

# Persisting Objects in Redis Key-Value Database

Matti Paksula

*University of Helsinki, Department of Computer Science*

*Helsinki, Finland*

*matti.paksula@cs.helsinki.fi*

**Abstract**—In this paper an approach to persist objects in Redis key-value database is provided. Pattern focuses on the consistency of objects under race and failure conditions. Proposed pattern is useful for cloud storage implementations that require speed and scalability. This approach can optionally be implemented with schemaless data model. Implementation can be done with any language, although the examples are presented with Ruby. Short overview on cloud architecture design is given as setting the context for using this persistence pattern.

**Keywords**—NoSQL, Key-value databases, Scalability, Cloud architectures, BASE, ACID, Ruby on Rails, Object persistence.

## I. INTRODUCTION

Recent beginning of "NoSQL movement" has brought a lot of attention to key-value databases or stores (from now on *KVS*). Cloud web applications require fast requests and the use of a relational database (from now on *RDBMS*) can sometimes be major bottleneck in the architecture [13]. This does not however imply that *RDBMS*es should be avoided. The problem is that they are being used as a generic building block for every problem. Simplified *KVS* brings speed, but also provides very little. This leads to major shift of responsibility from data persistence layer to developers.

NoSQL is easily understood as anti-SQL, but it should be understood as "Not *Only* SQL". *KVS* are not suitable for replacing relational databases, but to provide better approach in meeting non-functional requirements especially in cloud architectures. These databases or stores can not provide ACID style transactions. However, object persistence needs to be consistent nevertheless. This can be achieved with good design patterns that are implemented both in the architectural and component level. In this paper an example pattern for storing objects in *KVS* is given and at some level compared to *RDBMS* persistence.

The rest of this paper is organized as follows: First some background on cloud application architectures and key concepts of *KVS* are given. Some comparison of *KVS* against *RDBMS* is also done. Then introduction for object persistence in *KVS* is given. After these main principles of object persistence, different *KVS* implementations are shortly compared and then the key concepts of Redis *KVS* are explained. A Redis specific implementation of object persistence is illustrated with pseudocode like examples. In

the last part implementation is evaluated and examples of usage with ActiveRedis (Ruby on Rails object persistence library) are shown together with a rough performance test.

## II. ARCHITECTURAL MOTIVATION

Web application needs to serve requests as quickly as possible. Most of the requests consist mainly of reads from the database. Brewer's CAP theorem says that in distributed environment, like cloud, we have three requirements: *Consistency*, *Availability* and *Partition Tolerance*, but we can only have *two* out of these three. What this means is that system operations on data need to be consistent, it has to be available for the user and data has to be stored in multiple locations. In order to satisfy all three, we can select two and solve the third requirement by a workaround [1], [2].

Instead of writing our application to satisfy ACID requirements (*Atomicity*, *Consistency*, *Durability* and *Isolation*) we write our architecture using BASE approach (*Basically Available*, *Soft-state* and *Eventually consistent*) [3]. This means that we do our best to keep data available, but for example if some network link goes down, we still serve that data we have access to. Modifications to the data are guaranteed to perform eventually for example by deferring write operations to a job queue. Due the nature of web applications where requests are independent and have some time between them, this approach can be very successful.

Dropping ACID on architectural level gives us also the opportunity to drop it in the database level also. This is where the *KVS* option becomes interesting. Usually most of the operations are reads and scaling web application means scaling up the reads.

## III. KVS vs RDBMS

*RDBMS* has to satisfy ACID requirements when modifying the data. This makes partitioning data across different physical locations hard as data needs to be written in all the locations before *OK* can be returned. Also the write operations tend to be as hard disk IO becomes the bottleneck. *RDBMS* also have fixed schema and this can for example lead to bloated table designs with lot of unused columns.

*KVS* are really simple by design. A value can be saved with a unique key. That value can later be loaded by that same key. This is more generally known as hash addressing

described already in 1968 by R.Morris [6]. KVS do not natively support relations or full text queries, but can be a better match to some situations. Examples of such situations are storing operational data, metadata, logging, message queues, caching and serving pre-computed data. One approach to implement a KVS is to store keys in-memory and persist the dataset to file asynchronously. When the KVS server is restarted, all the data is loaded into memory from that file.

