ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Κλιμάκωση του συστήματος αποθήκευσης κλειδιού-τιμής RocksDB με κατανεμημένη αποθήκευση δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Κτιστάκης

Αθήνα, Φεβρουάριος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Scaling the RocksDB key-value store via data distribution on multiple nodes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Κτιστάκης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21/02/2018

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2018

. . . . . .

Παναγιώτης Κτιστάκης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Στην παρούσα διπλωματική εργασία παρουσιάζουμε το σχεδιασμό και την υλοποίηση του crocks, ενός γρήγορου συστήματος αποθήκευσης κλειδιού-τιμής, που κατανέμει ομοιόμορφα τα δεδομένα σε μια συστοιχία υπολογιστικών συστημάτων, και είναι βελτιστοποιημένο για ροές εργασίας που βασίζονται κυρίως στις τυχαίες εγγραφές. Στηριζόμαστε στο υπάρχον σύστημα αποθήκευσης RocksDB, το οποίο δεν υποστηρίζει το διαμοιρασμό δεδομένων. Το RocksDB βασίζεται στα LSM-δέντρα, κάτι που το καθιστά πολύ αποδοτικό και ικανό να αξιοποιεί τους διαθέσιμους υπολογιστικούς πόρους, όσο το δυνατόν καλύτερα. Η εφαρμογή μας καταφέρνει να κλιμακώνεται οριζόντια με την αύξηση του μεγέθους της συστοιχίας, και υποστηρίζει την προσθαφαίρεση κόμβων με ζωντανή μετανάστευση δεδομένων, καθώς και την ασφαλή επαναφορά της συστοιχίας χωρίς απώλεια δεδομένων σε περίπτωση αποτυχίας ενός ή περισσότερων κόμβων.

Λέξεις-κλειδιά

key-value stores, LSM-trees, distributed, scale-out, sharding, migrations

Abstract

In this thesis we present the design and implementation of crocks, a fast key-value store, that distributes data uniformly on a cluster of computer systems, and is optimized for write-intensive workflows that rely mainly on random writes. We base our work on RocksDB, an existing key-value store that is not distributed itself. RocksDB is based on LSM-trees, which makes it perform better, and utilize efficiently the available system resources. Our application is able to scale linearly as the cluster is getting bigger, and supports the addition and removal of nodes with live data migration, as well as recovering the cluster without data loss, in case of node failures.

Keywords

key-value stores, LSM-trees, distributed, scale-out, sharding, migrations

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Νεκτάριο Κοζύρη, για τη δυνατότητα που μου προσέφερε να ασχοληθώ με τον τομέα των Υπολογιστικών Συστημάτων στη διπλωματική μου εργασία. Θέλω ακόμη να ευχαριστήσω τον Βαγγέλη Κούκη, για την άρτια συνεργασία μας, την εμπιστοσύνη του, και την υποστήριξή του, κατά την εκπόνηση αυτής της εργασίας, καθώς και τον Χρήστο Σταυρακάκη, για την υπομονή, τον χρόνο και τις πολύτιμες συμβουλές του. Θα ήθελα τέλος να ευχαριστήσω την οικογένειά μου και τους φίλους μου.

Παναγιώτης Κτιστάκης
Φεβρουάριος 2018

Περιεχόμενα

Extended Abstract

0.1 Introduction

The goal of this thesis, is the design and implementation of a highly efficient, distributed key-value store, optimized for workflows that rely mainly on random writes.

Nowadays, internet services are constantly evolving, with more clients being served and more data collected. This trend has lead to the creation of new databases, aiming at better utilization of the available system resources, as well as distribution on multiple computer systems.

These new types of databases are often not based on the relational model and don't use SQL (Structured Query Language) for managing data. They use simpler — lower level solutions instead, because SQL may overcomplicate the database design and/or show a worse performance. The simplest and most efficient type of a NoSQL or non-relational database, as they are called, is the key-value store. Key-value stores, just store key-addressable values, and make no assumptions about the contents or structure of these values. They can be seen as hash tables, but unlike hash tables they usually store keys in order and have the ability to retrieve ranges of keys.

