Hibernating in the Cloud – Implementation and Evaluation of Object-NoSQL-Mapping *

Florian Wolf, Heiko Betz, Francis Gropengießer, and Kai-Uwe Sattler

Database and Information Systems Group
Ilmenau University of Technology
{*first.last*}@tu-ilmenau.de

Abstract: Object-relational mappers such as Hibernate are often used in applications to persist business objects in relational databases. The availability of commercial cloud-based database services opens new opportunities for developing and deploying database applications. In addition, highly scalable cloud services belong to the class of NoSQL systems promising to avoid the paradigm mismatch between the object-oriented programming model and the relational backend. In this paper, we discuss and analyze the usage of a scalable NoSQL solution such as Basho's RIAK as backend for Hibernate. We describe the necessary mapping and translation steps for an integration avoiding the detour on SQL. Finally, we present results of an experimental evaluation showing the benefits and limitations of this class of NoSQL backends for object-relational mappers.

1 Introduction

Today, modern business applications rely mainly on application server technologies such as Java EE or .NET where business entities like customers, products, or orders are represented by components or objects of an object-oriented language. In order to persist these objects, often SQL database systems are leveraged requiring an object-relational mapping. For this purpose, several frameworks exist. Most prominent examples are Hibernate [JBo], ADO.Net [ADO], and DataNucleus [Data], just to mention only a few. Although, object-relational mapping works quite well for most applications, it faces at least two drawbacks resulting from the underlying relational database technology:

Paradigm mismatch: Though, object-relational mappers hide the details and complexity of the mapping of the data structures and translation of object accesses to SQL queries, there is still a major difference. While in object-oriented applications traversing the graph of objects is the typical access pattern, operations in the relational model are set-oriented and access tuples by their attribute values. Thus, traversing an object relationship or retrieving a complex object results in complex joins and outer joins of multiple tables. This is also known as object-relational impedance mismatch [New06].

*This work was supported by the Thüringer Aufbaubank (TAB) under grant 2011 FE 9005.

Scalability issues: Relational database systems aim to provide strong consistency guarantees. In large distributed settings as required for example in Web 2.0 applications, these systems reach the limits in terms of scalability according to the CAP theorem [FGC$^+$97].

A possible approach to overcome these shortcomings is to use so-called NoSQL systems like Amazon DynamoDB [AWS], Cassandra [Apa], RIAK [Bas], or Neo4j [Neo]. These systems favor scalability and availability over consistency. Furthermore, they provide more flexible schema support and evolution. For instance, a graph model as supported by Neo4j seems to be better suited for persisting object graphs since graph traversal is more similar to the typical access pattern of object-oriented applications than performing joins. Secondly, systems like DynamoDB do not require a schema in the classic sense: each object can have its own set of attributes which makes it easier to add or remove new object classes than the rather limited `ALTER TABLE` operations in SQL.

Basically, two approaches for persisting application data in a NoSQL system exist – *Persistence API* or *Mapping framework*. The former one is tailored to a specific NoSQL solution. Examples of this approach are Morphia [Mor], an object mapper for MongoDB [Mon], as well as HelenaORM [Hel] and Object-Cassandra-Mapper [OCM], which are object mapper for Cassandra. The latter approach can deal with different NoSQL systems. Examples are DataNucleus, DataMapper [Datb] as well as Hibernate OGM [Hib].

Obviously, the mapping framework approach is the more flexible one. Especially today, where many cloud providers like Amazon, Microsoft, or Google offer different NoSQL services with different features and pricing models, it enables application developers to write their applications without tailoring them to a specific system and allows also to migrate to other storage systems.

Although, several non-relational object mapping frameworks already exist, the most famous one – Hibernate – lacks general support for different NoSQL systems. Currently, only Infinispan [Inf] is supported. The problem of this solution is that it tries to map the relational model to the data model of the NoSQL system, which leads to the mentioned paradigm mismatch.

Hence, the question we try to answer in this paper is, whether it is possible to integrate NoSQL support in Hibernate by avoiding the intermediate mapping to the relational model and while keeping compatibility to existing applications. We address this question by presenting a general "non-relational mapping" approach and discussing the integration in the Hibernate framework. Furthermore, we describe the implementation of this approach using the RIAK system as backend for Hibernate and discuss still existing and inherent limitations. Beside the functional properties, we investigate performance and scalability of our Hibernate/NoSQL in comparison to a setup with a traditional MySQL backend.

2 Related Work

In fact, the discussion about the usefulness of relational databases as storage backends for all kinds of applications dates back to the early 80s. Already in [HL82], limitations of relational database system, for example missing support for complex types and interactive access patterns needed for efficient support of engineering design, are described.

