.

Load threads	**Average response time**	**Average throughput**	**Max memory usage**
20	2.59 ms	7722 ops/s	2.817 GB
40	36.64 ms	1091 ops/s	2.857 GB
60	63.47 ms	945 ops/s	2.958 GB
80	141.87 ms	564 ops/s	3.020 GB

Table 7.3: Infinispan OData server performance statistics.

Performance results of Infinispan REST server are summarized in table 7.4.

Load threads	Average response time	Average throughput	Max memory usage
20	3.42 ms	5848 ops/s	2.057 GB
40	3.47 ms	11527 ops/s	2.052 GB
60	3.69 ms	16260 ops/s	2.071 GB
80	4.14 ms	19324 ops/s	2.088 GB
120	6.32 ms	18987 ops/s	2.507 GB
200	25.60 ms	7813 ops/s	3.770 GB
400	101.90 ms	3925 ops/s	3.770 GB

Table 7.4: Infinispan REST server performance statistics.

Infinispan REST server clearly outperforms Infinispan OData server during exposition to growing number of load threads increasing in every consecutive run. During the run with 80 load threads, a cluster of four Infinispan OData servers exceeded set threshold of 100 milliseconds for average server response time.

Infinispan REST server is implemented as one Scala class and fine-tuned for key-value access. This approach takes advantage over OData Jersey server (version 1.8) used in Infinispan OData server's core. OData standardization takes its price because there is used a pile of classes for checking and parsing OData queries possibly coming within client requests.

However, Infinispan OData server provides slightly better response time in case of smaller load (20 load threads for the cluster), and additionally, is ready to understand internal structure of JSON documents and supports OData query language.

Charts for average server response time and average server throughput are depicted in figure 7.2 and figure 7.3.

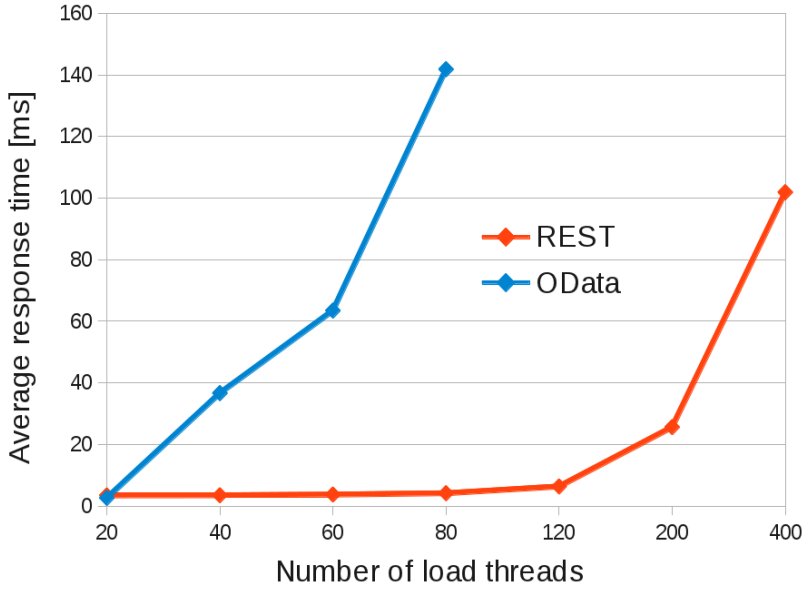

Figure 7.2: Infinispan OData and REST server average response time chart.

