Key-Value Store in Scala

Phillip Grote, Daniel Sharkov

Projektbericht Datenbankprojekt
Fakultät IV
Elektrotechnik und Informatik
der Technischen Universität Berlin

Fachgebiet:
Datenbanksysteme und Informationsmanagement

Berlin 2018
Winter Semester

Contents

Abstract

This report walks the reader through the process of building a key-value store written in the Scala programming language. A key-value store is, compared to the more traditional relational database paradigm, a simpler approach to storing data. Keys with their corresponding values can be stored, retrieved and deleted. Our work is based on three main building blocks: buffering, indexing and concurrent request handling.

1 Introduction

Over the last three and a half months we have designed and implemented a Key-Value Store using the Scala programming language. This Key-Value Store is able to store key-value pairs. Both key and value can be arbitrary strings. The Key-Value Store supports simple CRUD operations. Additionally, the database is able to retrieve a range of key-value pairs with a single request. Section 2 describes the buffering strategy in more detail, and Section 3 provides insights in how we implemented the indexing. Section 4 deals with the question of how to handle concurrent requests. Finally, Section 5 presents our conclusions.

2 Buffering

Retrieving data from disk can be many times slower as opposed to fetching it directly from RAM (Johnson and Shasha 1994, 439). In order to build a high-throughput database system it follows that working on reducing disk accesses should be a priority. Buffer management can be described as the process in which data is kept inside of main memory, based some access criteria. Over the years a lot of effort has been put towards the development of buffering strategies, leading to the advent of many algorithms in that area. ~~We will present buffering algorithms~~, our implementation and a performance evaluation of our solution.

2.1 Buffering Algorithms

~~The following section will present an overview of three common buffering algorithms we implemented including their advantages and disadvantages.~~

2.1.1 LRU

As a starting point, we considered the Least Recently Used (LRU) algorithm. This algorithm could be implemented by using a FIFO-queue, which ~~allows that an element can be pushed to the top of the queue~~. The fact that a random element can be pushed to the top differentiates this algorithm from a simple FIFO algorithm. The workflow is as follows:

- New records are placed at the front of the queue
- If at this time the queue is full, the last element is removed
- Accessed records that are already present are pushed to the top

LRU has been a popular algorithm over the years, since it "[...] never replaces more than a factor B as many elements as an optimal clairvoyant algorithm (where B is the size of the buffer)" (Johnson and Shasha 1994, 439). However, such a factor results in substantial fluctuations between optimal and worst-case performance.

2.1.2 2Q

The 2Q algorithm builds upon LRU by providing additional data structures to better identify pages that have a low-access density over time. A drawback to LRU is that it admits pages to the buffer only based on how often they have been referenced, ~~it does not take into consideration~~ the time-frame over which this referencing takes place (Johnson and Shasha 1994, 440-41). The 2Q algorithm aims to solve this with the introduction of two additional FIFO-queues. The workflow is as follows:

- Record was found:
  - In the main queue, approach as with LRU
  - In the first FIFO-queue, do nothing

  - In the second FIFO-queue, place at the head of the main queue

- Record was not found:
  - Place it in at the head of the first FIFO-queue
  - In case the first FIFO-queue is now full, page out its tail to and place it at the end of the second FIFO-queue

The result is an algorithm that ignores frequent accesses in short periods of time. Records are not promoted upon consecutive accesses, if they are in the first queue. Only if the record has been accessed with a specified latency, it will be promoted to the main buffer. A drawback to 2Q is that there is no clear way for determining the size of the queues other than experimenting.

2.1.3 ARC

The main principle behind the Adaptive Replacement Cache (ARC) algorithm is similar to that of 2Q – taking into consideration both frequency and recency of records accessing the buffer. There are some differences in the internal structure however. With ARC we have two main LRU queues and two "ghost" lists. The first LRU-queue holds elements that have been accessed once, whereas elements accessed for the second time are placed in the second LRU-queue. Elements removed from the first LRU-queue are placed in the first "ghost" list and elements removed from the second LRU-queue – in the second (Megiddo and Modha 2004, 60-61). For records that are discarded into the ghost lists only metadata is held. If a record that is inside a ghost list is accessed, the respective LRU-queue (to which the ghost list belongs) has its size increased and the other LRU-queue – its size decreased. Thus, a window of the same size is maintained, which slides across the recently and frequently accessed records, therefore adapting based on the observed access patterns.