The emerging success of the object-oriented programming language as well as the need for adequate storage backends for CAD systems [Wol91] led to the development of a variety of so called non-standard database systems. In [CM84] the authors propose GemStone – one of the first commercial object-oriented database system. Object-oriented databases try to avoid a paradigm mismatch, as existing between object-oriented applications and relational database systems, by enabling native persistence of application objects. They fully support object-oriented principles such as encapsulation, inheritance, or object relations. Further examples for non-standard database solutions are KUNICAD [HHLM87], a database system to support geometric modeling for CAD applications, or PRIMA [HMWM$^+$87], a database system to support design applications such as VLSI design and software engineering .

With the beginning of the 90s, the trend was shifting from object-oriented database systems to object-relational database systems [SM95]. The absence of standardized data models and declarative query languages were only some reasons for this development [Bro01]. Object-relational database systems are an extension of relational database systems – combining object-oriented principles with a powerful query language. Examples are Informix [IBM] or Oracle [Ora]. However, the impedance mismatch could not be fully solved with this technology, preventing a total success of object-relational database systems.

A contrary approach to object-relational database systems is the integration of SQL into the application layer. Well known techniques here are Module Language, Embedded SQL, or Direct Invocation [VP95]. One of the goals of these developments is to solve the impedance mismatch by abandon the definition of an object-relational mapping. However, one of the problems is the tight binding of the application to a specific database. Switching the database can lead to deep changes in the application code.

With the development of CLI (Call Level Interface), the binding got more abstract. JDBC and ODBC are concrete implementations of this standard. They can be seen as important steps on the way to object-relational mapping frameworks already mentioned in the introduction.

Driven by the current trends in cloud computing, object mapping to NoSQL storage backends becomes more and more important. NoSQL systems provide the availability and scalability guarantees needed in today's business applications. Some examples for appropriate object-to-NoSQL mappers like Morphia or HelenaORM are already mentioned in the introduction. A further famous commercial solution is Google's App Engine [App]. It provides built-in mapping from Java objects to Google's BigTable [CDG$^+$06].

Summarizing, the problem of persistent storage of application objects is an old problem. Using relational backends has the advantage of a proven, standardized data model as well as a powerful query language. However, due to different paradigms this introduces problems summarized under the term impedance mismatch [IBNW09]. Alternatives are native backends like object-oriented database systems. The current developments in storage techniques along with the increasing success of cloud computing are making native storage solutions more attractive again and provide the context for our work.

3 Hibernate ORM Framework

The Hibernate ORM framework is an Open Source project which allows for easy persistence of Java application objects in a relational fashion, using almost any available relational database system as storage backend via JDBC. Its data model follows the object-oriented approach, requiring only less modifications to the application code. Hibernate also supports relationships (associations) between objects. 1:1 associations are implemented by object references as members, whereas 1:N, N:1, N:M associations are implemented by embedded Java collections.

Basically, each Java class is mapped to a table and each object is mapped to a table row, whereby the object properties (class members) constitute table columns. Furthermore, associations between objects are stored in normalized form using foreign key relations. Necessary relation tables (join tables) for N:M associations are automatically created.

Persisting and deleting objects is supported by simple save, update, and delete operations. Retrieving objects is either performed manually using SQL-like HQL (Hibernate Query Language) queries or the Hibernate Criteria API or automatically during object access. In many applications, object retrieval is typically performed by *i)* querying an object that acts as an entry point and *ii)* traversing associations to other objects by accessing the object's references via the provided getter methods. Thereby, referenced objects are automatically retrieved by the Hibernate framework.

Under the hood, data model mapping, persisting and retrieving objects as well as traversing object associations result in relational database specific SQL statements. While this works well in case of simple object persistence and retrieval, several problems arise in case of association traversal due to the underlying relational data model (impedance mismatch), for which we give a short example in the following.

Many applications make extensive use of hierarchical data structures. In the simplest case, they implement trees, where nodes are just connected by parent-child relationships. Hibernate supports the bidirectional storage of these relationships. Every node in the tree stores a reference to its parent as well as a set of references to its children. In other words, all associations are stored with the object and hence, traversal in both directions – top down and bottom up – is well supported.

The relational data model treats trees as 1:N relationships. In normalized form, relations are stored unidirectionally at the N-side. Hence, bottom up traversal is favored. Retrieving a node's children means either joining parent and children table or performing a selection on the children table with the parent's key. If we just want to obtain the number of a node's children, we have to perform a `SELECT COUNT` on the children table. In the object oriented data model we just retrieve the size of the reference set.

In the general case, the relational data model separates objects from their associations through relation tables. This leads to additional scans and joins and hence a behavior that contradicts the object-oriented approach. Furthermore, the fixed relational schema prevents easy adaption to new associations. It is not possible to easily extend one-to-many associations to many-to-many associations. This would result in significant changes to the schema.