RDBMS have serious advantages over KVS: they are well tested, have good management tools and programming patterns available. Key-values on the other hand are fairly new, each of them have different approach and excel only in a very narrow segment [13].

Sometimes the difference between RDBMS and a KVS is almost non-existing. For example when using MySQL by indexed primary keys it can perform really well and depending on the needs can also be easier to partition than current KVS implementations. Also some operations, like sorting can be faster to do at the application level when data is suitable for that. There is no single technology to implement a scalable system. Such system has to be crafted with a combination of different technologies and architectural designs.

Developers existing familiarity with RDBMS and the fact that as matured systems they are easier to understand, might be good enough reason for not to consider KVS option. On the other hand scalability requirements might force to consider it.

#### IV. PERSISTING OBJECTS IN KVS

Objects are traditionally persisted in RDBMS with Object-Relational mapper (ORM). While KVS implementations differ, there are some common requirements for persistence.

##### A. Objects

Objects are instances of classes implemented in host language. As structure for object and methods etc. is in the code, the entity itself is defined by its unique identifier and attribute values. Simple attribute types like *strings*, *integers*, *booleans* and *floats* are suitable for storing in KVS.

A collection of objects can be defined as a collection of unique identifiers. From these identifiers we can get all the attributes from the database required to create these objects.

An objects attribute can be stored with following key-value pair *ClassName:Identifier:AttributeName*  $\Rightarrow$  *Value*.

Example: A car with two attributes as separate keys

Car:34:color  $\Rightarrow$  green

Car:34:speed  $\Rightarrow$  120

This approach can be problematic as described in the next section.

##### B. Consistency of objects

Concurrency of reads and writes makes previous approach problematic as KVS does not provide same kind of transactions that we are used to in the RDBMS world. If this is not considered, it can lead to potential inconsistencies. For example during attribute read operation, a concurrent update operation can be modifying the same object. When object is returned to application, it can have some old and some new values. Also, when deleting an object it is possible that execution is terminated for example by power failure and only some of the attributes were deleted.

##### C. Motivation for adapters

Object persistence adapters are needed between KVS and application even if the operations are trivial. Using an adapter between application and KVS helps to avoid race conditions and other programming errors. It is also possible to utilize components that are designed for some ORM interface when a KVS adapter implements it.

##### D. Schemaless data model

It is common that not all of the instances have exactly the same attributes than others. Some might have attributes that exist only in the minority of all objects. Schemaless data model is useful during the development phase, but also interesting for the production. For example, an administrator could add attributes on the fly for certain instances and the application could be designed to show only those attributes that are present in the instance. Schemaless approach does not necessary equal chaos if the architecture is designed to support it. Simple example of this is provided in the evaluation chapter.

#### V. KEY-VALUE STORAGEES

There are different implementations of KVS and most of them have not yet fully matured. Cloud services have defined a new set of problems what pioneers like Facebook and Amazon are addressing these problems with their own distributed implementations like Cassandra [4] and Dynamo [5]. These distributed large scale KVS have proven to work well for them, but for smaller scale (not massively big) web application they might be too heavy. Things we get for granted with RDBMS like indexing data, providing query language and views are currently under research [7], [13], [14].

Smaller scale KVS implementations include for example CouchDB, MongoDB and Redis. They differ from each other and are suitable for different kind of tasks. For example MongoDB provides a relational database like query language and is essentially a document database. All of these three support partitioning and replication of the data at some level. Mostly experimental object persistence adapters are available for all implementations.

## VI. REDIS INTRODUCTION

Redis does not support complex queries or indexing, but has support for data structures as values. Being very simple it is also fastest KVS implementations for many basic operations. Speed and data structures are interesting combination that can be used for simple object persistence.

Data structures in Redis are more generally called *Redis Datatypes*. These include strings, lists, sets, sorted sets and hashes. Redis provides commands that can be used to modify these types. For example list supports normal list operations, like push and pop.