2.2 Implementation

All of the three algorithms that were described above implement the trait Buffer, which defines the functions *get(), set(), range(), delete(), flushBuffer()* and *hitRate()*. The first four represent the possible database queries, *flushBuffer()* is used to clear the contents of the buffer and the last function provides a way to observe the efficiency of the buffer. The trait also provides a basic implementation of those functions, in which the buffer is bypassed. Each of the buffering algorithms can be instantiated using this trait.

All buffer implementations follow an approach, where a HashMap is used to track the keys and their respective values. This allows for amortized constant time on *get* and *set* operations ~~(granted the respective keys are found in the buffer)~~. When a record gets discarded from its respective buffer, then it is also removed from the map. Particularly in the case of the ARC algorithm records are removed from the map once they fall into a ghost list as per the algorithm's definition. On the 2Q and ARC algorithms, since they both have more than a single queue within the buffer, additional maps are used as a way to find where a particular key resides.

2.3 Evaluation

In the following section we provide an overview of how well the different algorithms perform for get queries compared to one another and to having no buffering mechanism. The results are listed in Table 1.

2.3.1 Setup

The experiment was performed on a set of 250 unique key-value pairs. The buffers have a size of 125 key-value pairs. The size is intentionally set to represent 50% of the total number of key-value pairs that were used. Doing so would mean that on average for each pair one swap will be performed.

2.3.2 Observations

On average the buffering algorithms provide roughly a 40% improvement over having no buffering. As one might observe, the 2Q is the worst performing algorithm, being around 25% slower than the other two algorithms. We attribute this to the way this algorithm works - the same key-value pair can be present in multiple queues, making the effective size of the buffer smaller and thus impacting performance negatively.

2.3.3 Conclusion

The results, while offering an improvement over the no-buffer alternative are not satisfactory. Even with its more complex mechanism ARC manages to overtake LRU by a small margin only in the last test. A possible reason for this and the overall performance can be attributed to each query being processed individually. This becomes problematic when an element that is present has been accessed again and needs to go to the top of the queue. Since there is equal probability for the element to be in any position within the buffer, doing so requires on average *n/2* operations, where *n* is the size of the buffer.

Our suggestion for improving upon the current buffer implementation is adding a mechanism, which collects all of the queries that were received in some specified interval of time and then sends the over to the buffer in bulk. In doing so the buffer will be scanned only one time and multiple operations will be executed at once and a resulting bottleneck could be avoided.

A second option is to only push the respective element to the head of the queue, without deleting its previous instance. A special flag will be set for the element to indicate that it is present more than once in the buffer. Once a predefined threshold is reached a scan starting from the top of the queue will be performed, deleting redundant instances of the same keys further down the list. This would mean slightly lower efficiency, due to an element taking more than one spot inside the buffer, however it should result in net gain in performance, due to reduction of operations that require linear time.

#Queries	No buffer	LRU	2Q	ARC
1000	56s	31s	42s	34s
2000	114s	58s	80s	64s
4000	230s	116s	156s	113s

Table 1: Retrieval time for randomly generated get queries

3 Indexing

The amount of data the Key-Value Store should be able to store exceeds the main memory. Therefore we had to implement some sort of persistence layer. Additionally, to minimize file I/O we had to choose an indexing strategy. Without an indexing strategy it would be necessary to load all files we stored in the filesystem into memory and search for the key-value pair the user asked for. This approach would be very time consuming and inefficient. Therefore ~~We~~ chose to use Sorted String Tables (SSTable). This choice was inspired by the design of Google's Bigtable (Chang et al. 2008). In the following we will present the basic idea behind SSTables and our implementation of this idea. Finally, we present the results of our performance evaluation.

3.1 Sorted String Tables

The idea behind this indexing strategy is that the key-value pairs are stored as strings sorted in lexicographical order by key to a SSTable. Each SSTable contains a sequence of blocks and one block is composed by several key-value pairs (the size of a block and the number of blocks in one SSTable is configurable). The index for such a SSTable will hold the first key of each block and the corresponding byte offset. To retrieve a certain key-value pair from such a file we must search the index for its highest key which is smaller or equals the key we are looking for. This key maps to the byte offset of the block in which the key-value pair might be. Now it is possible to only load this block into memory and search for the data we are looking for.