Since NoSQL storage solutions do not rely on fixed schemas, they seem to be well-suited candidates to build a storage backend that comes closer to the object-oriented data model. Hence, we try to solve the above stated problems by defining a non-relational object mapping described in the following sections.

4 RIAK as Hibernate Backend

Looking at the current market of existing cloud based NoSQL solutions reveals that two different storage models exist – wide-column stores and key-value stores. Wide column stores provide a schema-less and flexibly structured data model comparable to spreadsheets with access to single columns of a row. In contrast, key-value stores treat data just as BLOBs, which are accessible only as a whole via unique keys. Although, the data model is more low-level than the one provided by wide-column stores, we have chosen the key-value store approach instead of a wide-column store due to the following reasons. Firstly, wide-column stores typically rely on key-value stores in the background. Hence, we can create our own wide-column store with custom features if needed. Secondly, Hibernate only retrieves complete objects. Partial access to single properties of an object is not needed. Hence, usually it is not necessary to store data in a spreadsheet-like fashion. However, if objects shall be selected based on certain properties other than keys, knowledge of the internal structure is indeed required. In order to support these kinds of selections, most providers have introduced MapReduce support, e.g., Amazon with Elastic MapReduce [AWS], which works seamlessly with underlying key-value stores. By this means, selections on those BLOBs are as efficient as in a wide-column store.

For our work, we have chosen RIAK as a concrete key-value store backend for Hibernate. RIAK is open source and roughly comparable to Amazon S3 [AWS]. Data is organized by buckets and keys. Buckets define a virtual key space and, hence, group data logically. A value is uniquely identified by the bucket name and the key. It contains BLOB data as well as an arbitrary set of the so-called RIAK links for establishing references to other key-value pairs. A single RIAK link contains the bucket and key of the referenced key-value pair. Additionally, it is possible to name an association by specifying a tag. RIAK links are explicitly accessible through RIAK's API.

RIAK's API provides the following main operations: get/put/delete a single key-value pair, list existing buckets as well as keys within a specified bucket, and get/set RIAK links on a value. Additionally, it supports MapReduce integration, e.g., the built-in link-walk for traversing RIAK links. Starting from a given key-value pair it is possible to follow the outgoing links.

5 Hibernate on RIAK – Integration Concept

Integrating a key-value store like RIAK in an object-relational mapping framework comprises two tasks: mapping the object-oriented domain model to the data structures of the key-value store and replacing the translation to SQL statements by API calls of the key-value store. In the following sections we describe both steps for RIAK. However, it should be noted that this approach can be easily applied also to other key-value stores, including Amazon's cloud services.

5.1 Data Model Mapping

With having both RIAK's data model and API in mind, mapping Hibernate's class and object model is a straightforward approach. Each object is mapped to a single key-value pair. For each class, a bucket is created. In this way, all instances of a certain class can be retrieved with list bucket operations. The properties of an object are stored within the value of the key-value pair. As serialization format we use JSON, since it is easy to use and supported on most platforms. Associations between objects are represented by RIAK links.

Associations between objects can be stored either unidirectionally or bidirectionally. In the latter case, two related objects link each other. Thus, RIAK linking is very flexible compared to the relational data model. Reconsider our tree example from Section 3. In RIAK, we can store both – the link to the parent within the child objects as well as the links to the children in the parent object. If just the number of children or their keys are needed, only a single lookup of the parent object has to be performed. Furthermore, we can easily extend one-to-many associations to many-to-many associations and, hence, represent general object graphs by just adding new links.

5.2 Connecting Hibernate and RIAK

For integrating Hibernate and RIAK, the most interesting parts of the Hibernate architecture [JBo] are the *Session* and *JDBC* components. The session component provides an interface containing core functionality for persisting and retrieving objects. The JDBC interface connects Hibernate to a specific relational database by loading an appropriate JDBC driver. In the following, we discuss possible anchor points where RIAK support could be plugged in.

Session-API reimplementation: A first approach is to replace the original implementation of the Session component by a RIAK-specific component. In this way, we could fully exploit all RIAK specific features. However, in this case almost the entire functionality of Hibernate, including object life-cycle management, has to be reimplemented requiring a tremendous effort.

JDBC-API reimplementation: The second approach is to provide a RIAK driver which implements the JDBC API. The disadvantage of this approach is that it requires parsing all SQL statements and rewrite them to RIAK API calls – something that we try to avoid.

Hooking into the Session component: The most promising approach is to plug in the interaction with RIAK into the Session component. In this way, we have access to SQL-independent data structures and still profit from core Hibernate functionality like object-life-cycle management. Hence, we have chosen this approach and describe the details in the following.

Based on the discussion above, RIAK backend support is integrated as shown in Figure 1. In fact, all highlighted components have been modified in order to intercept and redirect communication to our *RIAK mapping framework* instead of calling the JDBC interface.