The whole dataset is kept in-memory and therefore can not exceed the amount of physical RAM. Redis server writes entire dataset to disk at configurable intervals. This can also be configured so that each modification is always written on the disk before returning *OK*. Master-Slave replication is also available and clustering is currently under development [9].

### Atomic commands and Redis Transactions

Every Redis command is executed atomically. For example the INCR command increments integer value of key atomically. Two separate commands are not atomic without combining them with *Redis Transactions*.

Transactions in Redis are actually queued commands that are executed atomically in sequence. If one command raises an error, all the other commands in the queue are still processed. Redis starts queuing commands after command MULTI and executes those in one atomic transaction with EXEC.

### A. Sorted sets and Hashes

Sorted sets are similar to RDBMS indexes. Sorted sets contain members that are sorted by a special integer value *score*. For example "ZADD *Foo:all* 10 10" stores value 10 to key *Foo:all* with score of 10.

Hashes are key-value pairs inside a key and suitable for storing objects attributes. For example "HMSET *Cat:2:attributes* color black age 20" adds two key-value pairs to the key *Cat:2:attributes*.

## VII. IMPLEMENTATION

In following implementation each persisted object has its own unique integer identifier. Based on this identifier a key *Foo:id:attributes* containing the objects attributes is created (assuming that object is named Foo). This key stores the attributes as Redis hash. When the object is created, a special sorted set *Foo:all* is updated to contain the identifier. With this "master" set it is possible to know if object is persisted or not.

Because write operations are different atomic operations, concurrency of writes can lead to inconsistent objects. If a delete operation is stopped (for e.g. power outage) in the between of attribute deletion and removal from the master

set, object loses its attributes. Also some garbage keys can exist in the memory if the object is removed from the master set of *Foo:all* and attributes are not deleted left. To prevent these scenarios each operation has to be designed for race conditions and process termination.

Redis specific implementation details are given in next pseudocode like algorithms. Redis commands are written in capital letters. Only the basic operations are described as further work is needed to provide simple relations and indexes. In the examples class named *Cat* is used over *Foo*, because cats have names and lengths unlike foos. Listing 1 shows higher level usage with Ruby adapter.

```
cat = Cat.new(:name => "Longcat",
             :length=>150,
             :color=>"white")
cat.save
same_cat = Cat.find(cat.id)

# Prints out "Longcat"
print(same_cat.name)
```

Listing 1. ActiveRedis example

### A. Create

Create operation is shown in algorithm 1. Operation fetches new identifier from shared sequence. Then all attributes are set in Redis hash. After this object is added to set of all persisted objects. This is done with ZADD command, that adds objects identifier integer with the same integer as score in the sorted set. Command HMSET accepts many key-value pairs (*HMSET key field1 value2 ... fieldN valueN*). Each attribute can also be written separately, this approach is shown in alternative create algorithm 2.

Increasing identifier and storing attributes are done in separate atomic operations. This could lead to situation where identifier is increased, but no object is persisted. This is acceptable for worst case scenario as the dataset is still consistent. Atomic transaction guarantees that attributes do not get written without being added to sorted set *all*.

---

### Algorithm 1 Create an object

---

**Require:** *attrs*

*id* ← INCR "Cat:sequence"

**MULTI**

{HMSET "Cat:3:attributes" "name" "lolcat" "age" "3"}

HMSET "Cat:*id*:attributes" *attrs*

ZADD "Cat:all" *id id*

**EXEC**

---

### B. Load

It is possible to get all the attributes after the object is persisted by using unique object identifier. Returning value

---

**Algorithm 2** Alternative object creation

---

**Require:** *attrs**id* ← INCR "Cat:sequence"**MULTI****for all** *key, value* in *attrs* **do**    **HSET** "Cat:*id*:attributes" *key value***end for****ZADD** "Cat:all" *id id***EXEC**

---

of ZSCORE operation against this key is the score of the key in sorted set. If key was not found then a special value *NIL* is returned. Attribute read returns an empty set also when an object does not have any attributes. Therefore reading score and attributes atomically is required to determine if object was stored without any attributes. Also a race condition with delete operation is possible when operations are not atomic.