An important aspect of SSTables is that in general they will not change after they have been written. If the value of a key-value pair changes, a new key-value pair will be generated. This new data will be held in memory and eventually stored in a new SSTable. Therefore to retrieve correct data it is necessary to keep track of the creation time of each SSTable and to start looking for the key-value pair in the latest SSTable.

To delete a key-value pair a new key-value pair will be stored in a new SSTable with a so called tombstone as value. A tombstone is a specific token. If the system reads this token, it will know that the key-value pair has been deleted.

To release space the system periodically runs a compaction job which will scan through the SSTables and will merge them so that the old and inaccurate key-value pairs can be discarded.

3.2 Implementation

At the lowest layer of our Key-Value Store is the persistence layer. The persistence layer is implemented by the FileManger. The FileManager is responsible for storing the data in files to the filesystem and retrieving key-value pairs from the filesystem, if they are not held in memory. In the overall architecture, the FileManger is communicating directly with the Buffer. The FileManager manages the SSTBuffer and one SSTIndex for each SSTable. We decided to map each SSTable completely in memory. This means that the FileManger stores the SSTIndex of each SSTable in a Scala ArrayBuffer. In the following we will present more details about the implementation of the FileManger by showing how the FileManger handels the requests from the Buffer.

3.2.1 Storing Key-Value Pairs

If the Buffer asks to store a key-value pair, the FileManger buffers this key-value pair until it is possible to generate a SSTables. This is done by inserting the key-value pairs into the SSTBuffer. The SSTBuffer buffers the key-value pairs in a Map (*scala.collection.mutable.Map[String, String]*) until the configured size is reached. As soon as the configured size is reached the FileManger will flush the SSTBuffer. This means that the SSTBuffer will generate a new SSTable by sorting the buffered key-value pairs by key. Additionally, a SSTIndex for this SSTable will be generated which is returned to the FileManger. The FileManger will then append the SSTIndex to the ArrayBuffer which stores the indices for each SSTable.

3.2.2 Retrieving Key-Value Pairs

If the Buffer asks for a key-value pair, the FileManger has to check the SSTBuffer first, because the SSTBuffer holds the latest key-value pairs. If the FileManger can not retrieve the data from the SSTBuffer the FileManger checks the latest SSTIndex to find a block in which the data might be. If a block is found the FileManger loads this block into memory and scans it for the key-value pair. If necessary the FileManger repeats these steps with the next SSTIndex until the key-value pair is found. Eventually the FileManger returns either a Map containing the data the Buffer asked for or an empty one.

3.3 Evaluation

In this section we present our evaluation results of the implemented indexing solution. The results are listed in the Table 2-5.

3.3.1 Setup

For the performance evaluation we isolated the FileManager from all the other parts of the Key-Value Store. As already mentioned a SSTable contains a sequence of blocks. For our evaluation we used a block size of 64000 bytes and a SSTBuffer size of 1280000 bytes.Furthermore, we generated a file with one million line separated key-value pairs.

We compared our final implementation against a much simpler first prototype. This first prototype stores all data in one file to the filesystem. To retrieve a key-value pair it has to load the complete data into memory before it is able to retrieve key-value pairs by key. Besides the obvious limitation of this prototype (it cannot handle more data than would fit into memory), it doesn't use indexing. That is why we expect that it will need much more time to retrieve a value by key as our implementation with indexing.

3.3.2 Observations

Storing Data. We inserted the test data line by line into both of our Key-Value Stores. We repeated this process ten times.

	No Index	**Index**
average	2035ms	4688ms
deviation	529ms	759ms
minimum	1717ms	3785ms
maximum	3606ms	5946ms

Table 2: Insertion time for randomly generated data

Retrieving Random Keys. After we inserted the generated data in our Key-Value Stores, we were able to retrieve values by random keys. We generated two random keys with the same utilities we used to generate the random data. Then we retrieved the corresponding values from our Key-Value Store implementations. We repeated this procedure ten times. As shown in table 3 the implementation with indexing was much faster than the implementation without indexing.

	No Index	**Index**
average	5437ms	156ms
deviation	840ms	56ms
minimum	4021ms	131ms
maximum	7356ms	323ms

Table 3: Retrieving time for two randomly generated keys

Retrieving Existing Keys. We tried to retrieve two values by random but existing keys. As always, we repeated this procedure ten times. As might be expected, our solution was again much faster than without indexing.