In order to save, update, and delete objects (label 0) during object-life-cycle-management, Hibernate provides several `Persister`-classes for different purposes. If the `Persister` needs already stored objects, e.g., for resolving associations, it contacts the `Loader` (label 3). The `Loader` fetches the required objects from the persistent storage and returns them to the `Persister`. Similar to the `Persister`, several `Loader` classes for different purposes exist. In order to retrieve objects via HQL or the Criteria API (label 1 and 2) specific `QueryTranslators` are used to parse the query in an abstract syntax tree (HQL AST). In this case, the `Loader` is only responsible for redirecting the HQL AST and returning the results as application objects.

The core of our extension is the `Engine` component. It performs the mapping from objects to key-value-pairs with the help of JSON serializers and deserializers. Furthermore, it initiates RIAK API operations. Following, we describe our extension in case of saving an object, traversing an 1:N association, and executing a HQL query in more detail.

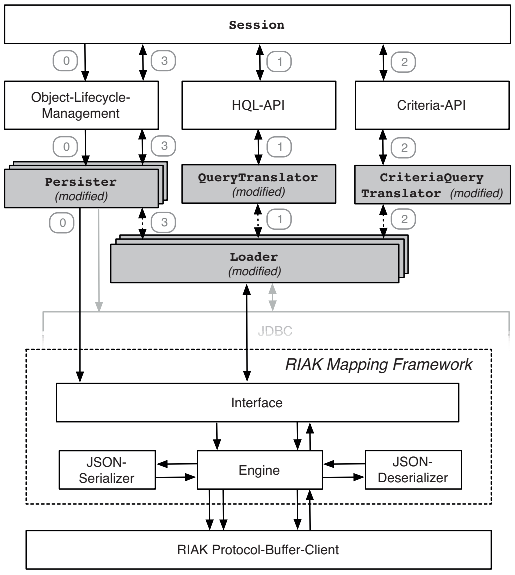


Figure 1: RIAK integration within Persister/Loader

For saving an object in RIAK, the object identifier and the object's entity type are required. Additionally, information about the attributes of the object and its associations are needed. Figure 2 depicts the resulting information flow.

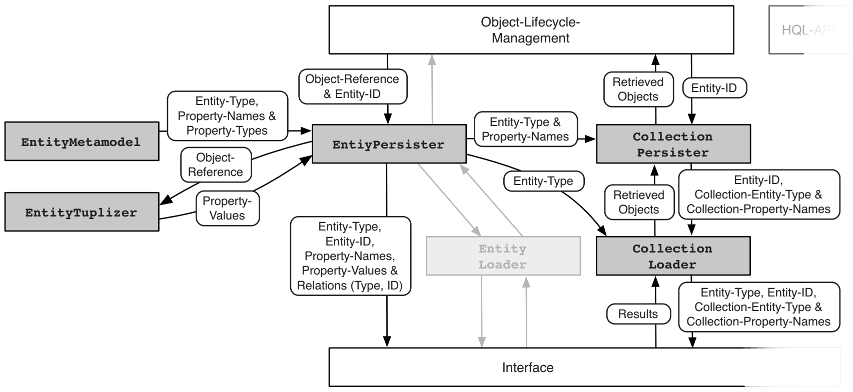

Object-Lifecycle-Management
HQL-API
Object-Reference & Entity-ID
Retrieved Objects
Entity-ID
Entity-Type, Property-Names & Property-Types
EntityMetamodel
EntiyPersister
Entity-Type & Property-Names
Collection Persister
Object-Reference
Entity-Type
Retrieved Objects
Entity-ID, Collection-Entity-Type & Collection-Property-Names
EntityTuplizer
Property-Values
Entity-Type, Entity-ID, Property-Names, Property-Values & Relations (Type, ID)
Entity Loader
Collection Loader
Results
Entity-Type, Entity-ID, Collection-Entity-Type & Collection-Property-Names
Interface

Figure 2: Saving an object and traversing 1:N associations

A Java object that has to be saved is passed to the `EntityTuplizer` by the `EntityPersister` for extracting the object's property values. The necessary information about the object type, its properties, and its associations are represented by the `EntityMetamodel`. The `EntityPersister` component has been modified to pass the information to the RIAK mapping framework which is responsible for storing the object persistently.

Performing a single traversal step during a top-down traversal in a tree leads to the information flow shown on the right hand side in Figure 2. Because in this case a 1:N association has to be resolved, Hibernate uses the appropriate `CollectionPersister` and `CollectionLoader` classes. The `CollectionLoader` has also been modified. It extracts information about the entity and the collection and forwards them to the RIAK mapping framework. The mapping framework retrieves all related objects and returns them as JDBC result set – a format natively understood by the `CollectionLoader`. In this way, we do not have to take care about proper object creation which is still done by the Hibernate framework.