Storing data on disk is not a trivial job. Normally a data structure like B-trees is used, which may theoretically have a low complexity, but they fail to fully utilize the disk's performance, as they need to store every single chunk of data on a different position [7]. If the database is big and the stored keys are random, this leads to constant cache misses and whole pages need to move from the memory to disk, regardless of the actual size

of the requested write.

To overcome these problems, Facebook has developed RocksDB, a key-value store that relies on a different data structure — LSM-trees. As we will see, RocksDB uses only sequential writes, and as a result, shows a better performance on certain workflows.

However, RocksDB can only be used as a library, and does not allow network communication, let alone distributed storage. On order to achieve that, one would either replace the storage engine of an existing distributed key-value store with it, or develop a new one based on it. We choose the latter, so that no limits are imposed from existing design decisions.

There are some existing distributed databases based on LSM-trees but they are not key-value stores and as a result, they have added complexity with an impact on performance. Existing key-value stores are either not distributed, or not based on LSM-trees. The only exception is ZippyDB, which is also developed by Facebook and relies on RocksDB. However, despite RocksDB being free software, distributed under the licences GPLv2 and Apache, ZippyDB is closed-source.

In this thesis we implement crocks, a fast, distributed and horizontally scalable key-value store that is based on RocksDB, and supports the addition and removal of nodes, with live data migration, as well as safe cluster recovery in case of node failures. The goal is making it as efficient as possible, especially in a random writes workflow, taking into account the limits of the network and of RocksDB itself.

0.2 Background

0.2.1 RocksDB

RocksDB is an embeddable, persistent key-value store, based on LSM-trees (log-structured merge-trees) developed by Facebook. It is written in C++, and is a fork of LevelDB, an earlier key-value store developed by Google.

Architecture

RocksDB uses the following types of files and data structures to create a database.

- **Write Ahead Log (WAL):** An append-only log file, in which RocksDB adds an entry for every write. As soon as a write enters the WAL, it is guaranteed to be persisted. In case of a failure, the database is recovered by replaying each entry of the WAL.

- **MemTable:** An in-memory data structure that contains the same entries as the WAL, but more easily accessible. By default it uses a skip list [4] but there are many alternatives.

- **SSTables (Sorted String Tables):** Immutable, persistent files, that store key-value pairs in ascending key order. They are the main component of a RocksDB database. Along with the key-value pairs, there is also an index stored, that makes looking up a key in the file easier. If there is enough space, indexes are also stored in memory, to reduce disk communication.

- **Levels/MANIFEST file:** Since SSTables are immutable, a key may be included in more than one, and even in the MemTable. However, only the latest value written is valid at any given time. In order to prioritize SSTables, they are organized in levels of the so-called LSM-tree, with lower levels (the ones at the top of the LSM-tree) having priority over higher ones.

  The first level (level 0) contains SSTables that have been created directly from a MemTable when it was flushed to disk. The rest, contain sorted runs of non-overlapping SSTables. For example an SSTable with keys from 'a' to 'b', another one with keys from 'c' to 'd' etc.

When a MemTable gets larger (by default when it reaches 64 MB), it is flushed to disk, creating an SSTable at the top of level 0. The MemTable and the WAL that is backing it, are deleted, and new ones are created to store the next writes.

Each level has a size limit, and when it is reached, some files move to the next level by merging with the overlapping files, and creating new ones that replace them. This procedure is called compaction, and happens in the background. If a key is contained in more than one SSTable, only the newest value stays in the generated files.

The limit of level 0, is 4 files by default. When exceeded, every file in levels 0 and 1 are merged into level 1. The limit of level 1 is 256 MB by default, and each other level has

the limit of the previous level, multiplied by a branching factor which is 10 by default. When the limit of a level greater than 0 is exceeded, one file is chosen and moves to the next level, by merging with overlapping files. More than one compactions may occur simultaneously as long as they don't use overlapping files.