	No Index	**Index**
average	4889ms	141ms
deviation	665ms	58ms
minimum	4121ms	113ms
maximum	6235ms	315ms

Table 4: Retrieving time for two randomly generated but existing keys

Retrieving Key Range. In the last test we evaluated the performance of retrieving values by a range of keys. We generated two random keys, a lower and an upper bound. Then we retrieved values by this range ten times. On average, it took 1316ms to retrieve this range which included 255053 key-value pairs.

3.3.3 Conclusion

The implementation without indexing is faster when it comes to inserting data into the database. This was not surprising, because it will not generate SSTables and corresponding indices. But it is worth it to accept this minor slow down, because indexing reduced the retrieving time significantly.

	Index - Range
average	1316ms
deviation	417ms
minimum	904ms
maximum	2092ms

Table 5: Retrieving time for key range - 255053 key-value pairs

Range queries are much faster then point queries. It took less than 6μs to retrieve a single key-value pair. Compared with the, on average, 70ms to retrieve a single key-value pair (141ms to retrieve two key-value pairs) the retrieving by range is faster by orders of magnitude.

4 Concurrency

Up to this point the key-value store has the modules in place, that allow for a reduced number of IO-operations and for storing data efficiently on disk. However, the application is limited to having one user per instance. Ideally the system should be able to support a large number of users concurrently.

4.1 Client-Server Architecture

Implementing a client-server architecture is the first step towards achieving the goal of concurrency. With this approach the application is broken down into two independent components, the first being the actual database (the server part) and the second – the user interface (the client side). With the database running on the JVM, there are two main approaches to establishing a connection between the components – the Java IO API or the Java NIO API. In the following paragraph we give a brief overview of the two options and ~~try to~~ justify our choice.

4.1.1 Java IO and Java NIO

Both IO and NIO are based on the socket paradigm and both offer blocking IO operations for reading and writing from and to a socket. NIO also allows for non-blocking asynchronous connections. The latter is provided via an additional level of abstraction in the form of channels and selectors, where each connection is represented by a channel, which has a number of selectors. Selectors provide an intuitive way to test whether a channel is ready for either input or output operations. This in term allows only a single thread to handle all incoming connections. In comparison when using Java IO, a new thread must be created for each new connection i.e. the number of threads required grows linearly with the number of clients connected to the server, which can lead to high memory usage in a situation with thousands of active connections. The drawback for NIO is that it fails to take advantage of processors with multiple cores, due to ~~having~~ only one thread.

4.1.2 The Thread-Pool and Callback Function

To alleviate the potential under-use of hardware resources, the system has been fitted with a thread-pool, in which processing of the queries takes place. The Callable interface is then used to pass a function with a return value to that thread-pool. A problem that arises with the use of this interface is that retrieving the result ~~form~~ the computation is a blocking operation. A more optimal approach would have been the use of the CompletableFuture interface, since it provides a way to obtain the result in an asynchronous manner. We solve the problem by providing a callback function from the NIO module, which is used to notify the thread that deals with the connections of newly ~~processes~~ queries.

4.2 Locking

On relational database systems, as part of the ACID principle, one can choose from different isolation levels. Based on the selected level some or all of the three phenomena - non-repeatable reads, dirty reads and phantom reads - can be mitigated. Due to the simplicity of our key-value store and its lack of transactions, there is no need to worry about these problems. The word "transactions" here refers to all operations, which consist of multiple queries that are to be executed in an atomic way. Some locking strategy is however still required, since multiple users and hence threads can access the same resource concurrently.

4.2.1 Pessimistic- and optimistic locking

In database design there exist two general techniques for synchronizing data and avoiding race conditions – pessimistic locking and optimistic locking. The first can be described as the more standard approach, in which critical sections are introduced into the code, with the goal of making threads wait before entering, until some condition holds. With optimistic locking no actual locking takes place – instead each thread upon accessing an area of code, in which potential conflicts might arise, updates a shared variable with a timestamp of the time right before execution. If for the remainder of the execution that timestamp remains unchanged by another thread, the query is allowed to finish, otherwise a rollback must be initiated. While one might argue that in a system with a low level of concurrency this approach could lead to faster execution, we decided to select pessimistic locking. To us this seems to be the more general-purpose decision, which should by default be present, with optimistic locking being possibly included as an extra feature to be used in those cases where it makes sense. In the following paragraph we explain the pessimistic lock implementation that we adopted.