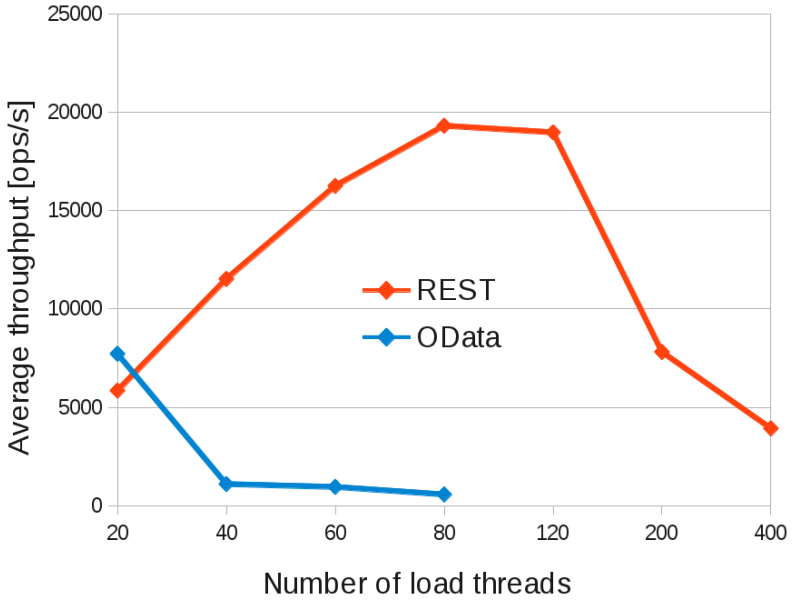

Figure 7.3: Infinispan OData and REST server average operational throughput chart.

7.7 OData server: Key-value and query access comparison

Finally, this section provides basic performance comparison of two possible access approaches implemented in the Infinispan OData server; to find out difference between raw key-value access to the data and access using OData query language.

Example of key-value access[7]:
`host:8887/OIE.svc/odataCache_get?key='person1'`
– selects one cache entry in dependence on a `key`.

OData query access:
`host:8887/OIE.svc/odataCache_get?$filter=id eq 'person1'`
– selects one *Person* document according to internal JSON field `id`.

Before the test run, Infinispan set up for queries needs to be tuned up to be able to provide even better performance and be comparable with simple key-value access approach. Infinispan configuration file `indexing-perf.xml` is used because default settings are not optimal. This is a configuration suggested and used by Sanne Grinovero and Adrian Nistor for performance testing of embedded queries [28]. Slightly modified `indexing-perf.xml` file is included in Infinispan OData server.

Infinispan OData server is started for both scenarios (key-value and OData query access approach) with `indexing-perf.xml` configuration and following JVM options and system properties mentioned among performance test parameters:

```
number of load threads: 20 (per cluster)
number of entries: .... 2000
warm-up period: ....... 60 seconds
main load period: ..... 5 minutes
request delay: ........ 0 milliseconds (full load)
```

7. OIE: ODataInfinispanEndpoint

```
entry size ............ small JSON, 5 indexed fields

Infinispan configuration file: indexing-perf.xml

Additional options
(recommended for better indexing performance):

-XX:+UseParallelGC -XX:MaxPermSize=128m
-Dorg.jboss.resolver.warning=true
-Dsun.rmi.dgc.client.gcInterval=3600000
-Dsun.rmi.dgc.server.gcInterval=3600000
-Dcom.arjuna.ats.arjuna.coordinator
     .CoordinatorEnvironmentBean.asyncPrepare=true
-Djava.awt.headless=true
-Dinfinispan.unsafe.allow_jdk8_chm=true

Logging disabled, jmx-statistics disabled
```

Access type	**Average response time**	**Average throughput**	**Max memory usage**
Key-value	3.14 ms	5675 ops/s	1.410 GB
OData query	24.89 ms	815 ops/s	1.749 GB

Table 7.5: Performance of key-value access approach and OData query access approach used in Infinispan OData server.

Gathered statistics are summarized in table 7.5 which depicts that basic key-value approach outperforms access for data using OData query language.

Internal logic of `InfinispanProducer` class chooses access approach in dependence on incoming parameters from OData service function call.

In the case of key-value approach, Infinispan cache is accessed directly according to `key` parameter. More precisely, `AdvancedCache` in-

stance is asked to return a `value` according to the `key`, and thus, Infinispan `SearchManager` is not consulted for query results.

There is a number of factors which can potentially have impact on performance in case of OData query access[8]: complex OData `$filter` query needs to be parsed from URI; `queryInfo.filer` instance is created and passed to `InfinispanProducer`; translation from OData query language to Apache Lucene query takes place; `SearchManager` is consulted for obtaining results from a cache in dependence on built Apache Lucene query.

Further investigation of above-mentioned factors is out of scope of this thesis and we suggest to provide thorough analysis as a future work.

8. In this scenario: using `get` operation and service filter option to filter exactly one `value`.

8 Summary of results and discussion

This chapter provides summary of measured performance results during the Infinispan servers benchmarking process. Three scenarios were used for various suitable types of servers comparison.