Deletions take place by adding special entries called tombstones for the key, which stay in the database, possibly overwriting earlier entries upon compaction, until reaching the last level where they are deleted forever.

To read the value of a key, RocksDB searches for it in every possible location in priority order and returns as soon as it finds it. The possible locations in order, are the current MemTable, any inactive MemTables that are not yet flushed to disk, each SSTable in level 0, and a specific SSTable in levels 1 through the last, namely the one that has the requested key in its key range.

Performance

In order to evaluate the efficiency of RocksDB, we define a few metrics and calculate their value.

- **Write amplification:** The number of bytes actually written to disk for each byte intended to be written by the user.

  A key-value pair is first written in the WAL, and then in an SSTable in level 0, which brings the write amplification to 2. For the level 0 to level 1 compaction, 4 files are moved to the next level, and as many as 8 new files are created, adding $8/4 = 2$ to the write amplification. For the rest compactions, 1 file is moved to the next level, and produce as many files as the number of overlapping files. With a branching factor of 10, the number of overlapping files is about 7 [1], bringing the total write amplification to $4+7(n-2)$, for a database with n levels including level 0.

- **Read amplification:** The number of pages needed to be read from disk, to retrieve the page that actually contains the requested data. We define it using pages

[1] It's not 10 as one would expect, because firstly SSTables are compressed, and secondly, levels are not uniform and RocksDB chooses the file with the fewer overlapping files for compaction.

and not bytes, because a page is the smallest amount that it is possible to be read (4 KB in a typical architecture).

Since each key can be in each one of the SSTables in level 0, and in one SSTable in each of the other levels, with a database of, say, 6 levels, 9 different pages need to be checked.

This read amplification is mitigated using bloom filters — data structures which are relatively small and fit in memory, and with a 1% chance of a false positive and no chance of a false negative, test whether a key is present in an SSTable. The use of bloom filters practically drops read amplification to about 1.

- **Space amplification:** The number of bytes located on disk, for each byte of real data, at any given time.

  Since each level is larger than the previous by a factor of 10, at the worst case every key is present in the last level, and the other levels contain updated values, which brings space amplification to 1 + 1/10 + ... = 1.111.

  However, because limits are fixed, the last level may be only slightly larger that the one before, and cause a space amplification greater that 2. For this reason, we use a slightly different compaction algorithm, where the limit of the last level is it's actual size, and the limits of the other levels are set as the limit of the next, divided by ten. That way the space amplification is guaranteed to be no more than 1.111.

It is worth noting that RocksDB provides different ways of customizing the LSM-tree structure, with different trade-offs between the kinds of amplification.

Features

Some of the most important features of RocksDB, that we are going to use during the implementation of our application, are the following:

- **Column Families:** Different LSM-trees in the same database, using a shared WAL.
- **Write Batches:** Write multiple keys-value pairs atomically.

- **Snapshots:** Database snapshots, created by increasing the reference counter of each file in the current version of the database, and keeping a copy of the current LSM-tree structure.
- **Iterators:** Iteration in a snapshot of a column family, in ascending or descending key order.
- **SSTable creation/import:** SSTables can be created by hand, and then imported into a column family of a database. They can be added either on top of the LSM-tree, with its keys having priority over the existing ones, or at the bottom with the reverse effect.

0.2.2 Protocol Buffers

Protocol Buffers (protobuf) are a language-neutral mechanism, created by Google, for serializing structured data, quickly and in a compressed form.

The data structure is defined once using the protobuf language, and a compiler generates source code to help easily write and read data, to and from a variety of data streams. The programming language of the generated code is chosen by the user. A protobuf message definition looks like this:

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}

message SearchResponse {
  repeated Result results = 1;
}

message Result {
  string url = 1;
  string title = 2;
  repeated string snippets = 3;
}
```

Listing 1: *Example of a protobuf message definition*

For each type of message, the compiler defines a class in the chosen programming language, that contains a different attribute for each field, as well as some methods for editing and reading fields (getters/setters), serializing and deserializing etc.

0.2.3 gRPC

gRPC is a high performance, RPC framework. It implements the exchange of protobuf messages using the HTTP/2 network protocol.

gRPC extends the protobuf language to allow the definition of RPCs, that take as input and give as output, protobuf messages. Both the input and output can be either a single message, or a stream of messages. An RPC definition looks like this:

