ROYAL INSTITUTE
OF TECHNOLOGY

Providing a Data Model to the CATS key-value store

ALEXANDRU ADRIAN ORMENISAN

Master of Science Thesis in
Information and Communication Technology
Supervisor: Dr. Jim Dowling
Examiner: Prof. Seif Haridi
Stockholm, Sweden, June 2013

TRITA-ICT-EX-2013:179

Abstract

Search or social media giants are no longer the only individuals that face the problems of managing Big Data. Many of today's applications and services experience sudden bursts in growth, with increased data generation rates, that require storage and analysis support for large amounts of data. Traditional relational database management system (RDBMS) have been adapted to a distributed environment in an effort to make them suitable for Big Data, but they do not scale linearly and tend to obtain little extra performance as they grow in size. On the other hand, solutions built natively for a distributed environment, referred to as "Not only SQL" (NoSQL) provide a limited data model with few possible operations compared to structured query language (SQL). However, providing a data model with more complex, SQL like operations, raises some particular challenges in a distributed environment.

This thesis presents the design of a data model on top of the CATS key-value store. The purpose of this data model is to provide support for more complex data, compared to the simple key-value operations currently supported by CATS. Objects containing a number of fields can be stored and retrieved. Secondary indexes on different fields allow the search of objects based on the value of these indexed fields. The thesis also presents mechanisms for colocating data that is used together in order to reduce the latency of operations by exploiting data locality. The ability to dynamically adapt the way data is saved to disk according to different data access patterns can also help to provide faster services. The evaluation of a prototype of the system provides measurements on the overhead associated with the data model compared to the underlying key-value store.

Acknowledgements

I would like to express my gratitude towards my examiner *Prof. Seif Haridi* and my supervisor *Dr. Jim Dowling* for the opportunity to carry out this research. This thesis work would not have been possible without the guiding of *Dr. Jim Dowling*. Discussions with him always proved to generate new ideas that I could further research.
Second I want to thank *Dr. Cosmin Arad* for his valuable feedback during this period, as well as *Lars Kroll* and *Paris Carbone* for providing me with great input on my work.
Finally I would like to thank my parents for their love and continuous moral support.

Contents

Chapter 1

Introduction

The high connectivity granted by the internet, provides a prolific environment for applications and services, allowing them to get from an unknown starting service, to success in a small period of time. However this sudden growth in popularity can kill a service unless it can scale graciously, maintaining at least the same levels of quality.
The ability of an application to scale is frequently correlated with the scalability of the underlying storage layer which can be assessed through three parameters: volume of stored data, throughput and latency. Simply put, these three parameters tell how much can be stored, how many concurrent operations it can support and how slow is each operation.

In the age of information, data is being generated at high rates, with total data volume breaking one boundary after another - petabytes and exabytes in the last few years. Giants like Google, Amazon or Facebook, casually talk about storing and analyzing petabytes of data, requiring storage volumes that are no longer compatible with traditional storage solutions like relational database management system (RDBMS). In order to cope with this large volume of data, which received the name of *Big Data*, these giants turned to distributed solutions and developed their own data stores: Bigtable[1], Cassandra[2], Dynamo[3]. These new solutions, called "Not only SQL" (NoSQL), are described as an alternative to traditional, relational databases and include key-value stores like Dynamo, column oriented stores like BigTable or Cassandra, document databases like MongoDB[4]. They specialize for a particular problem where the performance of relational databases is considered not good enough, and give up some of the features that are considered unnecessary. In the case of BigTable, Dynamo and Cassandra they are tailored exactly for scaling with ease, being distributed in nature but they offer a relatively simple interface.

The value of data lies not just in storing it, but also in processing it and extracting useful information, so a processing framework is essential for any data store, including the new NoSQL solutions. The relational databases offer two types of data

processing: ad hoc interactive quering as online transaction processing (OLTP) and data warehouse, analytical processing as online analytical processing (OLAP). In order to offer a processing framework similar to OLAP, to process large quantities of static data, MapReduce[5] was developed. Within MapReduce, users would define their analytical jobs using *map* and *reduce* functions and the framework would parallelize them and execute them on the cluster of nodes the data store is running on. Following the emergence of MapReduce, similar new processing tools appeared, offering a more expressive interface for users to work with. Pig[6] and Hive[7] are two examples of such tools that are built on top of an open source implementation of MapReduce called Hadoop[8]. Other tools like Dryad[9] redesigned the whole processing framework. These tools provide a way to define and execute long-running, analytical jobs. Another tool called Dremel[10] offers query services over read-only data, stored in a columnar format. The importance of processing tools is clear and the new NoSQL community is working towards offering similar processing capabilities to the ones provided by relational databases.

The ease, with which these new data stores could scale, drew a lot of attention and many developers started using them. The interfaces provided by these NoSQL solutions are new and different from the relational databases and many developers found the lack of the familiar SQL and relational data model hard to overcome. Porting existing applications from the traditional relational databases was another problem, where something close to a full redesign was necessary in order to be compatible with the new data stores. Following the demand, solutions that would provide a more familiar interface appeared. At first, a solution was to provide a layer on top of existing data stores, like Megastore[11] which offered a semi-relational data model on top of BigTable. Shortly after, new data stores like Spanner[12], would incorporate these requirements from design.

1.1 Motivation

With the recent demands of applications, scalability has become mandatory and with this in mind, key-value stores received additional attention. CATS[13, 14] is a scalable key-value store, developed at the Swedish Institute of Computer Science (SICS), providing strong consistency of data through the use of a novel idea – consistent quorums. Despite its strong properties and good performance, the simple put/get key-value interface that it offers, augmented recently with key-value rangequeries[15] is too little for the complex usage many applications require.

It is clear that application developers expect a familiar and rich interface when choosing the underlying data store. With this in mind, the development of a richer data model on top of CATS is the next step for augmenting its interface. The data model will provide the ability to store and query more complex, structured data in the key value store.

Providing support for data processing is clearly important for every data store. An important part of an interactive query system is its searching capabilities. Being able to retrieve data only based on its key limits the use and performance of such a query system. In order to improve the search capabilities of the data store, secondary indexes can be used. The secondary indexes allow for improved performance when searching for data based on an attribute of the data different than its key. One such example is the following: if we want to store books in our data store, the key-value interface provided would mean that we would need to store the whole book as a value and its isbn code (unique per book) as its key. This means that we can easily retrieve books if we know their exact isbn code. If we want to search for books of some specific genre, we would need to scan the whole isbn code keyspace and check each book if it is of the genre we are searching for. If we use secondary indexes, we could index the books on the genre attribute and then searches based on genres will use the secondary indexes to replace the expensive exhaustive scan.

In order to improve the usability of a key-value store, there is a need for a data model that would allow storing of data, improved search and data processing capabilities.

1.2 Contribution

The main purpose of this thesis is to create a data model that can provide more complex operations on the stored data. Having this as a starting point, the contributions of this thesis are:

- providing support for storing complex data and access it by the use of primary and secondary indexes
- basic query support for stored data by the use of secondary indexes
- a mechanism for adapting the layout of objects within the storage. This mechanism can be used for data colocation as well as optimizing latency for different data access paterns.
- an Object-Relational Mapping (ORM) for accessing the stored data

1.3 Delimitations

The data model provides operations on complex structured data, but incurs several operations from the underlying CATS data store in order to accomplish each of its own operations. CATS provides guarantees of strong consistency for each of its operations, but in order to provide same consistency for the data model operations, all the CATS operations necessary to perfom one of our data model operations would need to be executed within a transaction. There is ongoing work at SICS on transactions, so our data model will assume the existence of such a mechanism, but until transactions are fully integrated into CATS, the data model will not be able

to provide any consistency guarantees for the data.

In order to provide more complex operations on top of CATS, we have to choose a proper key structure for the data and make full use of the operations provided by the CATS interface. This means that we will be grouping similar data together and will be using key-value range queries using key prefixes to retrieve it when needed. This grouping of similar data will lead to imbalances in volume and number of operations for certain ranges in the key space. Without a load balancer there will be certain hot ranges in the key space which will hurt the performance of the operations. However, there is ongoing work at SICS on load balancing and its integration with the data model will solve the hot ranges problem.

1.4 Thesis Outline

Chapter 1: Introduction Offers a brief introduction to the field and motivation for this work.

Chapter 2: Background Presents a set of notions necessary to understand this work.

Chapter 3: Related work A brief presentation of related work that has been taken in consideration during the design of the data model.

Chapter 4: Data Model Design Describes the design of the data model

Chapter 5: Implementation Presents the state of the data model prototype that was implemented in order to evaluate the feasibility of our data model

Chapter 6: Evaluation Describes a set of experiments, performed in order to evaluate the overhead imposed by our data model layer.

Chapter 7: Conclusion Includes some final remarks as well as ideas for future work.