Figure 3 illustrates the information flow during the processing of a HQL query. The query string is translated into a HQL AST by the `HqlParser` and passed to the `Walker` which extracts necessary information like requested entity types or property names. For interacting with the RIAK mapping framework, the `QueryTranslator` and `QueryLoader` have been slightly modified. The RIAK mapping framework retrieves the requested data from the storage backend and passes them back to the `QueryLoader` as JDBC result set.

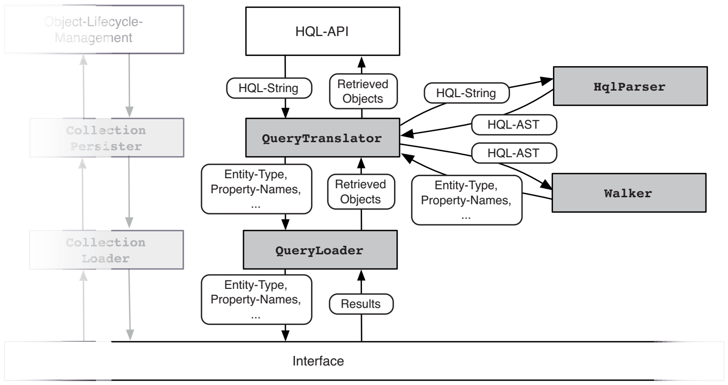


Figure 3: HQL-Query Loading

All object-related operations such as save, update, delete, and query are implemented in the RIAK mapping framework by performing appropriate RIAK API calls. The required information are gathered from the Hibernate core as described above. For saving an object, the value is obtained by serializing the object into a JSON string. The `EntityID` represents the necessary key. All associations to other objects are added as links to the meta-information of the object. For retrieving a 1:N association, the object on the 1-side is retrieved with the help of the provided `EntityID`. Using the `CollectionEntityType` ,the type (and hence the bucket) of the N-side objects can be identified. The linked objects (RIAK values) are retrieved in parallel by several threads. The results are deserialized and stored in a shared result list.

6 Evaluation

In the previous section, we have shown that the integration of RIAK in Hibernate as storage backend is basically possible. Instead of forcing a relational mapping to RIAK, we chose a storage model which is close to the object-oriented data model in a way that associations are stored with objects and not separated from them. In this section, we investigate whether this approach is beneficial in terms of performance and scalability. In order to better classify the test results, we compare them to a centralized relational backend. We chose MySQL [MyS], since it is freely available and widely used. However, we stress that the goal of this evaluation is not a comparison of both backends. Instead, MySQL just acts as a baseline to be able to assess the performance and scalability behavior of the NoSQL backend.

6.1 Micro Benchmark

To the best of our knowledge, no standardized Hibernate benchmarks exist. Hence, we perform micro benchmarking in order to compare both approaches. Our workload comprises three scenarios which are typical for most business applications – retrieving a single object, storing a single object, and traversing object associations. The first scenarios describe the retrieval of an existing instance and the storing of a new instance of a given class. Typical examples are requesting information regarding a given customer or creating a new customer. The latter scenario focuses on the most critical part of the object-relational mapping – the association handling. A widely spread traversal scenario is, as already mentioned, tree-traversal – either top-down or bottom-up. A typical example is: retrieving all order positions from a given order belonging to a given customer.

The test data is randomly generated. A class comprises ten attribute types – five string and five integer values. Objects are uniquely identified by a key (MD5 hash value). In MySQL, the key is used as primary key. In RIAK, the key is used to unambiguously identify a key-value pair.

The used metric is the time needed for storing and retrieving objects. The time is measured between issuing the operations and accessing the operation results. Including the time for accessing the results is necessary, because Hibernate works with lazy fetching. In detail this means, an object is not initialized till the first method invocation occurs. Based on the measured time, other metrics can be calculated, e.g., the number of queries per seconds.

During the tests, the number of objects in the data store, the number of concurrent client requests as well as the object size are changed. In order to prevent cache issues, the whole data store is cleaned between different tests. In order to get stable results, the tests are executed several times with different keys.

6.2 Test Setup

For both setups (RIAK and MySQL), we use virtual machines which are distributed over seven physical hosts connected by 1 GBit/s Ethernet. Each host has two Intel Xeon CPUs E5645 (12 cores and 24 threads; 2.40 GHz) and 40 GB of RAM. Each virtual machine has 1 CPU, 8 GB of RAM, 512 MB of Swap, and 4 GB of hard disk space.

The RIAK setup consists of 32 nodes. The number of partitions is also set to 32. Each node corresponds to a single virtual machine. The storage mechanism in RIAK is set to `riak_kv_memory_backend`. This means, all data is stored in main memory. All further configuration options are set to default values.