4.2.2 Lock Implementation

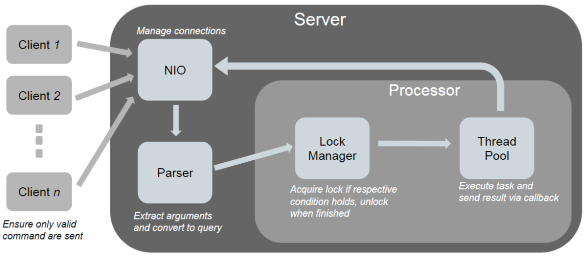
Server
Manage connections
Client 1
Client 2
Client n
NIO
Processor
Parser
Lock Manager
Thread Pool
Ensure only valid command are sent
Extract arguments and convert to query
Acquire lock if respective condition holds, unlock when finished
Execute task and send result via callback

Figure 1: Key-value store with concurrency

Since reading on the same records can be done without any conflicts, whereas writing causes conflict with existing writers and readers on the same record, we decided that a write-read lock would be the optimal choice. A read-write lock divides threads into readers and writers. An unlimited number of readers is allowed inside a critical section, if no writers are inside of it and only one writer thread is allowed, if no readers are inside of the critical section. The initial idea was having locking done on the record level, meaning that each record has its own read-write lock. This lead to the decision of mapping the records to a ReentrantReadWriteLock object.

The ReentrantReadWriteLock, while working well for *set* and *get* queries did not offer an intuitive way to lock elements within a *range* query, as the result of a *range* query can not be known in advance and thus one can not know which records are to be locked. For this reason and since we were not successful at finding a suitable replacement library, a decision was made to build a simple read-write lock from scratch. The lock would follow the same rules as the ReentrantReadWriteLock for *get* and *set* queries, however it would also incorporate an additional data-structure, which is to hold the currently running *range* queries. With this addition the rules for accessing the buffer become the following – readers can proceed if no writers are currently present, writers can proceed if no readers are currently present, and their request does not fall within the *range* of an in-progress *range* query and *range* queries can proceed as long as no writers fall within the range. The locking object that resulted does not offer reentrance, due to the additional complexity that this would have introduced. We believe this compromise is justified, ~~since only three critical sections exist~~ – ~~before accessing the buffer for any of the three available queries~~. This makes it trivial to avoid reentrance-related problems.

5 Conclusion

Our work is composed by three major parts: buffering, indexing and concurrent request handling. The evaluation of our buffering strategies showed that there is some further investigation necessary to find out why the more sophisticated buffering strategies (2Q and ARC) underperformed our expectations. Therefore we presented some ideas that could improve their implementation. Regarding the indexing the evaluation showed no surprises. Our implementation which uses SSTables and corresponding indices outperformed the naive implementation significantly. Nonetheless, there is a lot of room ~~of~~ improvement. For instance we don't implement a compaction procedure which would limit the waste of storage space. Finally, we were able to implement a first version for handling concurrent ~~request,~~ but due to time restriction we haven't been able to test this implemention, yet. Further investigating would show how our concurrent request handling behaves under load.

References

Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. "Bigtable: A Distributed Storage System for Structured Data." *ACM Trans. Comput. Syst.* (New York, NY, USA) 26, no. 2 (June): 4:1–4:26. ISSN: 0734-2071. doi:10.1145/1365815.1365816. http://doi.acm.org/10.1145/1365815.1365816.

Johnson, Theodore, and Dennis Shasha. 1994. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm." In *Proceedings of the 20th International Conference on Very Large Data Bases,* 439–450. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-153-8. http://dl.acm.org/citation.cfm?id=645920.672996.

Megiddo, Nimrod, and Dharmendra S. Modha. 2004. "Outperforming LRU with an Adaptive Replacement Cache Algorithm." *Computer* (Los Alamitos, CA, USA) 37, no. 4 (April): 58–65. ISSN: 0018-9162. doi:10.1109/MC.2004.1297303. http://dx.doi.org/10.1109/MC.2004.1297303.