The first scenario focused on basic differences in speed of all four Infinispan servers – Hot Rod, Memcached, REST and OData server. The principal purpose of the second scenario was to compare REST and a new OData server under stress conditions where a number of load threads is increased during every consecutive test run. The third scenario compared two possible access approaches which Infinispan OData server provides as a hybrid NoSQL store – key-value and OData query based approach.

To start results summary, Memcached Infinispan server is discussed as the first. The main reasons for a usage of Memcached protocol in Infinispan were the fact that Memcached protocol is commonly used and it also provides decent speed capabilities. Infinispan supports text version of Memcached protocol. However, Memcached clients are not aware about cluster topology and their server list needs to be updated when a new node is joined[1] to the cluster.

Infinispan Hot Rod server was implemented in order to surpass above-mentioned constraints. Additionally, what is important from performance point of view, Infinispan Hot Rod clients are aware about cluster topology and Infinispan-specific binary Hot Rod protocol supports smart routing. Therefore, clients exactly know which node to ask for a data and that results into less unnecessary inter-node communication with direct positive impact on a performance.

As was shown in the first testing scenario, Hot Rod and Memcached servers are not true competitors for HTTP based OData and REST text protocols; Infinispan-specific binary Hot Rod client and simple socket-based Memcached client perform better as data is carried over the network more effectively. Therefore, second scenario was dedicated mainly for comparison of Infinispan REST and OData server performance.

REST server, implemented as a simple and lightweight Scala class, outperformed OData server in key-value scenario with respect to number of maximal load threads. However, Infinispan OData server, based

1. Or when one of nodes crashed.

on OData Jersey server, needs to follow OData standards in order to provide functionality for querying over JSON documents. Infinispan OData server was able to provide faster response time than Infinispan REST server under low load conditions.

Finally, the third testing scenario revealed performance differences between key-value and OData query based access approach to Infinispan OData server. Simple key-value approach performs better and we suggest further performance analysis of this scenario as a possible future work.

At the end of this section is provided short summary of all four Infinispan servers.

Hot Rod: very fast Infinispan-specific binary protocol, topology aware, smart routing, possibility of both key-value and document store. Remote querying functionality using DSL and Google's ProtocolBuffers for entity encoding.

Memcached: fast and commonly used protocol, text version only is used in Infinispan, accessible through simple sockets, plenty of clients, only key-value access.

REST: HTTP based access, only key-value access approach, its implementation is very lightweight.

OData: HTTP based access, follows OData standards, possibility of remote queries over JSON documents, both key-value and query based access.

9 Conclusion

In the context of this thesis has been developed a new standalone Infinispan server which is able to understand internal field structure of JSON documents and provides possibility of querying over these stored values. This was achieved by using FieldBridge construct from Hibernate search project.

Additionally, this server provides Open Data Protocol communication interface and exposes service operations to various clients (consumers). A new Infinispan OData server opens functionality of underlying Infinispan caches to wider audience as its core is based on odata4j framework.

Another thesis' goal was to benchmark this new server against already existing Infinispan Hot Rod, Memcached and REST servers. In order to fulfill this requirement, the PerfCake testing framework is used in a new project – Infinispan cakery which acts as a load generator and benchmarking tool for Infinispan servers.

Performance laboratory of JBoss Data Grid quality engineering team was used for the purpose of server benchmarking. Performance was measured using 8 server machines – 4 load generators and 4 connected in 4-node Infinispan cluster. The whole process of performance testing was automatized in Red Hat's Jenkins system.

All servers were compared using basic scenario; where Hot Rod protocol outperforms the others. OData and REST servers, as a true competitors, were compared using advanced stress test scenario; where OData server responds faster under small load and REST server under heavy load.

Performance of Infinispan OData server was monitored during development by Infinispan cakery and JProfiler tools. The first server prototype helped to reveal a bottleneck in odata4j framework, and consequently, these libraries were modified for speed-related needs of Infinispan OData server. Performance improvements in odata4j framework tremendously helped to speed up the server which is currently able to compete with Infinispan REST server under respective load.

Bibliography

[1] SUMATHI, S a S ESAKKIRAJAN. *Fundamentals of relational database management systems*. London: Springer, 2007, xxv, 776 p. ISBN 35-404-8397-7.

[2] DATASTAX CORPORATION. Why NoSQL?. In: *DataStax* [online]. 2013 [cit. 2014-01-04]. Available from: `http://www.datastax.com/wp-content/uploads/2012/10/WP-DataStax-WhyNoSQL.pdf`

[3] MCCREARY, Dan a Ann KELLY. *Making sense of nosql: a guide for managers and the rest of us*. S.l.: O'Reilly Media, 2013, 286 p. ISBN 978-161-7291-074.