---

**Algorithm 3** Load an object

---

**Require:** *id***MULTI***id* ← ZSCORE "Cat:all" *id**attrs* ← HGETALL "Cat:*id*:attributes"**EXEC****if** *id* is zero **then**    **return false****else**    **return** *attrs***end if**

---

### C. Update

Updating an existing object is equal to the creation as shown in algorithm 1. Alternative creation algorithm (algorithm 2) can provide better performance when only some of the keys are updated. In the worst case performance is the same as in create.

Update can also be used when adding and removing attributes to and from an existing object. This requires a host language that supports adding new attributes dynamically in objects or some other method.

### D. Delete

Delete (in algorithm 4) is done by combining deletion of attributes and removal from sorted set *all* into one atomic transaction.

### E. Count

Counting (in algorithm 5) the number of persisted objects is the cardinality of sorted set *all*. Time complexity is  $O(1)$ .

---

**Algorithm 4** Delete an object

---

**Require:** *id***MULTI****DEL** "Cat:*id*:attributes"**ZREM** "Cat:all" *id***EXEC**

---

---

**Algorithm 5** Count objects

---

*i* ← ZCARD "Cat:all"**return** *i*

---

## VIII. EVALUATION

This implementation is limited, but shows that Redis Datatypes and operations have enough power to implement flexible object persistence library while maintaining object consistency. These operations work as a starting point for implementing simple object persistence. In addition to ORM this persistence pattern can be used to address feature requirements in a typical web application.

### A. Real examples with ActiveRedis

Previous proposed implementation is extracted from ActiveRedis, an object persistence library for Ruby on Rails 3 that uses frameworks shared modules like attribute validations [10]. ActiveRedis provides similar programming interface like ActiveRecord, the SQL persistence library for Ruby on Rails [12]. Following three real examples are given for Ruby on Rails 3 using ActiveRedis.

1) *Showing objects with different attributes:* Let's consider a system that has users who have profiles. Profile can contain many key-value pairs, like *age*  $\Rightarrow$  34. Profile is loaded with ActiveRedis model and shown in the view by the keys as shown in listings 2 and 3.

```
def show
  @profile = Profile.find(params[:id])
end
```

Listing 2. ProfileController#show that loads profile

```
<h2>Your profile</h2>
<ul>
  <% @profile.attributes.each_key do |key| %>
    <li><%= key %>: <%= @profile.attributes[:key] %></li>
  <% end %>
</ul>
```

Listing 3. Showing profile in HTML ERb template

2) *Photo metadata:* Photos are stored in RDBMS and accessed with ActiveRecord. Photo contains identifier for ActiveRedis model *ExifMetadata* that can have *n* key-value pairs from the JPEG like *Color Space*  $\Rightarrow$  *sRGB* and *Canon Image Size*  $\Rightarrow$  *Large*.

3) *Last logins:* Messages are stored in ActiveRedis model *LastLogin*. When *User* that is stored in ActiveRecord logs in, a LastLogin is created and this can be shown in the front

Table I  
PERFORMANCE TESTS

Operation	ActiveRedis	ActiveRecord + PostgreSQL
Create 10,000 posts	14.55s	36.48s
Count 10,000 posts	0.00s	0.00s
Load 10,000 posts	3.33s	6.25s
Update 10,000 posts	15.77s	15.83s
Delete 10,000 posts	6.60s	10.28s

page. LastLogins contains a Redis list that can easily either be trimmed to contain only  $n$  logins or ranged to show only latest 10 logins.

### B. Performance

Comparing performance with RDBMS is not really feasible as RDBMS and KVS should be used for different things. This gives however some rough numbers for comparing the speed. Performance tests were done with ActiveRedis and ActiveRecord with PostgreSQL. Ruby interpreter takes about half of the CPU resources while doing these tests, so the numbers are not valid for measuring throughput results. It does however show the order of magnitude that is achieved. Also test results vary also by the type of object that is persisted.

In the test 10,000 *Posts(author:String, topic:String, content:Text)* with random length values were created, loaded, updated, counted and deleted. Results in the table I show that with this special scenario ActiveRedis performs significantly faster than ActiveRecord. As all the fields are updated in update operation (worst case), update time equals create + overhead.