Chapter 2

Background

This chapter presents a brief introduction to NoSQL databases and describes some data processing frameworks. The prototype data model is implemented in an event-driven component model called Kompics[14] so a brief description on the Kompics concepts is given in one of the sections of this chapter. The implemented data model is built on top of the CATS[13, 14] key-value store and one of the background sections will briefly present this system.

2.1 NoSQL databases

As a response to the perceived weaknesses of relational databases, a new breed of databases emerged, called NoSQL. These databases do not come to completely replace and make relational databases obsolete, but instead target specific cases and try to optimize their performance for those cases.

Based on their data models, the main NoSQL types of databases are:

Key-value stores Data stores included in this family have a simple data model that allows clients to put and request values per key. The values are stored as uninterpreted blobs of data and there is no support for ad-hoc interactive querying or data warehouse analytics jobs. Many of the existing key-value stores have been heavily influenced by Amazon's key-value store: Dynamo[3].

Column oriented stores These types of storage solutions save and process data by column instead of row and a table can be seen as a sparse, distributed, persistent, multi-dimensional map. Bigtable[1] and Cassandra[2] are some representative solutions for this family of data stores.

Document databases These data stores can be seen as an enhanced version of key-value stores, allowing nested values of semi-structured data. The stored values are schema free, meaning they do not follow a fixed structure, but the store is aware of their structure allowing for more efficient search that

can return just the parts of the data that is of interest. MongoDB[4] is a well-known document database.

Graph databases Based on graph theory, these data stores persist objects and relations as nodes and edges of a graph.

Scalability is addressed by building a distributed system and spreading the load over different machines. In order to be able to support the massive volume of data and traffic, a large number of machines are necessary. The developers of these new distributed data stores chose to use cheap commodity computers and make use of redundancy to cover for the possibility of failure: data and services are both replicated. Having distributed data stores brings forth a new set of challenges: data consistency, availability of the provided service and tolerance to network partitions. Every data store struggles to offer the best guarantees by sarcifing as little as possible on the performance. These problems were well described by Eric Brewer in the consistency, availability and partition tolerance (CAP) theorem[16].

2.1.1 CAP Theorem

The CAP acronym stands for:

Consistency refers to the state of the system after an operation on a key. To be more precise, consistency deals with how a user sees the results of concurrent reads and writes on a specific key.

Availability of the system refers to its ability to serve requests comming from users even in the face of partial failures. With systems built on large numbers of commodity machines, failures of machines are expected and the system is expected to continue to operate smoothly even when a machine that is somehow related to an operation crashes.

Partition tolerance stands for the property of the system to continue to operate even in the face of network partitions. With distributed systems network failure can happen and nodes or groups of nodes (machines) can become temporarily isolated. The system is expected to behave smoothly in case of nodes joining and leaving the system.

The CAP theorem says that a distributed system can only have two of the above properties and since network partitions are bound to happen, partition tolerance is a must for any such distributed system. This leaves the designers of new distributed data stores to choose between availability and consistency.

2.1.2 Consistency

With distributed systems, where the failure of a node is bound to happen, data needs to be replicated, and the replicas need to agree on the order of operations on

each key. In the case of key-value stores, each value is stored in a register which can provide different levels of consistency:

Sequential – for registers abiding sequential consistency, the operations on each replica appear in a sequential, local order. Read operations on a certain replica will return values that appear in a sequential order on this specific replica.

Atomic – for atomic registers, the operations on each replica appear in a sequential, global order. This means that read operations on any replicas will return values from a globally observable sequence.

With regards to visibility of read and write operations, data stores follow two main consistency models:

Strong consistency – "All read operations must return data from the latest completed write operation, regardless of which replica the operations went to"[17]. According to the CAP theorem, if such a property is to be guaranteed by the data store, then availability has to be sacrificed.

Eventual consistency – "In a steady state, the system will eventually return the last written value"[17]. This means that a user might get an older value if the system did not have the time to propagate the change to all replicas. This model might lead to divergent versions of the same data, and the user might need to decide which version is correct. Eventual consistency can be used if the data store is to provide the availability property.

2.2 Kompics

Kompics[14] is a concurrent, message-passing component model that facilitates the development of distributed systems. Kompics is important for our thesis since both the implementation of the data model and the CATS key-value store are written using this framework.

The Kompics abstraction that encapsulates the logic of a protocol is called *component*. These components are event-driven state machines that execute concurrently with other components and communicate asynchronously by message passing. The messages are encapsulated in *events* which are passive and immutable typed objects. Each component exposes the events it can process or emit through an interface called *port*. Two complementary ports of different components are connected through an abstraction called *channel*. Channels forward events in both directions in FIFO order. In order to forward only particular events through the channel, *event filters* can be set for each direction. *Event handlers* are procedures of a component that are triggered reactively when events of specific types are received by the component. A component subscribes its event handlers to the ports it requires.

The Kompics framework also offers a great environment for testing and debugging through its simulation support.

2.3 CATS

CATS[13, 14] is a distributed key-value store implemented using the Kompics framework. It follows a Chord[18] like behavior, using successor list replication and periodic stabilization in a ring based topology. Thus, each node in the ring is responsible for the keys between its predecessor's id and its own id and each of these ranges is replicated on a number of nodes that form a replication group. Each node is responsible for a range and replicates a number of predecessor ranges.

The ABD[19] algorithm can guarantee linearizability of data in static replication groups. In the case of churn, where nodes may leave or join a replication group, concurrent operations might contact non-overlapping quorums, thus breaking linearizability. CATS solves this problem by employing consistent quorums which make sure that every member of the group shares the same view of the group as everyone else. This means that for each key, at a certain time, the replication group is well defined and only overlapping quorums can be formed.

Originally CATS offered only in-memory storage for the data, but has recently been extended with persistent support[20]. CATS offers a simple interface consisting of key-value put and get operations and recent work has extended it with rangequery support[15].

Chapter 3

Related Work

With NoSQL databases seeing a lot of attention, work has been done to offer a more familiar interface and data model, with some stores providing now, a semi-relational model. Striving to offer every bit of performance, one topic drew our attention and that is improving the data layout in order to enhance operation performance. Storing Big Data is not quite enough and there is a need for data processing tools in the form of both analytics as well as interactive ad-hoc querying.

3.1 BigTable, Megastore and Spanner

BigTable[1] is Google's initial data store. It is designed to store large amounts of data, up to petabytes in volume, on thousands of commodity machines. BigTable is a column oriented data store and its data model can be seen as a sparse, distributed, persistent multi-dimensional sorted map. Providing great performance, BigTable saw a wide acceptance inside Google, with many projects using it as their storage layer. However, the different interactive services that started using BigTable, needed a more complex API to work with, similar to the relational model, and so Megastore[11] came to light. Megastore is built as a layer on top of BigTable, blending the NoSQL scalability with the convenience of traditional relational database management system (RDBMS). It provides a semi-relational data model and by the use of synchronous replication it offers atomicity, consistency, isolation and durability properties (ACID) semantics within partitions of data. Within the Spanner[12] paper, the authors state that many applications within Google started using Megastore, due to its richer and more familiar interface, despite the price it paid in performance. Seeing the wide acceptance of a semi-relational model, Google's newest data store, Spanner, also provides this semi-relational data model. Spanner is a scalable, globally distributed data store managing cross-datacenter replicated data. This new data store introduces externally-consistent distributed transaction by the use of their novel clock uncertainty interface. Both Megastore and Spanner show interest in changing the layout of data in order to improve performance, by colocating data that is used together. Megastore introduces the notion of *Entity Groups*

and Spanner that of *Directories* in order to group data commonly used together.

3.2 Walnut

Walnut[21] is an object store developed by Yahoo! as an underlying storage layer for its application. The interesting idea in PNUTS is that it strives to provide support for all ranges of data from small semi-structured data or medium sized objects to large files used by analytical. In order to accommodate both little and large data, Walnut cannot use the typical write ahead log strategy used for small objects or the in place data update used for large objects and uses a custom hybrid protocol based on Paxos[22].

3.3 MapReduce

Being able to store petabytes of data is not the same with processing it and extracting useful information. Many of the presented data stores are built with scalability in mind, offering enough space to accommodate BigData, but with their limited application programming interface (API) it is quite hard to process this data. The traditional relational databases offer tools for ad-hoc interactive queries - online transaction processing (OLTP) as well as analytics support - online analytical processing (OLAP) and the NoSQL community is working towards offering similar functionality for processing data.

MapReduce[5] comes to offer support for processing large amounts of static data. It offers the user the possibility to write *map* and *reduce* functions in order to process the data, while the MapReduce framework deals with the parallelization and monitoring of workers performing these functions over large clusters of commodity machines. The programmers need no experience with parallel computing or distributed systems, since the MapReduce will take care of partitioning the data, scheduling workers on different machines and handling machine failures.