[4] TIWARI, Shashank C. *Professional NoSQL*. 1st ed. Indianapolis, IN: Wiley Publishing, Inc., 2011, p. cm. ISBN 04-709-4224-X.

[5] COUCHBASE. Why NoSQL?: Three trends disrupting the database status quo. In: *Couchbase | Document-Oriented NoSQL Database* [online]. 2013 [cit. 2014-01-04]. Available from: `http://info.couchbase.com/WhyNoSQLWhitepaper.html`.

[6] VAISH, Gaurav. *Getting started with NoSQL*. S.l.: Packt Publishing Limited, 2013. ISBN 978-184-9694-988.

[7] SEOVIĆ, Aleksandar, Mark FALCO, Patrick PERALTA, Cameron PURDY a Tangosol FOUNDER. *Oracle Coherence 3.5: create internet-scale applications using Oracle's high-performance data grid*. Birmingham, U.K.: Packt Pub., 2010. From technologies to solutions. ISBN 978-1-847196-12-5.

[8] MARCHIONI, Francesco a Manik SURTANI. *Infinispan data grid platform*. Birmingham: Packt Publishing, 2012, iv, 132 p. ISBN 9781849518222.

[9] *Infinispan Homepage · Infinispan* [online]. 2009-2013 [cit. 2014-01-01]. Available from: `http://infinispan.org/`

[10] Getting Started with Infinispan. *Infinispan Homepage · Infinispan* [online]. 2009-2013 [cit. 2014-01-01]. Available from: http://infinispan.org/docs/6.0.x/getting_started/getting_started.html

[11] MARKUS, Mircea. What's new in Infinispan 6.0. In: *Upload & Share PowerPoint presentations, documents, infographics* [online]. 2013 [cit. 2014-01-01]. Available from: http://www.slideshare.net/JBUG_London/whats-new-in-infinispan-60/

[12] Infinispan User Guide. *Infinispan Homepage · Infinispan* [online]. 2009-2013 [cit. 2014-01-01]. Available from: http://infinispan.org/docs/6.0.x/user_guide/user_guide.html

[13] Memcached/doc/protocol.txt at master · memcached/memcached. *GitHub* [online]. 2014 [cit. 2014-01-01]. Available from: https://github.com/memcached/memcached/blob/master/doc/protocol.txt

[14] ZAMARREŇO, Galder. Infinispan's Hot Rod Protocol. In: *Upload & Share PowerPoint presentations, documents, infographics* [online]. 2013 [cit. 2014-01-01]. Available from: http://www.slideshare.net/galderz/hot-rodjud-con2010/

[15] NISTOR, Adrian. Infinispan: Embedded and remote queries in Infinispan 6.0.0.Beta1. *Infinispan* [online]. 2013 [cit. 2014-01-01]. Available from: http://blog.infinispan.org/2013/09/embedded-and-remote-queries-in.html

[16] BAN, Bela a Vladimir BLAGOJEVIC. Reliable group communication with JGroups 3.x. *JGroups - The JGroups Project* [online]. 2002-2013 [cit. 2014-01-01]. Available from: http://www.jgroups.org/manual-3.x/html/index.html

[17] *Home - Open Data Protocol | OData* [online]. 2013 [cit. 2014-01-04]. Available from: http://www.odata.org/

[18] CHAPPELL, David. Introducing OData: Data Access for the Web, the Cloud, Mobile Devices, and More. In: *David Chappell :: White Papers* [online]. 2011 [cit. 2014-01-04]. Avail-

able from: http://www.davidchappell.com/writing/white_papers/Introducing_OData_v1.0--Chappell.pdf

[19] *Odata4j - An OData framework for Java* [online]. 2013 [cit. 2014-01-04]. Available from: http://www.odata4j.org/

[20] Overview | Open Data Protocol | ODataOpen Data Protocol | OData. *Home | Open Data Protocol | ODataOpen Data Protocol | OData* [online]. 2013 [cit. 2014-01-04]. Available from: http://www.odata.org/documentation/overview/