## IX. CONCLUSION

During the writing process of this paper Redis started to provide hash datatype that is being extensively used in the persistence pattern described here. This made few existing Redis adapters and the initial persistence pattern obsolete. All of these technologies are very new. Redis, ActiveRedis and Ruby on Rails release 3.0 are under stabilization and not currently feasible for production. A message from redis mailing list sums up the current situation:

”However, I have to admit, I’ve definitely had sleepless nights. Redis is a new technology. We explored new territory and had little experience (internal or otherwise) to rely upon. We’ve had to debug and patch the driver, write our own query layer, backups, data injection scripts, all sorts of stuff you take for granted from the SQL world, and we’re still tweaking it, and wishing for pre-made solutions.” [8]

Key-value databases have not yet fully matured, but do give interesting options for developing distributed and scalable web applications. Each of different implementations are good for a set of problems, but not for every

problem. Therefore well defined architectural design patterns and adapters for storing objects are needed to provide a tested approach for developing stable software. Relational databases are still good for handling relations and supporting strong consistency requirements.

### ACKNOWLEDGMENT

I would like to thank Juha Suuraho, Atte Hinkka and Petrus Repo for providing valuable comments on the technical implementation details of the persistence pattern.

### REFERENCES

- [1] Brewer, E. A. 2000. Towards robust distributed systems (abstract). In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (Portland, Oregon, United States, July 16 - 19, 2000). PODC '00. ACM, New York, NY, 7. DOI= <http://doi.acm.org/10.1145/343477.343502>
- [2] Gilbert, S. and Lynch, N. 2002. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 2 (Jun. 2002), 51-59. DOI= <http://doi.acm.org/10.1145/564585.564601>
- [3] Pritchett, D. 2008. BASE: An Acid Alternative. Queue 6, 3 (May. 2008), 48-55. DOI= <http://doi.acm.org/10.1145/1394127.1394128>
- [4] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network, product presentation at SIGMOD 2008.
- [5] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. 2007. Dynamo: amazon’s highly available key-value store. In Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (Stevenson, Washington, USA, October 14 - 17, 2007). SOSP '07. ACM, New York, NY, 205-220. DOI= <http://doi.acm.org/10.1145/1294261.1294281>
- [6] Morris, R. 1968. Scatter storage techniques. Commun. ACM 11, 1 (Jan. 1968), 38-44. DOI= <http://doi.acm.org/10.1145/362851.362882>
- [7] Agrawal, P., Silberstein, A., Cooper, B. F., Srivastava, U., and Ramakrishnan, R. 2009. Asynchronous view maintenance for VLSD databases. In Proceedings of the 35th SIGMOD international Conference on Management of Data (Providence, Rhode Island, USA, June 29 - July 02, 2009). C. Binnig and B. Dageville, Eds. SIGMOD '09. ACM, New York, NY, 179-192. DOI= <http://doi.acm.org/10.1145/1559845.1559866>
- [8] Message in Redis mailing list <http://groups.google.com/group/redis-db/msg/ca398a90ea78bfc5>
- [9] Redis home page <http://code.google.com/p/redis/>
- [10] ActiveRedis home page <http://www.activeredis.com>
- [11] ActiveModel and Rails 3 explained <http://yehudakatz.com/2010/01/10/activemodel-make-any-ruby-object-feel-like-activerecord/>

- [12] Ruby, S., Thomas, D., and Hansson, D. 2009 Agile Web Development with Rails, Third Edition. 3rd. Pragmatic Bookshelf.
- [13] Armbrust, M., Lanham, N., Tu, S., Fox, A., Franklin, M., and Patterson, D. A. Piql: A performance insightful query language for interactive applications. First Annual ACM Symposium on Cloud Computing (SOCC)
- [14] Acharya, S., Carlin, P., Galindo-Legaria, C., Kozielczyk, K., Terlecki, P., and Zabback, P. 2008. Relational support for flexible schema scenarios. Proc. VLDB Endow. 1, 2 (Aug. 2008), 1289-1300. DOI=<http://doi.acm.org/10.1145/1454159.1454169>