3.4 PigLatin and HiveQL

The map-reduce interface is seen as being too rigid and solutions like Hive[7] or PigLatin[6] come to offer a more expressive language on top of MapReduce. PigLatin is a language designed by the Yahoo! engineers to work on top of Hadoop[8] which is an open source, map-reduce implementation, As the authors describe it, PigLatin is a language striving to fit in a sweet spot between the declarative style of SQL and the procedural style of MapReduce. Hive is a data warehouse solution also implemented on top of Hadoop, by the Facebook data infrastructure team. It offers a SQL like declarative language called HiveQL. With MapReduce offering the support for data processing over large clusters, more expressive languages like PigLatin or HiveQL offer a friendlier interface for users to write their queries.

3.5 Dremel

Having data stored in columnar format and with MapReduce not optimized for this kind of data, Google designed a new tool for ad hoc queries on top of read only data stored in columnar format, called Dremel[10]. Dremel is able to run aggregation queries on top of large data using multilevel execution trees and a novel columnar storage representation.

3.6 Dryad

Microsoft approach to parallel and distributed processing of data is the Dryad[9] system. Dryad provides a data-parallel execution engine where users can provide their own code for processing data as vertices in a dataflow graph. It is seen as an alternative to MapReduce, offering more flexibility to the user in exchange for sacrificing a bit on the simplicity of the framework architecture.

Chapter 4

Data Model Design

The data model we design on top of CATS is a schema based, row oriented data model. The information schema contains the type definition for the saved data and provides a mapping from human-readable database and type names to efficient key-encodings for those names. Users can define databases as a private scope for their data. Databases group together data and metadata used for searching and quering this data. The data itself is stored in the form of objects with fields and each such object needs to have a type definition and a unique identifier within its type. Type definitions describe the inner structure of an object and for the moment are fixed, in the sense that new fields cannot be added after the creation of this type definition. To facilitate search based on the values of different fields of the objects, we support secondary indexes. Thus objects can be searched either by using the primary key or the secondary indexes. The secondary indexes are also a key part of the querying capability of our data model.

4.1 CATS API

Since our data model is built on top of the CATS key-value store, our operations have to be designed to work on top of the provided API. CATS, interface includes three operations:

- put(key, value)
- get(key) : value
- rangequery(range[1], limit) : value1, value2, ..., valueN

In order to offer more complex operations on top of CATS it is necessary to store additional meta-information into the store. This extra information can be stored in two ways:

- by storing additional key-value pairs

[1] A range is defined based on two keys: startKey and endKey and two booleans startInclusive and endInclusive

- by embedding the information we want to store into the key structure

The rangequery operation is built on top of the get and put operations and is quite expensive, thus, it is generally avoided and simple get operations are preferred instead. However, in the cases that the exact value of the keys is unknown, but we know in which range it resides, we have to use the rangequery. In order to try and optimise the performance of the rangequery, we need to define the keyspace in such a way, that all of the ranges that are of interest, contain as little data as possible. Rangequery is already an expensive operation, so we do not want to increase its latency by making it retrieve more keys than necessary. One way to obtain this is to make sure that we can define a range that contains only the data of interest to us, without overlapping with other such ranges. We can guarantee this property for ranges, by having keys that follow a strict structure, and use the common prefixes of the keys to define the range.

4.2 Data model abstractions

Most of the abstractions described in this chapter have an identifier that is unique among the same class of abstractions, but need some other information in the form of parent abstraction identifiers in order to uniquely identify them in the whole keyspace. The main abstractions used throughout our data model along with their identifiers are displayed in Table 4.1.

Table 4.1. Data model abstractions and identifiers

Abstraction	**Identifier**
database	databaseId
type definition	typeId
object	objectId
index definition	indexId

4.2.1 Databases, type definitions, objects and fields

Databases are the main containers of data in our data model and split the keyspace into well defined ranges. Users define these containers in order to group specific data and metadata in one of these specific ranges of the keyspace. Databases contain data as well as metadata in the form of type definitions, object state and secondary indexes.
The means of saving data through our data model is through user defined objects. Objects have a strict structure which is described by a type definition. Since type definitions are defined within the context of a database, a type definition is perfectly identified only by the tuple *(databaseId, typeId)* whereas an object can be uniquely identified by the tuple *(databaseId, typeId, objectId)*. From the user's perspective,

an object is a collection of fields and the type definition clearly reflects this by containing field definition tuples such as *(fieldId, fieldName, fieldType)*. The *field identifier* is automatically assigned by our data model and the user is unaware of it. The field identifier is used as a mapping between the human readable fieldName and a more eficient encoding used within the keys. On the other hand *fieldName* and *fieldType* are explicitly provided by the user upon each field definition. The fields themselves can be objects or primitive types: boolean, integer, long, float, double, string or byte array.
Our data model supports secondary indexes for object fields as well, and the index definition is also stored within the type definition range of the keyspace as it will be later described in the keyspace section.

In order to store these abstractions in CATS, we need to define the keys in a way that uniquely identifies each abstractions within the whole CATS keyspace. So far the keys for the following abstractions have to include the specified identifiers:

- database - perfectly identified by ($databaseId$)
- type definition - perfectly identified by ($databaseId, typeId$)
- object - perfectly identified by ($databaseId, typeId, objectId$)

However, as it will become clear in the following sections the keys need additional information in order to optimise the use of the CATS operations and to support additional operations.

4.2.2 Secondary indexes

The retrieval of objects using their identifiers is not always useful, especially when there is a large number of them and we are not in the possession of the identifier of the object we want. One potential way to find an object when the value of a subset of its fields is known is to scan through the whole database and compare each of the retrieved object with the targeted values. Such an operation is of course inefficient as we probably need to scan a large number of objects in order to find what we search for. The second option is to use secondary indexes by introducing additional information that could speed up this search. This means we have to trade one cost for another - we pay a price in extra space needed to store the secondary indexes while decreasing search latency costs. Secondary indexes are typically defined within the context of a type definition, so the tuple *(databaseId, typeId, indexId)* perfectly defines an index. Our data model supports single field indexes. We will refer to the tuple *(databaseId, typeId, indexId, valueId)* as an *indexValue*. An *indexValue* identifies a collection of objects that satisfy the value of the index.

In CATS, we can store secondary indexes in two ways:

a) Store the *indexValue* in a single <key,value>, with the key being defined using *(databaseId, typeId, indexId, valueId)* and value=$\{objectId_{i1}, objectId_{i2}, ...objectId_{ik}\}$. This means that each time we search using the secondary indexes we perform

one get operation in CATS, while when we need to put an object, we have to read and modify the value by adding the new *objectId* to the collection. This means that we pay a higher price when we store an object, but a lower price upon retrieving it.

b) Store the *indexValue* as a collection of <key, value> having the following key definition *(databaseId, typeId, valueId, objectId)* and value at the moment can be empty, but later can be used to store other information. This means that each time we search using secondary indexes we perform a CATS rangequery with a range defined on *(databaseId, typeId, valueId)*, but when we want to put an object to storage, we do not need to get and modify the indexValue since we can simply perform directly the put operation with the key defined as such *(databaseId, typeId, valueId, objectId)*

The two strategies trade cost in latency between storing and searching objects using secondary indexes. In our model we chose the second strategy in order to keep a low latency for the object put operation.

With a carefull choice of secondary index keys, we can use rangesqueries to retrieve object identifiers that belong to the searched objects based on key prefixes. Secondary index keys for objects of same value have to be defined such that the prefix contains *(databaseId, typeId, valueId)* and can clearly identify the range for the rangequery. This means that we can use rangequeries to do searches similar to the following: if we defined an object called "car" that has a production year and we wish to search for all cars produced in the year "2000" the above prefix will allow us to retrieve all the object ids with one rangequery. However if want to allow for searches that include operators like "<, <=. >, >=" we have to make sure that our prefixes for the secondary keys still allow the use of one ranagequery to retrieve all of the object identifiers that are within the range defined by these constraints. Ranges used for secondary indexes will thus, be defined based on a start prefix:*(databseId,typeId,valueIdStart)* and and end prefix:*(databaseId,typeId, valueIdEnd)*.

4.3 Data model operations

The designed data model supports four types of operations depending on the data model abstraction they operate: database, type definition, index and object

Database operations
The data model provides operations for creating and dropping databases:

- createDatabase(databaseId)
- dropDatabase(databaseId)

Type operations
These operations of the data model operate on the type definition metadata and

allow the user to create type definitions, to retrieve a certain type definition or to retrieve all the type definitions within the context of a database:

- putType(databaseId, typeName, typeDefinition)
- getType(databaseId, typeId)
- getAllTypes(databaseId)
- deleteType(dbId typeId)

Index operations
These operations deal with the index definition and deletion for a specific type and add metadata to the type definition, so that when objects are put, the respective secondary indexes are also updated:

- createIndex(databaseId, typeId, indexName, indexDefinition)
- dropIndex(database, typeId, indexId)

Object operations
This set of operations include one way for storing data, through the putObject operation and three different ways to retrieve data out of the CATS key-value store:

- putObject(databaseId, typeId, object)
- getObject(databaseId, typeId, objectId)
- query(databaseId, typeId, queryCommand)
- scan(databaseId, typeId, objectIdRange)

4.4 Defining the keyspace

As described previously, information can be embedded even in the key structure. For some of the operations we need a key structure that ensures an optimal use of rangequeries by being able to clearly define a certain range and guarantee that ranges of interest do not overlap.