[21] OData Core | Open Data Protocol | ODataOpen Data Protocol | OData. *Home | Open Data Protocol | ODataOpen Data Protocol | OData* [online]. 2013 [cit. 2014-01-04]. Available from: http://www.odata.org/documentation/odata-v3-documentation/odata-core/

[22] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, Clemente Izurieta *Comparison of JSON and XML Data Interchange Formats: A Case Study* [online]. 2009 [cit. 2013-12-27]. Available online: http://www.cs.montana.edu/izurieta/pubs/caine2009.pdf

[23] FieldBridge (Hibernate Search 4.4.0.Final). *Community Projects - JBoss Community* [online]. 2006-2013 [cit. 2014-01-03]. Available from: http://docs.jboss.org/hibernate/stable/search/api/org/hibernate/search/bridge/FieldBridge.html

[24] THE APACHE SOFTWARE FOUNDATION. *Maven - Welcome to Apache Maven* [online]. 2002-2014 [cit. 2014-01-04]. Available from: http://maven.apache.org/

[25] Visitor Pattern | Object Oriented Design. *Design Patterns | Object Oriented Design* [online]. 2005 [cit. 2014-01-01]. Available from: http://www.oodesign.com/visitor-pattern.html

[26] *PerfCake – A Lightweight Performance Testing Framework* [online]. 2011-2013 [cit. 2014-01-01]. Available from: https://www.perfcake.org/

[27] Java Support for Large Memory Pages. *Oracle | Hardware and Software, Engineered to Work Together* [online]. 2014 [cit. 2014-01-03]. Available from: `http://www.oracle.com/technetwork/java/javase/tech/largememory-jsp-137182.html`

[28] GRINOVERO, Sanne a Adrian NISTOR. Infinispan/query/src/test/resources/indexing-perf.xml at master · infinispan/infinispan. *GitHub* [online]. 2013 [cit. 2014-01-05]. Available from: `https://github.com/infinispan/infinispan/blob/master/query/src/test/resources/indexing-perf.xml`

A Following of the OData standards

This section provides quick summary of the most important OData conventions which are followed by Infinispan OData server implementation.

HTTP methods

`POST` – For a creation of a new JSON document.
`GET` – For accessing stored JSON documents.
`PUT` – For replacing JSON document in the store.
`DELETE` – For removing JSON document from the store.

HTTP response codes

Successful response for `POST` request: HTTP status code `201`.
Successful response for `GET` request: HTTP status code `200`.
Successful response for `PUT` request: HTTP status code `200`.
Successful response for `DELETE` request: HTTP status code `204`.
A response when entry was not found: HTTP status code `404`.

HTTP headers

Successful entry creation (HTTP `POST` request) returns also HTTP `location` header with an address at which the created document can be accessed.

Every response sets up HTTP `DataServiceVersion` header with the current supported version of OData service.

Standardized JSON response

Returned data from Infinispan OData service is serialized according to OData JSON standards. There can be distinguished two possible responses here: one particular JSON document, or a collection of more JSON documents can be returned.

One result is serialized this way („d“ stands for a „data“):

```
{ „d" :
    {„id":„person1",
     „firstname":„Neo",
     „lastname":„Matrix",
     „gender":„MALE",
     „age":26}
}
```

When a filter query returns more results, they are serialized as an array of document results:

```
General form: { „d" : [{ ... }, { ... }, { ... }]}
```

```
{ „d" :
   [{„id":„person1",
     „firstname":„Neo",
     „lastname":„Matrix",
     „gender":„MALE",
     „age":26},
    {„id":„person2",
     „firstname":„Trinity",
     „lastname":„Matrix",
     „gender":„FEMALE",
     „age":28}]
}
```

The fact that service provides access to `$metadata` document, which describes exposed data entity sets and operations using EDM, is also a part of the OData standards.

Support for system query options and operators belongs to OData standard as well and that is elaborated in appendix (B).

B Supported query options and operators

OData queries can be used for filtering a collection of entities in a result set. A `$filter` option can be appended to a function import and used for instance as

`http://ODataService.svc/service-op-name?$filter=<expression>`

where `<expression>` consists of operators and options. A list of supported filter queries operators can be found in table B.1. Note: operators has to be used lowercase.

OP	**Description**	**$filter=**
eq	equal	name eq 'John'
and	logical and	name eq 'John' and surename eq 'Smith'
or	logical or	name eq 'John' or name eq 'Jack'

Table B.1: Supported query operators (filter expressions).