MySQL runs in a single virtual machine, with 40 GB of hard disk. No cluster or replication techniques are used. All configuration options are set to default values. MyISAM is used as storage engine. The primary key in each relation is indexed.

The client machine executing the tests is also virtual and has 12 cores. The used Java version is 1.6.0_26. All further configuration options are set to default values.

Of course, comparing a hard disk setup against an main memory setup is quite unfair. However, letting RIAK read/write from disk would just result in an offset shift of the access times. It does not influence the trend of the access times due to RIAK's distributed characteristics.

6.3 Hypotheses

Following, we outline the hypotheses for our evaluation. They summarize RIAK's performance and scalability behavior we expect during the execution of the test scenarios.

RIAK is based on consistent hashing [KLL$^+$97] which distributes data across all nodes equally. Hence, we expect that the time for storing and retrieving a single object scales at most linearly with increasing number of stored objects. In MySQL, the index is used for retrieval and has to be updated in case of an insertion. Hence, we expect that times for storing and retrieving a single object increase with the increasing number of stored objects.

The time for storing and retrieving a single object depends on its size. Here, we expect a similar behavior for both backends. The times for storing and retrieving a single object increase with its increasing size due to more data traffic.

With the growing number of parallel requests, we expect at most a linear scaling in RIAK, since all requests can be equally distributed across all nodes. Since MySQL is centralized, it scales worse than RIAK.

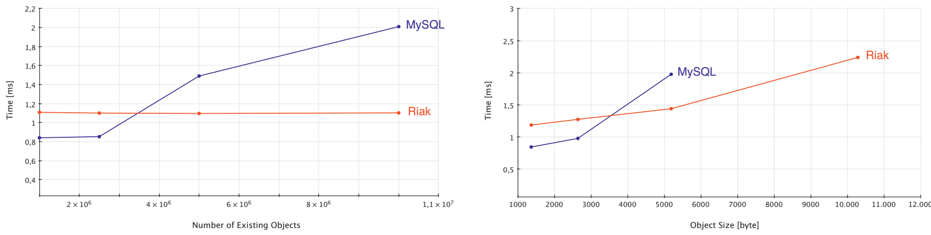
Top-down and bottom-up traversal can be performed very efficiently with RIAK. In the first case, child objects of a given parent object are identified via RIAK links and are fetched by parallel working threads. With MySQL, each child object is fetched via a select query. Although, this can also be done in parallel, we expect worse performance than with RIAK, because of the lack of horizontal scalability. In case of bottom-up traversal, MySQL performs joins. In RIAK, every child holds a RIAK link to its parent. We expect a performance drop by using MySQL with the increasing number of stored objects, whereas RIAK scales at most linearly.

In case of top-down traversal, we expect that MapReduce performs better than our externally implemented traversal algorithm, because data is directly processed in parallel on the RIAK nodes. Although, MapReduce introduces additional overhead, we expect better scaling with the increasing number of child objects compared to the externally implemented traversal algorithm.

6.4 Implementation and Analysis

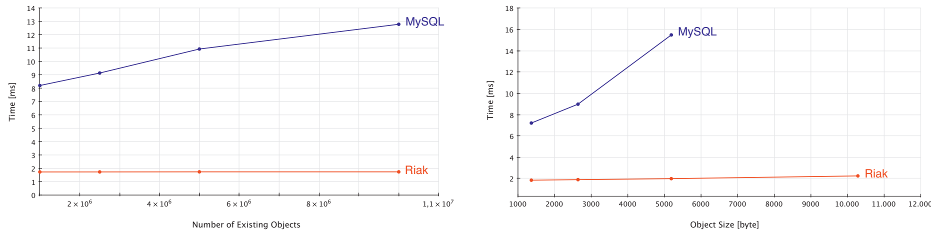
Retrieving a single object A single object is retrieved by its unique key. Figure 4(a) and 4(b) show the results. RIAK introduces additional overhead due to its distributed character. This leads to worse response times compared to MySQL in case of a small number of stored objects (Figure 4(a)). However, as expected, the response times evolve almost constantly with the increasing number of stored objects. At a storage size of 500,000 objects, RIAK starts outperforming MySQL. The response times in case of an increasing object size also evolve as expected. With default configuration, MySQL supports only row sizes up to approximately 8,000 Byte without using BLOBs or text. Hence, we finished testing at this point.