The following limitations were regarded before designing the key structure:

- each database needs to have it's own range and no two databases should overlap.
- each of the following: type definitions, object state, secondary indexes and data objects have their own well defined ranges that do not overlap, according to 4.1.
- type name listings need to be stored in sequential order and with no other data interleaved in order to optimise the use of rangequerie to perform the getAllTypes operation.
- secondary indexes need to be stored in sequential order and with no other data interleaving in order to optimise the rangequery used for searching objects.
- data objects need to be stored in sequential order in order to be able to perform optimised scans of objects.

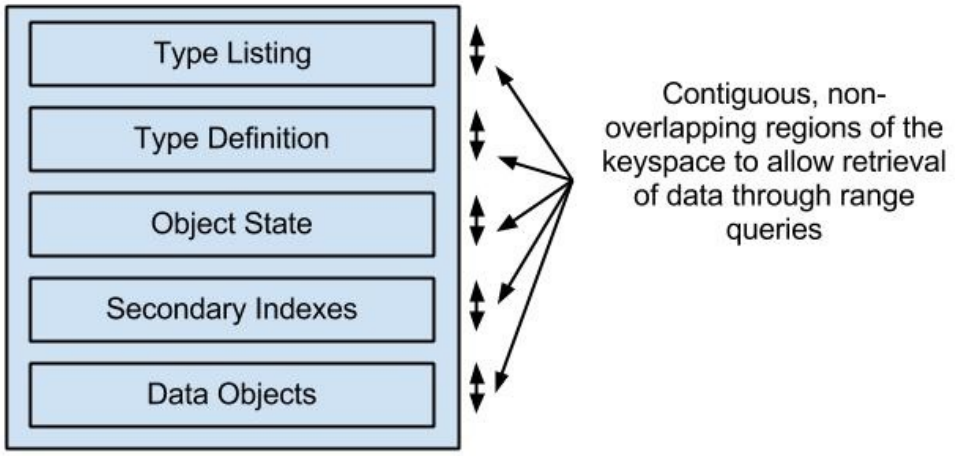

Figure 4.1. Types of data stored in a database

With the aforementioned limitations in mind we designed a key structure that avoids unwanted overlappings for optimal performance. The format employed is decimal path separated as displayed in Table 4.2. Using this format, we can use the prefixes of keys to define the specific ranges needed for the operations of our data model.

Table 4.2. Key structure per data type

Data Type	**Key Format**
type name listing	databaseId.typeListingFlag.typeId
type definition	databaseId.typeDefinitionFlag.typeId
object state	databaseId.objectStateFlag.typeId.objectId
data object	atabaseId.dataObjectFlag.typeId.objectId
secondary index	databaseId.indexFlag.typeId.indexId.indexValue.objectId

4.5 Expansion and compaction of objects

In the presented data model, objects are the basic unit of storage. Each object has a type definition and belongs to a database. Each object is practically a container of fields and each field itself can be another object or a primitive type. Since the underlaying storage is a key-value store, the data has to be partitioned and stored in a key-value manner. With this underlying support, objects can be saved as one

<key, value> where the value is the serialised form of the object, or as a collection of <key,value> where each value is the serialised form of one of the fields of the object. This allows us to optimise the layout of the stored data for different operations.

Choosing either of these strategies depends on several factors, such as:

- what is the performance of our underlying layer when it comes to getting one big object compared to a range query retrieving a collection of smaller value
- how do we access the data in the object - do we usually update/retrieve the whole object or do we update/retrieve subsets of the fields of the objects
- are some of the fields of the object updated with different frequency

Storing the whole object as one blob of data, means we can you use the cheaper get operation of CATS to retrieve the object, but we cannot access its individual inner fields without reading the whole object. Furthermore, when we have to modify one of the fields we need to modify the whole object altogether.
Storing the object as a collection of fields, means we need to use the more expensive range query operation to retrieve the whole object, but we can also read/update individual fields without having to read/update the whole object.

A combination of both strategies is possible at the cost of storing additional metadata - object state. From here forward we will say that storing the whole object as one blob of data means the object is in compacted state. On the other hand, storing the object as a collection of fields means the object is in expanded state. Having objects as fields of objects means that the expansion can go to arbitrary depth as in figure 4.2. Different fields can themselves be saved in compacted or expanded mode. If the parent object is in Compacted state, there is no need to store any object state metadata for the children, as we know that they are nested in the parent object. If the parent is in Expanded state, we need to store the object state metadata for each of the object's fields that are stored outside its parent.

The keys for saving the object on disk need to be structured accordingly in order to clearly show the parent hierarchy and to offer the ability to define a clear interval (a subspace of the key space) for reading the whole content of that container using a rangequery. The expansion of containers, will, of course, turn any get operation on that container into a rangequery.
When we save the object expanded, we have to make sure that all the object field keys contain the objectRootId so that we know that they are part of another container. In order to have this, the keys need to be adjusted, in particular, the objectNrId, from the key, now becomes: <objectRootId.objectChildId.objectChildId...> so the data object key now becomes <databaseId.dataObjectFlag.typeId.objectRootId.objectChildId.objectChildId...>. As we can see the keys can become quite large, so this fact also has to be taken into consideration when expanding objects.

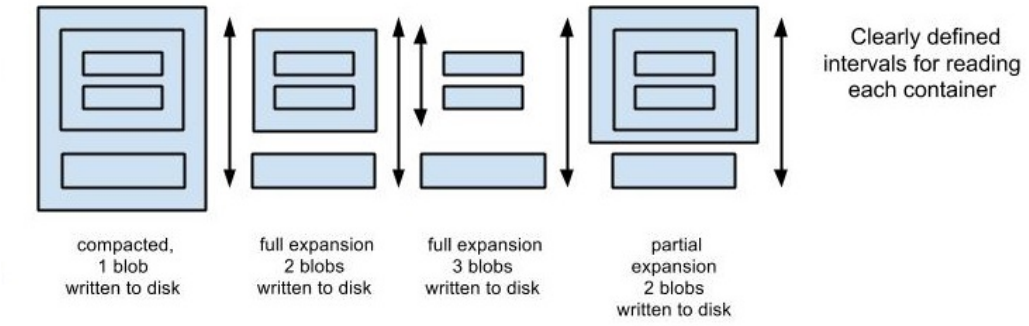

Figure 4.2. Expansion of objects

4.6 Improving update latency of objects

Having objects saved in storage in compacted mode, means that if we want to update an inner field, we would need to write the whole object again. With large objects, this means the client needs to send the whole object that it wants to change to the server, or the server has to read the object from disk, change the subpart of it and write it back to disk.

In order to fix this problem, we introduce a new option, that of *forced expansion*. If the object is in compacted state and we have to update one of its inner fields, we will *force* the object in expanded state, meaning we will write the sub-fields directly on disk, without changing the old value, but we will take this forced expansion in consideration when we do a get operation or when we do a normal expansion.
Let's assume we want to modify an object field, *fieldK*, which has as parent object *obj1* and two sub-fields *fieldK2* and *fieldK3* at the same level, namely *fieldK.fieldK1* and *fieldK.fieldK2* respectively. In order to save *fieldK* to storage, we will first need to read the state of obj1 (a get operation) and the state of *fieldK* and its children(a rangequery operation). In order to do the update on *fieldK* we need to check the state of that field and its parent obj1. The following cases can be distinguished:

1. **obj1's state is compacted** - figure 4.3. If the parent object is compacted, it means that the entire object, with all its fields, is curently saved to disk as one blob of data. Since we want to speed up the put operation of this inner field, we will forcely expand the parent object and save the inner field separately on disk. This means we need to perform three parallel put operations for saving the inner field data and changing the state of both the parent object and the updated field:

    i. put state of obj as expanded

Figure 4.3. Update when parent object is in compacted state

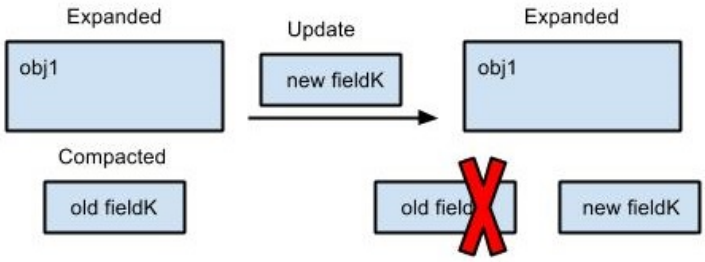

Figure 4.4. Update when parent object is in expanded state and the updated field is compacted, outside parent