A `$filter` is not only one option and can be extended by other supported system options. Infinispan OData service also supports: `top` and `skip` options (see table B.2).

OP	**Description**	**$filter=<expression>&**
$filter=<exp>	filters results	$filter=name eq 'John'
$top=n	selects top `n` entries	$top=5
$skip=n	skips `n` entries	$skip=5

Table B.2: Supported system query options.

From implementation point of view, `$filter` query is taken as the first, and results are filtered. Then, if specified, other system query options are applied and collection of results is returned; some results may be skipped, or only a few top results is returned. `$top` and `$skip` query options have to be used after `$filter` option. Other options will be implemented according to community requirements.

C Infinispan Hot Rod vs REST server

Server type	Load threads	Average response time	Average throughput	Max memory usage
Hot Rod	20	0.07 ms	237860 ops/s	1.891 GB
Hot Rod	40	0.35 ms	114285 ops/s	1.975 GB
Hot Rod	60	0.85 ms	70588 ops/s	1.982 GB
Hot Rod	80	1.26 ms	63492 ops/s	1.985 GB
Hot Rod	120	2.11 ms	56872 ops/s	1.987 GB
Hot Rod	200	3.83 ms	52219 ops/s	1.988 GB
Hot Rod	400	8.22 ms	48662 ops/s	1.991 GB
REST	20	3.42 ms	5848 ops/s	2.057 GB
REST	40	3.47 ms	11527 ops/s	2.052 GB
REST	60	3.69 ms	16260 ops/s	2.071 GB
REST	80	4.14 ms	19324 ops/s	2.088 GB
REST	120	6.32 ms	18987 ops/s	2.507 GB
REST	200	25.60 ms	7813 ops/s	3.770 GB
REST	400	101.90 ms	3925 ops/s	3.770 GB

Table C.1: Infinispan Hot Rod and REST server performance statistics.

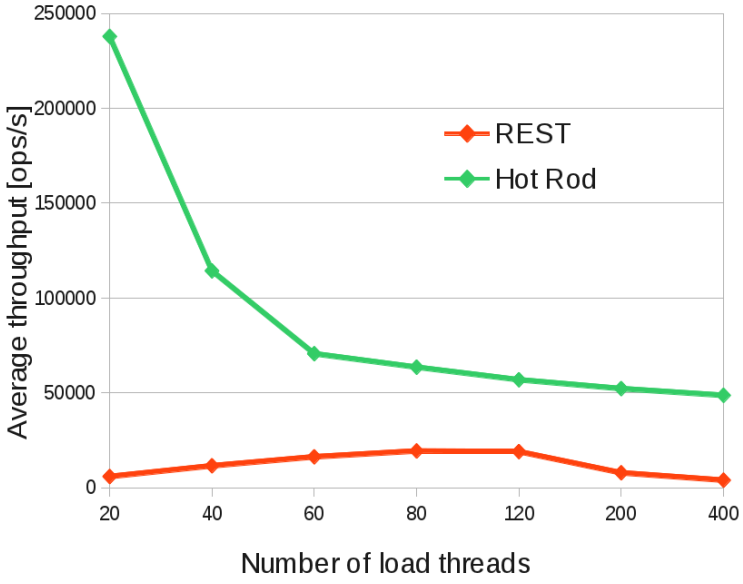

Figure C.1: Infinispan Hot Rod and REST server average operational throughput chart.

D Content of the attached zip file

- **Infinispan OData server source code**

  – located in folder `infinispan-odata-server`.

  Project `README.md` file contains information about the process of building, running the server and practical usage examples is located in main project directory. Infinispan OData server depends on `odata4j` libraries of version 0.8.0-SNAPSHOT (see attachment below).

- **Built distribution of Infinispan OData server**

  – `infinispan-odata-server-1.0-SNAPSHOT.jar`
  file as 1.0-SNAPSHOT distribution of OData Infinispan server; ready to be started and used.

- **Source code of modified odata4j framework for needs of Infinispan OData server**

  – located in folder `odata4j-actions`; version 0.8.0-SNAPSHOT.

- **Built odata4j framework libraries**

  – located in folder `odata4j-0.8.0-SNAPSHOT-libraries`;

- **Infinispan cakery source code**

  – located in folder `infinispan-cakery` together with `README.md` file describing the process of running this tool.

- **Thesis LaTeX source code**

  – located in folder `thesis`