Storing a single object A new object is stored under a unique key. Figure 5(a) and Figure 5(b) show the results. As in case of object retrieval, the times for storing an object



(a) Object Size 1362 Byte (b) Number of Existing Objects 1000,000

Figure 4: Retrieving a Single Object

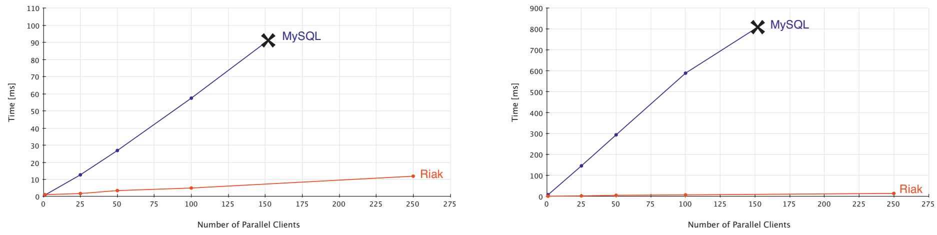


(a) Object Size 1362 Byte (b) Number of Existing Objects 1000,000

Figure 5: Storing a Single Object

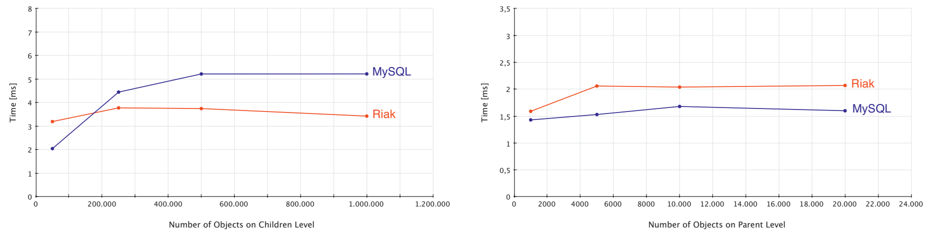
in RIAK evolve almost constantly. Here, MySQL performs worse in all cases. Reasons might be, as already mentioned, additional index updates and hard disk accesses. The latter aspect has tremendous impact on the performance with an increasing object size (Figure 5(b)). With RIAK, storage times only increase slightly with an increasing object size. This is in contrast to our expectation.

Scalability In this scenario, we investigate the scalability characteristics of both backends under an increasing number of concurrently working clients. Every client retrieves or stores an object. Figure 6(a) and 6(b) show the measured times when all clients finished operation execution. As expected, RIAK scales almost constantly with an increasing load size. MySQL shows worse performance values. The highlighted cross in both charts indicates a MySQL connection problem. It was not possible to connect more than 150 clients simultaneously.



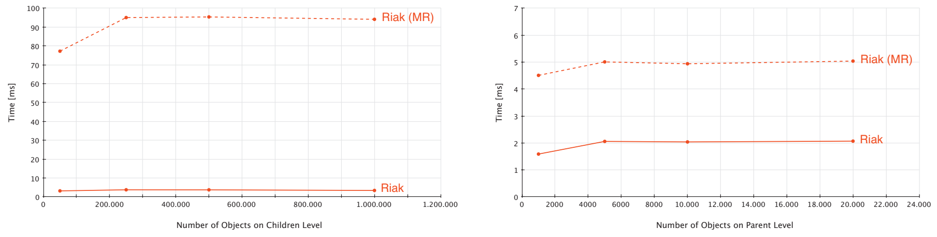
(a) Retrieving a Single Object (b) Storing a Single Object

Figure 6: Object Size 1362 Byte, Number of Existing Objects 1000,000



(a) Top-Down Traversal (b) Bottom-Up Traversal

Figure 7: 1 Level Traversal, 50 Children per Parent, Object Size 1362 Byte



(a) Top-Down Traversal (b) Bottom-Up Traversal

Figure 8: MapReduce, 1 Level Traversal, 50 Children per Parent, Object Size 1362 Byte

Traversing object associations Here, we focus on top-down as well as bottom-up traversal in a tree structure. In the first case, we retrieve all child objects belonging to a given parent object. In the latter case, we retrieve a given child's parent. Figure 7(a) and Figure 7(b) depict the retrieval times for RIAK and MySQL. As expected, during top-down traversal, RIAK scales almost constantly with an increasing number of children, since all child objects can be retrieved in parallel. Due to the small amount of child objects, also MySQL shows good retrieval times. However, performance might drop if the number of child objects increases tremendously as depicted in Figure 4(a). The bottom-up traversal reflects the overhead in RIAK due to its distributed character. Here, only a single object, the parent of a given child, is retrieved. Although, MySQL executes a join in order to determine the parent of a given child, it performs better than RIAK. However, if we would increase the number of levels during traversal, this would result in more join operations which increase traversal times. In RIAK, since every child knows its parent (bidirectional RIAK links), the number of retrieved objects equals the number of traversal levels. This might have a positive effect on retrieval times.

Figure 8(a) and Figure 8(b) depict the comparison of our externally implemented traversal algorithm and the MapReduce traversal algorithm. In contrast to what we expected, our algorithm performs better than MapReduce. The reason might be, that our algorithm retrieves objects in parallel (with parallel connections) whereas in the case of MapReduce all determined child objects are returned as an entire result over a single connection.