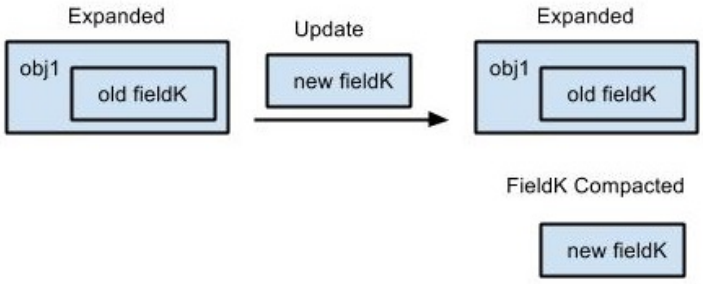

Figure 4.5. Update when parent object is in expanded state, and the updated field is inside parent

ii. put state of fieldK as compacted

iii. put fieldK

2. obj1's state is expanded, fieldK's state is compacted, outside the parent object - figure 4.4. Since we read the state of fieldK as beeing compacted, it means that it is outside the parent, stored as one blob of data, and we can simply

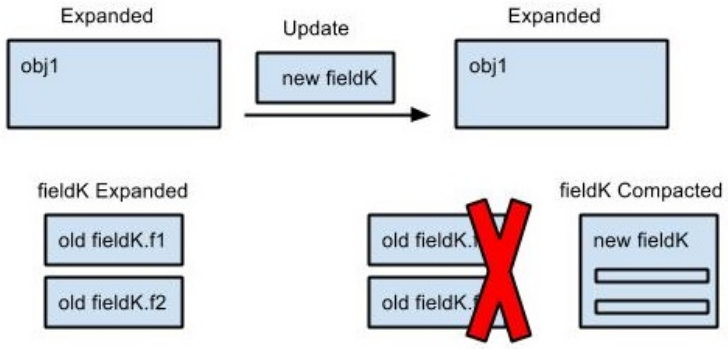

Figure 4.6. Update when the updated field is expanded, outside parent

overwrite its value with the new value we received. In this case we do one put only for storing the update field. We do not need to modify any of the states

3. **obj1's state is expanded and fieldK is inside the parent object** - figure 4.5. We detect this state, by reading the parent's object as expanded and no state for the inner field we want to update. Since we want to lower the latency of this put operation, we will forcely pull out the inner field from the parent object, by simply saving the inner field's value to disk and also changing its state to compaacted. We do not need to change the state of the parent object since it is already expanded. In this case, we need to do two put operations:
   i. put state of fieldK as compacted
   ii. put fieldK
4. **fieldK is expanded** - figure 4.6. This is the slowest case as we need to delete old values, besides saving the new ones. With the update field expanded, we need to delete its expanded sub-fields and then overwrite its value and also change its state to compacted. In this case we need to do the following operations:
   i. delete state of children objects
   ii. delete children objects
   iii. put fieldK state to compacted
   iv. put fieldK

4.7 Data colocation

Currently objects of the same type are stored continuously, without overlapping with objects of other types in order to have fast performing, cheap scans. However,

for certain operations, objects of different types need to be used in combination by the upper layer application. This involves getting the objects independently, possibly from multiple nodes, with each get operation having its own latency and imposing its own bandwidth usage. To improve on both latency and bandwidth usage, Megastore and Spanner propose to store together data that is commonly used together. This involves grouping objects of different types, or in relational terminology merging rows from different tables as in figure 4.7

In order to achieve this colocation of data, we use our compaction/expansion mechanism: we force one object to become a field of another and then we expand that object. This means that we need to introduce this forced child field into the parent type definition and also introduce a new piece of information, an indirection from the original key <databaseId.dataObjectFlag.typeId.objectId> to the new key <databaseId.dataObjectFlag. parentTypeId.parentObjectId.objectId>.

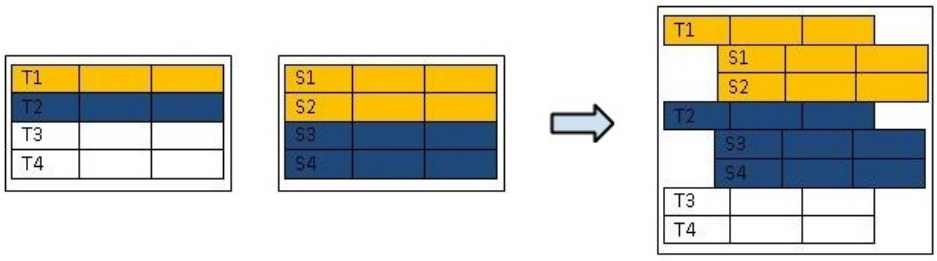

Figure 4.7. Data colocation

4.8 Denormalized Data

Denormalization can be defined as duplicating data in several places, besides its normal location, in order to decrease the latency of read operations that require only the piece of denormalized data. This, of course will put additional work onto the put operation, which needs to keep the denormalized data consistent in all the different places. Secondary indexes are introduced to allow the search of objects by field value, but they can also be used to decrease the latency of certain read operations, by saving denormalized data as its value. Since the put operation has to fix the secondary indexes anyway, this denormalization of data does not hold any performance penalty. This way, we can bypass the retrieval of the actual object, and thus reduce the latency of the operation. For example, if we know that we will often search for photos by name and then retrieve their url, we can jump over the retrieval of the whole object and retrieve only the url by denormalizing the object

data and saving the url in the secondary indexes' value.

```
1 class Photo {
2         String id;
3         @Index String name;
4         @Denormalize(``name'') string url;
5         int size;
6         @BigData byte[] photo;
7         GeoTags tags;
8         Date timestamp;
9 }
```

4.9 Object relational mapping

In order to ease the usage of our data model, we design a Java framework, similar to Java Persistence API (JPA). Users familiar with JPA will be able to use our framework to put, get, scan and query for objects. Similar to JPA we support entities, in the form of lightweight Java classes, which represent the actual type definition of the objects stored using this data model.

The users interact with the data model through the EntityManager abstraction. The EntityManager performs the type definition operations automatically, by parsing the entity class into a type definition and saving it to storage.

Currently supported annotations include:

Entity can be applied on Class level. Marks a class as a type definition and allows objects of this particular type to be stored though the EntityManager

Id can be applied at Field level. Marks a field as the object identifier

Index can be applied at Field level. Creates a secondary index on this field

Immutable can be applied at Class or Field level. Marks the class or field as being immutable. No updates of the marked object or object field will be allowed.

BigData can be applied at Field level. Marks the field to be compressed or saved in the file system. No secondary indexes are allowed on this field

Denormalize can be applied at Field level. Takes as argument an indexed field and will duplicate the value of the object's field into the secondary index's value

The EntityManager interface gives access to the following data model object operations:

- getObject(dbName, objectclass)
- putObject(dbName, object)
- scan(dbName, objectclass, range)
- query

The current query interface does not support joins, being able to execute queries restricted to one type only, based on its fields. The query operation is however split among a set of interface methods:

projection Methods defining what fields get returned

1. select(dbName, objectclass) - defines the type of objects to query.
2. select(field) - if no fields are selected, the whole object is returned

selection Methods defining search values for different fields

1. basic operator - operator(value) - includes equal(eq), greater than(gt), greater than or equal(gte), less than(lt), less than or equal(lte)
2. concatenation operator - operator(fieldName) - and, or

An example of such a query definition is the following.

```
1
2  @Entity
3  class Person {
4          @Id private String personnummer;
5          private String name;
6          @Indexed  private int age;
7          ...
8  }
9
10 {
11 ...
12 EntityManage em = EntityManager.getManager();
13 String dbId = ``3'';
14 Set<Person> persons =
15         em.select(dbId, Person.class).
16         where(``age'').lt(18).
17                 or(``age'').gt(20).
18                 and(``occupation'').equal(``student'');
19 ...
20 }
```

Chapter 5

Implementation

In order to be able to evaluate and see if our theoretical approach is viable in practice, we developed a prototype for the designed data model. Our prototype is developed in Java, using the Kompics[14] framework and the CATS[13] key-value store. In our implementation we also use the extending work on CATS from two previous master thesis' providing us with rangequeries[15] and with persistence storage support[20].

5.1 Data Model Layer

The data model is implemented as a DataModel Kompics component and a set of Operations it can perform. The DataModel component is responsible for communicating:

- with the local CATS layer in order to perform operations required by our data model.
- with client, through the network, in order to initiate data model operations

The data model interacts with the CATS layer through the following set of messages:

- key-value get operation - CatsGetRequest and CatsGetResponse
- key-value put operation - CatsPutRequest and CatsPutResponse
- key-value rangequeries - CRQGet.Req and CRQGet.Resp

Operations can be initiated on the data model layer, by either a thin or a thick client. The difference between a thick and a thin client is that the thick client is able to maintain a cache of the type definitions locally in order to speed up operations by parsing and sanity checking objects provided by the user. Also, in order to have a better performance, the data nodes also cache the type definition. Our initial experiments showed that a thin client with data nodes that don't do cacheing of type definition and need to retrieve it on every operation have very low

performance, making them virtually unusable. Because of this we decided to cache the type definitions on both clients and data nodes. The current implementation assumes that once created, type definitions do not change.

5.2 Operations

Operations can be initiated by clients through the following request messages:

- GetType
- PutType
- GetAllTypes
- PutObj
- GetObj
- ScanObj
- QueryObj

All the operation logic is encapsulated in Operation classes. Since our data model operation requires multiple CATS operation done in paralel or sequentially, we have designed each operation to be either a ParallelOperation or a SequentialOperation and have used a composite pattern in order to build more complex operations. Since CATS stores values as opaque strings, each of the operations use custom parsers in order to decode the values returned.

5.3 Identifiers

Choosing identifiers started as the following small problem: should the identifiers be integers, long, string or byte arrays? Since our data model will store a large number of objects and for each object it will store additional information like secondary indexes and object state, making the identifier encoding flexible might help in the future. The decision was made to follow a byte array implementation of identifiers as the one presented by Jeff Dean[23].

An identifier consists of a byte array of variable length prepended by its size. For ease of key encoding, we considered identifiers of maximum 256 bytes, so one unsigned byte will be enough to represent the length of the byte array. So an identifier will vary between 2 bytes and 257 bytes and will follow the format: prefixByteLength | byteArray. Identifiers grow in number of bytes as the identifier spaces defined by smaller byte arrays get filled. So an identifier will start with (byte)0, prepended by (byte)0 for length - smallest identifier = (byte)0.(byte)0. We use the dot notation here as concatenation. when identifier (byte)0.(byte)255 is assigned, the next issued identifier will be (byte)1.byte(0).(byte)0. By prepending the byte array with its length we can use lexicographical comparison of byte arrays in order to compare two identifiers, because only identifiers of same size will get their byte array compared, the rest will be compared based on length.

5.4 Keys

Keys are encoded according to the definition from the design:

type name listing - dbd.tLF[1].typeId

type definition - dbId.tDF[2].typeId

object state - dbId.oSF[3].typeId.objId

data object - dbId.dOF[4].typeId.objectId

secondary indexes - dbId.iF[5].typeId.indexId.indexValue.objectId

5.5 Queries

Our implementation also offers a simple query parser and analyser in the form of a binary tree that contains the *and, or* operators as nodes of the tree and the *eq,gt,gte,lt,lte* as leafs of the tree. We will refer to this tree from now on as *queryTree*. This queryTree offers us two services: to evaluate the tree against a certain object or to retrieve ranges for a certain index. When building the queryTree we use the *and,or* precedence from relational algebra, which means *and* has precedence over the *or*. By using rangequeries and the ranges generated by our query tree, we retrieve a collection of objectIds and then by performing individual get operation we retrieve the actual values of these objects. Before returning the result to the user, we filter the objects through our queryTree to check for constraints on fields that were not indexed.

5.6 Clients

Our implementation offers five different clients:

ConsoleClient A client that can parse strings as input and issue appropriate data model operations

EntityManager A client that that offers a ORM interface and can be used to persist or search for objects

TestConsoleClient A variation of the console client that receives as input predefined queries, runs them through the data model and evaluates the result against the expected one. This client is used for testing the correctness of our implementation

[1] typeListingFlag
[2] typeDefinitionFlag
[3] objectStateFlag
[4] dataObjectFlag
[5] indexFlag

TestEntityManager A variation of the entity manager client that receives as input predefined Entity classes and a testing method containing EntityManager calls and runs them through the data model, checking if the result is the expected one. This client is also used for testing the correctness of the implementation.

YCSBClient A client used for evaluating the performance of our implementation by the use of the well known YCSB[24] tool.

All of the clients rely on a ClientInterface component that provides routing capabilities, by connecting to the CATS bootstrap server and retrieving a set of nodes where requests can be served. In the current CATS implementation the bootstrap server returns all the nodes in the system.

5.7 Limitations of implementation

Our implementation was built having in mind the future addition of transactions. At this moment transaction were not available in CATS, so our data model does not give any consistency guarantees. If we limit our data model to store all object compacted, then reading/writing objects will follow a read-your-writes consistency. However, if we store objects expanded, then at the moment, any two writes that overlap, might corrupt the data as the puts for different fields might interleave. Also if a read and a write overlap, the read might also return a corrupted values. However this limitation was know in advance, but the implementation is structured in such a way to allow easy integration of transactions.

Chapter 6

Evaluation

6.1 Yahoo! Cloud Serving Benchmark (YCSB)

In order to evaluate the latency and throughput of our prototype data model we have used the open source tool YCSB. YCSB is a universely accepted cloud benchmark that can be easily adapted to most database solutions in order to determine their latencies and throughput. YCSB provides a set of "bindings" that are user to interact and evaluate many well know databases like BigTable, Cassandra, MongoDB and others. Additionally, YCSB ofers a mechanism to modify the parameters of the evaluation by the use of workload files.

We created our own binding by simply extending the *com.yahoo.ycsb.DB* class in order to provide an implementation for get, insert, update, and scan operations according to our solution. This implementation combined with the client interface allows us to perform putObject and getObject operation through our data model.

While trying to connect YCSB to our system, we encountered a small problem in the way the tool was generating identifiers for the get and put operation. The tool was prepending the generated long identifiers with the string "user". An identifier of this structure would mean that all operations would end up in a particular subspace of our keyspace, so we decided to remove this prefix form all keys and just use the long part of the identifier.

6.2 Experiment setup

The evaluation of our solution took place on a cluster of seven machines belonging to the SICS. The configuration of these machines consisted of:

- Gigabit Ethernet LAN
- 32 GB of RAM
- two-six-Core AMD Opteron(tm) Processor 2435 CPUs

Out of these seven machines, one machine hosted the YCSB client running 24 threads in paralel and the bootstrap server necessary for starting CATS. The remaining six machines were used for deploying our implementation. Since CATS does not provide load balancing, we had to manually assign the identifiers to the machines in order to balance the data on all of them. Experiments under different type definition configurations ran on the same node id definition in order to be able to correctly compare results. Since in our implementation indexes and data objects are stored in different keyspace regions and since CATS does not provide load balancing, it was necessary to assign two machines dedicated to the index region and four machines to the data region. The load on the two index machines varied depending on the experiment, while the load on the other four machines was relatively constant.

The CATS configuration for running the experiments consisted of:

- data replication factor: 3
- underlying persistence store: LevelDB
- underlying persistence store cache: 16MB

We chose a low cache size in order to make sure that during our experiments our searches also hit disk and not mainly the cache, since it wasn't our intention to test LevelDB's caching performance.

Data Model configuration:

- objects stored in compacted mode
- thick clients that cache the type definition

6.3 Workload setup

We have evaluated our data model by runing 2 workloads under 3 different type definition setups, with 24 threads per client. The type definitions had to be loaded in the database before runing the experiment and they were beeing cached on the client, so practically the type definition was read only for the first operation.
The two workloads varied only in the number of records, one running with 50.000 records and one with 100.000 records. The other workload parameters are:

- readAllFields is set to true
- fieldLength=100
- fieldCount=10
- operationcount set to 200.000
- readproportion=0.5 - proportion from operationcount
- updateproportion=0.5 - proportion from operationcount
- requestdistribution=uniform

Under this workload, a record translates to an object containinig 10 string fields, each of 100 chars long, meaning the total size of the object will be roughly 1kB if no indexes are involved. Since the field values are 100bytes, it means that for each index, roughly 120 extra bytes per object need to be stored.

The three type definition we used varried in number of indexes:

no index - we store only the object in the database, so 1kB per object

2 fields indexed - we store an object of 1kB and an additional 240bytes for the secondary indexes

4 field indexed - we store an object of 1kb and and an additional 480bytes for the secondary indexes(roughly half the size of the object)

With a replication of 3 and 100.000 records of 1kB, we store roughly 300MB of data. As for index data, since a secondary index key is roughly 120bytes, for the same number of records with the same replication factor we add an additional 40MB per index. In our case we add 80MB for two indexes, respectively 160MB for four indexes. Since each machine has a cache of 16MB, our cumulated cache for the six machines running our implementation is 96MB.