7 Conclusion

The goal of our work was to integrate the key-value store RIAK as backend in Hibernate, avoiding an intermediate mapping to the relational model. Thereby, we tried to answer the question, whether a scalable NoSQL backend (for which RIAK is just an example sharing many common properties with, e.g., Amazon DynamoDB and S3) is feasible and fulfills the expectations in terms of performance and scalability. Section 5 shows that such an integration is possible, but requires several changes in the Hibernate core. Currently, we have only implemented a subset of Hibernate's features including save, update, and delete operations as well as a basic HQL support. Exploiting all of Hibernate's features in combination with RIAK might require substantial modifications in the Hibernate framework. Especially, enabling full HQL support on RIAK is a challenging task.

In the evaluation part of this paper, we investigated performance and scalability behavior of RIAK and MySQL with respect to typical Hibernate application scenarios. The goal was not to compare both systems, but rather to let MySQL act as a baseline in order to answer the question if and when RIAK has performance advantages. Although, RIAK introduces some overhead which hampers performance in small scenario setups, the test results show that Hibernate profits from the distribution and scalability characteristics of RIAK. This is especially the case in situations, where system load in terms of data size, the number of objects, and the number of concurrent requests is increasing tremendously. The measured times for bottom-up traversal do not reflect RIAKs potential. However, with an increasing number of traversal levels, we expect a substantial performance boost. In contrast to our expectations, the built in MapReduce support does not lead to additional speedup in our test scenarios.

References

[ADO] ADO.NET Overview. `http://msdn.microsoft.com/en-us/library/h43ks021(v=VS.100).aspx`.

[Apa] The Apache Cassandra Project. `http://cassandra.apache.org`.

[App] Google App Engine. `https://appengine.google.com`.

[AWS] Amazon Web Services. `http://aws.amazon.com/`.

[Bas] Basho: Basho Documentation. `http://wiki.basho.com`.

[Bro01] P. Brown. *Object-relational database development: a plumber's guide*. Number Bd. 1 in Prentice Hall PTR Informix series. Prentice Hall, 2001.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15. USENIX Association, 2006.

[CM84] George Copeland and David Maier. Making smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, June 1984.

[Data] DataNucleaus. `http://www.datanucleus.org`.

[Datb] DataMapper. `http://datamapper.org`.

[FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *SOSP*, pages 78–91, 1997.

[Hel] marcust/HelenaORM. `https://github.com/marcust/HelenaORM`.

[HHLM87] Theo Härder, Christoph Hübel, Stefan Langenfeld, and Bernhard Mitschang. KUNICAD - A Database System Supported Geometrical Modeling Tool for CAD Applications (in German). *Inform., Forsch. Entwickl.*, 2(1):1–18, 1987.

[Hib] Hibernate OGM. `http://docs.jboss.org/hibernate/ogm/3.0/reference/en-US/html_single/`.

[HL82] Roger L. Haskin and Raymond A. Lorie. On extending the functions of a relational database system. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, SIGMOD '82, pages 207–212. ACM, 1982.

[HMWM+87] Theo Härder, Klaus Meyer-Wegener, Bernhard Mitschang, Andrea Sikeler, and Andrea Sikeler. PRIMA - a DBMS Prototype Supporting Engineering Applications. In *VLDB*, pages 433–442, 1987.

[IBM] IBM - Informix product family - Relational, embeddable and hassle-free offerings. `http://www-01.ibm.com/software/data/informix/`.

[IBNW09] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A Classification of Object-Relational Impedance Mismatch. In *DBKDA*, pages 36–43. IEEE Computer Society, 2009.

[Inf] Infinispan - Open Source Data Grids - JBoss Community. `https://www.jboss.org/infinispan`.

[JBo] Hibernate - JBoss Community. `http://www.hibernate.org`.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, pages 654–663. ACM, 1997.

[Mon] MongoDB. `http://www.mongodb.org`.

[Mor] Morphia - A type-safe java library for MongoDB. `http://code.google.com/p/morphia/`.

[MyS] MySQL :: The world's most popular open source database. `http://www.mysql.com`.

[Neo] Neo4j: World's Leading Graph Database. `http://neo4j.org`.

[New06] Ted Neward. The Vietnam of Computer Science. `http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`, 2006.

[OCM] charliem/OCM. `https://github.com/charliem/OCM`.

[Ora] Contents. `http://docs.oracle.com/cd/B19306_01/appdev.102/b14260/toc.htm`.

[SM95] Michael Stonebraker and Dorothy Moore. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., 1995.

[VP95] Murali Venkatrao and Michael Pizzo. SQL/CLI - A New Binding Style For SQL. *SIGMOD Rec.*, 24(4):72–77, December 1995.

[Wol91] Wayne Wolf. Object Programming for CAD. *IEEE Des. Test*, 8(1):35–42, January 1991.