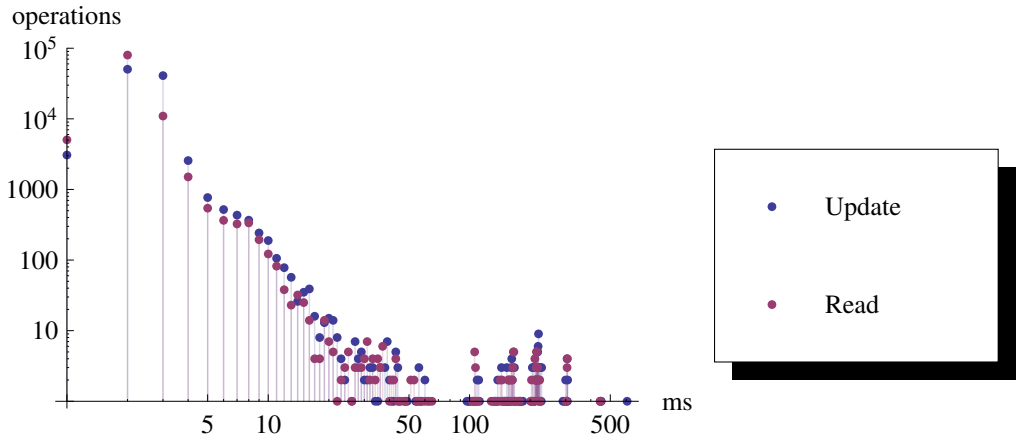

Figure 6.1. Latency - 200.000 operations, 50.000 objects, no indexed fields

6.4 Experiment 1

In this experiment we ran the evaluation with 50.000 objects and the three type definition setups and with objects stored in compacted state in order to evaluate the overhead of maintaining indexes during put/get object operations.

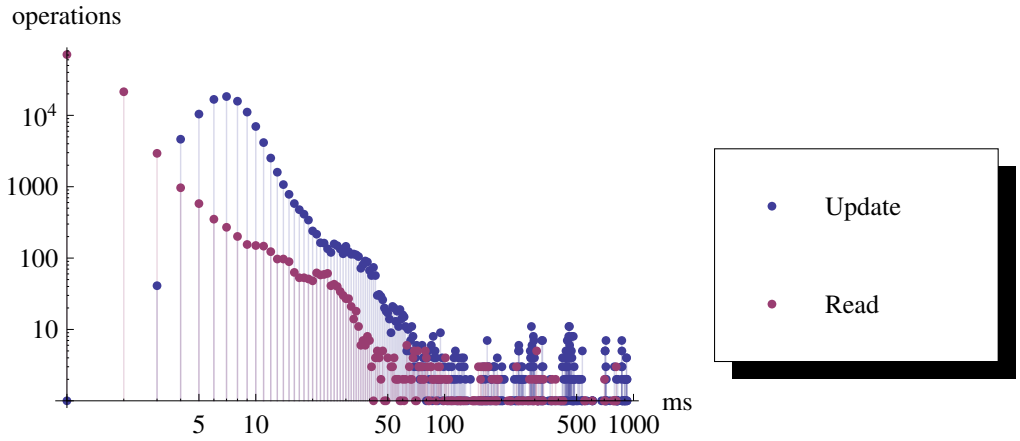

Figure 6.2. Latency - 200.000 operations, 50.000 objects, two indexed fields

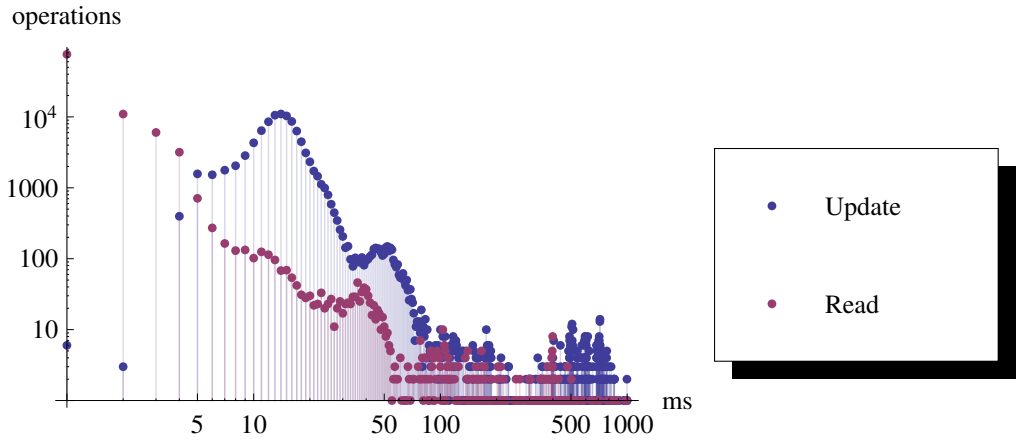

Figure 6.3. Latency - 200.000 operations, 50.000 objects, four indexed fields

In the case of the type definition with no indexes the put and get operations on objects were conceptually almost the same as CATS put/get operation, with a small overhead for parsing objects on the client side. As we can see in figure 6.1 during this kind of load, the performance is very good, and we only have a small tail. The figure presents on the x-axis the latency in miliseconds and on the y-axis the number of operations that finished with the respective latency. The update 95th percentile latency was 4ms and read 95th percentile latency was 3ms. The throughput provided by YCSB is of 6900 operations per second.

Figure 6.2 presents the latencies for running the same workload but with a type definition that contained two indexes. The performance dropped and we start seeing more requests that had a big delay. For this run YCSB reported an update

95th percentile of 17ms and a throughput of 2700 operations per second. As we can see maintaining indexes adds quite some overhead and the throughput drops to less than half compared to the case when we have no indexes.

When running the same workload with a type definition containing 4 indexes, as in figure 6.3, the system becomes slower and the update latency drops, with a 95th percentile at 40ms. The throughput also drops to arounds 1400 operations per second.

In all cases however, most of the operations finish successfully, with less than 0.05% of the operations timming out. Even in the case with fours large secondary indexes, we still see a 95th percentile update of 40 ms, which could probably be acceptable in a real system.

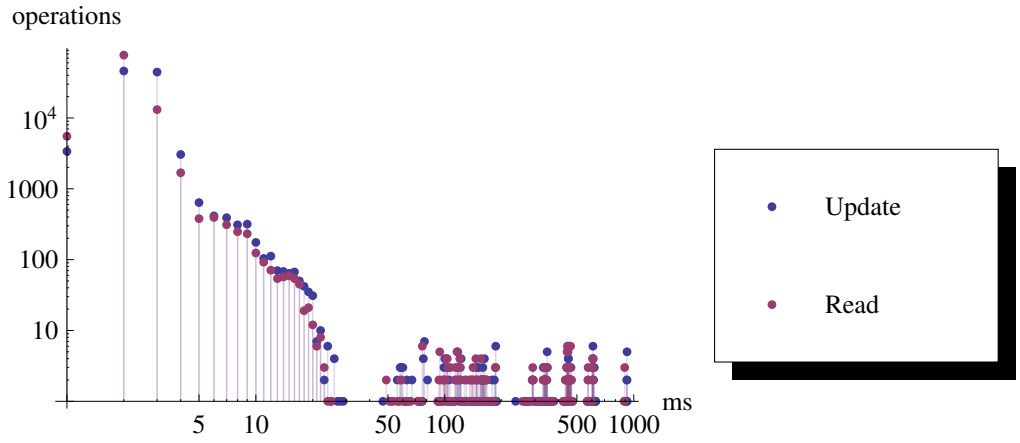

Figure 6.4. Latency - 200.000 operations, 100.000 objects, no indexed fields

6.5 Experiment 2

In order to see if increasing the number of records affects the performance of maintaing indexes, we ran the same experiments with 100.000 records, doubling the previous volume of data.

As we can see in figure 6.4, the case when we have no indexes seems quite unaffected by this increase, with update 95th percentile latency of 4 ms and a read 95th percentile latency of 3ms and a throughput of 5700 operations.

Figure 6.5 also show a slight decrease in performance compared to the experiment1, where we ran the evaluation with 50.000 records, and the update 95th percentile latency gets to 22 ms while the throughput gets to around 1900 opera-

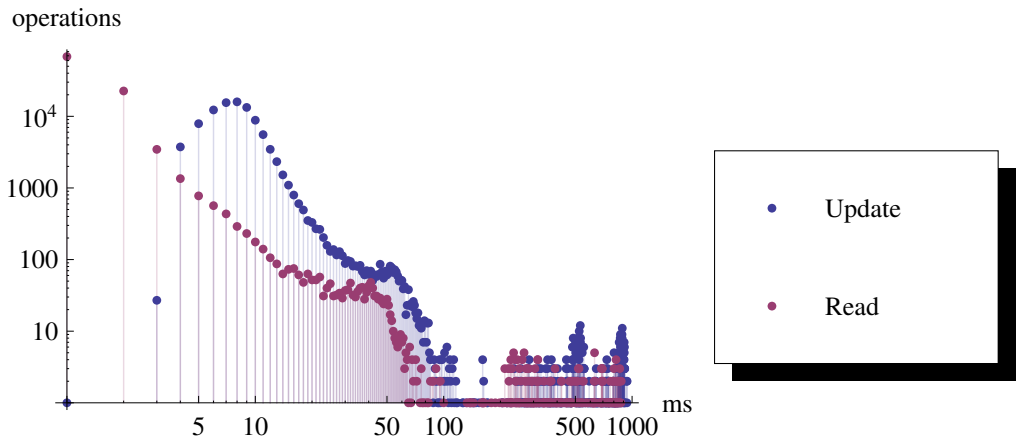

Figure 6.5. Latency - 200.000 operations, 100.000 objects, two indexed fields

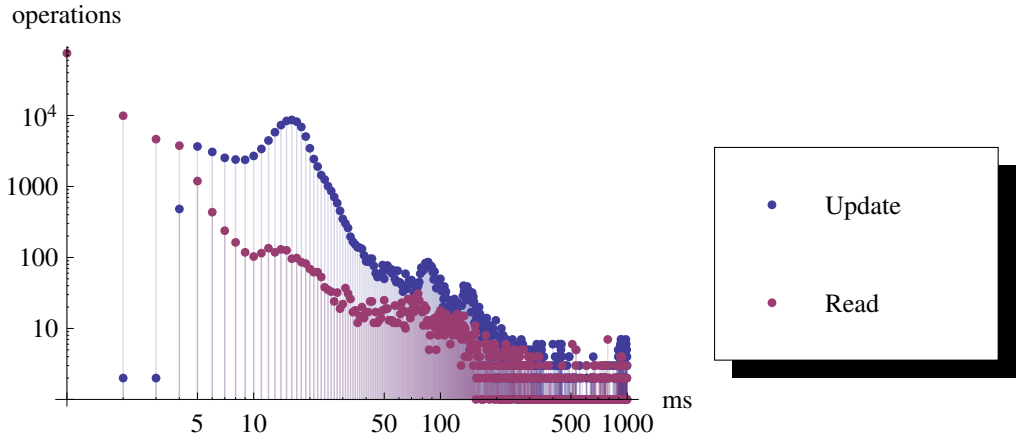

Figure 6.6. Latency - 200.000 operations, 100.000 objects, four indexed fields

tions per second.

However, in the case of four 4 indexes, figure 6.6, the update 95th percentile latency increases to 100ms and the throughput drops to around 840 operations per seconds however, even now almost all operation finish on first try, with only around 0.1% of them timming out at 1000ms

6.6 Discussion

The first problem confirmed by our experiments is the need for load balancing the system. One reason for the decrease in performance with more indexes is the

bottleneck that are the index servers themselves. Secondary index keys are smaller then the object they index and in our situation a lot more. Trying to balance the volum on data, we assigned two machines out of the total of six to indexes. However, in sheer number of operations, maintaining the indexes can produce quite a high volume of CATS put operations. In case of four indexes, for each put we might have to do up to four deletions and four puts, when the indexed fields have new values. In the case of YCSB, with randomly generated values, it is more than lickely that all of the indexes had to be fixed at each operation.
The experiment results reinforce our design approach to try and lower the cost on puts, by expanding the object on put operations, and delaying the cost onto future read operations.

Chapter 7

Conclusion

Many services and applications struggle to store and maintain Big Data, but also try to make sense of this data and extract usefull information. An interface that allows to just store and get one piece of data is of limited use. With more distributed storage solutions beeing developed, it has reached a point where the sheer performance of such a system does not guarantee large scale adoption. Sometimes slightly slower systems might get wider acceptance due to a more friendly API. The goal of this thesis has been to develop a richer, more friendly API on top of a distributed storage system, in particular the CATS key-value store and to evaluate the cost in performance that we have to pay. Evaluation has shown that there is much work to do to optimize even the basic object operations if we want to be competitive. With latency increase for object operations of around 20 times compared to the CATS key-value operations, there is a dire need of optimization in order for our data model to become competitive. The design of our data model, already proposes possible solutions to lessen the performance loss that our protype seems to suffer of. A mechanism to adapt the way objects are stored can move latency cost from one operation to another depending on the system requirements. We can make further use of colocating and denormalizing data in order to improve the performance of our prototype.

7.1 Future work

The prototype implementation and evaluation should be just a first step towards implementing the full data model that can perform interactive data processing on BigData. The design chapter already describes several features to be implemented, but there are also a number of more complex ideas that require aditional research work.

Distributed transactions - Our data model relies on storing additional metadata as well as splitting data in order to support more complex operations. A repercussion of this choice is the fact that we now rely on transactions for correcteness and consistency. Research is beeing done in this area at SICS.

Load balance - Our keyspace was designed to ease the development of the described data model, but at the same time, introduced some new problems. By grouping some keys together we managed to make use of the offered services in order to provide ever more complex services. However, this grouping of similar data introduces imbalance in our keyspace. We tend to move together data that has similar access paterns, so we end up with regions of the key space that suffer a dramatic increase in number of operations, while other ranges are not accessed so often. Load balancing at the underlying layer can help dissipate these hot ranges, while still maintaining the logical structure of our keyspace. There is active work beeing done at SICS in this area and it is our hope to use it in our work to improve the oferred performance.

Geo-replication - Geo-replicated data can increase the resilience of a system to large scale failures or can be used to move data closer to its user and decrease access latency. With our data model, defining geo-replication becomes adding more attributes to our data. The parameters of geo-replication could also be modified at runtime to offer an even more flexible mechanism. If the system detects heavy access from one particular user, it could try to move relevant data closer to it in order to improve its experience.

Dynamic change of object layout on disk - With the presented mechanism for changing the way data is stored on disk, an interesting research topic would be a data manager with machine learning support that could learn to identify different data access paterns and adapt the layout of the object in storage in order to improve the performance.

Providing a full query execution engine - Our prototype provides a basic query implementation, with many missing features. Future research in this area needs to be invested in order to provide better query support. Our query implementation lacks support for join operations and there are many optimizations that could be done in order to improve its performance.

Bibliography

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, Jun. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1365815.1365816

[2] A. Lakshman and P. Malik, "Cassandra - A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 35, Apr. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1773912.1773922

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, p. 205, Oct. 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1323293.1294281

[4] K. Chodorow and M. Dirolf, "MongoDB: The Definitive Guide," Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1941134

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107, Jan. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1327452.1327492

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*. New York, New York, USA: ACM Press, Jun. 2008, p. 1099. [Online]. Available: http://dl.acm.org/citation.cfm?id=1376616.1376726

[7] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687553.1687609

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems*

and Technologies (MSST). IEEE, May 2010, pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1913798.1914427

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, p. 59, Jun. 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1272998.1273005

[10] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive Analysis of Web-Scale Datasets," *Communications of the ACM*, vol. 54, no. 6, p. 114, Jun. 2011. [Online]. Available: http://dl.acm.org/ft_gateway.cfm?id=1953148&type=html

[11] J. Baker, C. Bondç, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. L´eon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," pp. 223–234, 2011. [Online]. Available: http://www.citeulike.org/user/m_brugger/article/8530095

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," pp. 251–264, Oct. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387905

[13] C. Arad, T. M. Shafaat, and S. Haridi, "CATS: linearizability and partition tolerance in scalable and self-organizing key-value stores," May 2012. [Online]. Available: http://soda.swedish-ict.se/5260/1/cats-sics-tr-2012-04.pdf

[14] C. I. Arad, "Programming Model and Protocols for Reconfigurable Distributed Systems," Ph.D dissertation, KTH - Royal Institute of Technology, Stockholm, 2013.

[15] S. H. Afzali, "Consistent Range-Queries in Distributed Key-Value Stores," M.S. Thesis, KTH - Royal Institute of Technology, Stockholm, 2012.

[16] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, Jun. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=564585.564601

[17] T. Lipcon, "Design Patterns for Distributed Non-Relational Databases." [Online]. Available: http://www.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases

[18] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer

lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, Feb. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=638334.638336

[19] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=200836.200869

[20] M. E. ul Haque, "Persistence and Node Failure Recovery in Strongly Consistent Key-Value Datastore," M.S. Thesis, KTH - Royal Institute of Technology, Stockholm, 2012.

[21] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, "Walnut: A Unified Cloud Object Store," in *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*. New York, New York, USA: ACM Press, May 2012, p. 743. [Online]. Available: http://dl.acm.org/citation.cfm?id=2213836.2213947

[22] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=279227.279229

[23] J. Dean, "Challenges in building large-scale information retrieval systems," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining - WSDM '09*. New York, New York, USA: ACM Press, Feb. 2009, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1498759.1498761http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//people/jeff/WSDM09-keynote.pdf

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, p. 143, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1807128.1807152

List of Figures

List of Figures

Acronyms

ACID atomicity, consistency, isolation and durability properties. 9

API application programming interface. 10, 13, 39

CAP consistency, availability and partition tolerance. 6

JPA Java Persistence API. 24

NoSQL "Not only SQL". 1, 2, 5, 9, 10

OLAP online analytical processing. 2, 10

OLTP online transaction processing. 2, 10

ORM Object-Relational Mapping. 3, 29

RDBMS relational database management system. 1, 9

SICS the Swedish Institute of Computer Science. 2–4, 31, 39, 40

SQL structured query language. 2, 10

YCSB Yahoo! Cloud Serving Benchmark. 30, 31, 37