```
service RPC {
  rpc Search(SearchRequest) returns (stream SearchResponse) {}
}
```

Listing 2: *RPC definition example*

Like protocol buffers, a dedicated compiler generates code for the chosen programming language. This code provides an API for the implementation of the servers that provide the RPCs, and the clients that call them.

There are two types of APIs, the synchronous and asynchronous. Using the synchronous API, whenever a message is expected from the other end (server/client), the respective function call blocks indefinitely. So on the server side, each call needs a dedicated thread that stays alive but blocked, while awaiting for messages. In order to serve more than one calls simultaneously, the same number of threads are needed. This causes an important overhead.

The asynchronous API on the other hand, uses a completion queue. Requests for network communication are given a unique tag, and the respective functions return immediately. Whenever a message is received, the tag enters the queue by gRPC. So a single thread can be removing tags from the queue and serving requests as needed.

0.2.4 etcd

etcd[2], is a reliable, distributed key-value store, for the most critical data of a distributed system. It provides a simple and well-defined gRPC API and is secure and fast. It guarantees consistency by using the Raft [12] consensus algorithm.

Apart from the common operations of a key-value store (get, put, delete), etcd has many more features. We will use key watching and transactions.

By watching a key, a client is able to be notified whenever one or more keys are modified, without having to periodically make requests. It is a gRPC procedure call, that takes as input a protobuf message with the desired keys, and returns a stream of messages with the updated key-value pairs.

Transactions, allow making atomic read-modify-write operations using compare-and-swap. Entries are updated, provided that the values have not been updated since last read. If they have, the whole procedure is repeated until it succeeds.

0.3 Design

We present the main design decisions of our application.

0.3.1 Sharding

The key space needs to be somehow partitioned, and have the different partitions evenly distributed in the cluster. We will also have to provide a mechanism for sending a portion of a node's database to another node, when for example one is added to the cluster. When that happens, the node that sent the keys, will have to delete them and deleting large portions of a database is expensive because of all the tombstones. For that reason, we will partition a node's database in shards, which is the smallest possible part that can move from one node to another, and use a different column family for each shard. Column families can be created and dropped on demand, without any overhead.

[2] `https://github.com/coreos/etcd`

Using multiple column families has an additional benefit. Each column family uses its own LSM-tree, so each shard uses fewer levels, and the write amplification falls significantly.

We partition the database in shards using a hash function by mapping each key to the shard `hash(key) modulo num_shards`. In that way the uniformity of the shards is easily guaranteed, something that would be almost impossible if we were using, say, ranges, because smaller ranges would have to be constantly transfered from one shard to another, in order to keep them balanced.

The hash function doesn't need to be cryptographically secure. It only needs to be fast and provide uniform results. We chose a variation of `MurmurHash` that is also used internally by RocksDB.

0.3.2 Architecture

Since we want to scale indefinitely, having nodes with certain roles, such as a proxy/-coordinator is not a good idea, so instead we have clients connect and communicate with each node independently. If having clients connect through a proxy, is needed for a certain application, it could be implemented, using the provided client API.

The necessary information (basically each node's address and shards), is stored at all times in an etcd cluster, and nodes stay updated by watching etcd for changes. Clients are also informed using etcd, but the don't watch it constantly, to mitigate network traffic. Instead, they request the updated information, only when they receive a response from a node, informing that he is not the master of the relevant shard.

0.3.3 Cluster states

The cluster during its lifetime passes through the stages of `INIT`, `RUNNING` and `MIGRATING`. In the `INIT` state, the initial nodes are set up and they announce their addresses to etcd. Once everything is ready, the administrator distributes the available shards to the different nodes, and sets the state to `RUNNING`, in order to start serving client requests.

A client that intends to make a request, first receives the cluster information from etcd.

Then he finds the shard that corresponds to the request's key, and sends the request to the master of the shard, like this:

```
shard_id = hash(key) mod num_shards
shard = shards[shard_id]
node = shard.master
node.put(key, value)
```

Listing 3: *Put*

If the master has changed, the node sends the corresponding status, and the client gets the updated information from etcd, and repeats the request. If not, the node serves the request and returns the status reported by RocksDB.

While in this state, the administrator can set up new nodes and/or request existing nodes to shut down. The new nodes are not assigned any shards yet. When ready, he redistributes the shards to the available nodes to make them uniform, with the fewer possible number of transfers, and requests migrations to begin by setting the cluster state to `MIGRATING`.

In the `MIGRATING` state, each node that needs to get new shards, according to the updated info, starts requesting them one by one from the former shard masters. The former master, when requested, stops accepting requests for the shard, passes the mastership to the new node on etcd, turns the column family of the shard into a single sorted run of SSTables, and starts sending them.

The receiving node, receives these files and adds them one by one into the database, on the bottommost level, in order to have the lowest possible priority. While importing a shard, he serves write requests normally. Read requests require special care. If a key is not found in the database it may or may not exist on the former master. At any given time, the largest key imported is known, so if the requested key is in the range of the imported keys, the node can safely assume that it does not exist, and responds accordingly. If not, he passes the request to the former master and forwards his response.

0.4 Implementation

crocks consists of three different parts:

- The executable `crocks` which is the server — a cluster node.
- The shared or static library `libcrocks.so` or `libcrocks.a` respectively, and the relevant public header files that provide an API for the implementation of clients.
- The executable `crocksctl`, a tool that makes read/write requests to the cluster, and some administrative operations.

0.4.1 Shared data

The shared data stored in etcd, is a protobuf message with the following definition.

```
message NodeInfo {
  string address = 1;
  int32 id = 2;
  int32 num_shards = 3;
  bool available = 4;
  bool remove = 5;
}

message ShardInfo {
  int32 master = 1;
  bool migrating = 2;
  int32 from = 3;
  int32 to = 4;
}

message ClusterInfo {
  enum State {
    INIT = 0;
    RUNNING = 1;
    MIGRATING = 2;
  }
  State state = 1;
  int32 num_nodes = 2;
  repeated NodeInfo nodes = 3;
  repeated ShardInfo shards = 4;
}
```

Listing 4: *Ορισμός ClusterInfo*

It is stored serialized, on the key “info” which is being watched by the nodes, and updated using a transaction. For example, to order the migration of a shard from node 1 to 2, the administrator sets the ‘migrating’ field to true, ‘from’ to 1 and ‘to’ to 2.

0.4.2 Server

The server uses the asynchronous gRPC API. For each different RPC (get, put, migrate etc.) we define a corresponding class, all of which implement a `Proceed()` method that progresses the request as much as possible, until some network communication is required, which it requests asynchronously and returns.

Each serving thread has its own gRPC completion queue. It creates one instance for each RPC class, and uses the memory address of the `Proceed()` method as a unique tag, to differentiate the different objects. Once a request arrives, gRPC puts the tag in the queue, the serving thread removes it and calls the corresponding method. In `Proceed()`, a new instance is created to handle the next request. Then the protobuf message with the request is read and served. If the RPC is unary, a response is sent, and if it’s streaming, `Proceed()` returns until the next message is received and the tag turns up in the queue again. The RPC objects remember where they’re left off using a C++ `enum`, as a state machine.

There are multiple serving threads, chosen by the administrator via a command-line option, while serving migrations happens in a different dedicated thread, because it is expected to block a lot, doing IO. Thread-safety on data structures that are not thread-safe by default, is guaranteed using mutexes (`std::mutex`). Shared mutexes (`std::shared_mutex`), can also be used if available, which results in multiple threads taking the locks as readers, without blocking, as long as migrations are not taking place, and there is no one modifying the underlying structures.

In order to guarantee that there are no race conditions during migrations, we are using reference counters on shards. We dump the shard to disk using an iterator that operates on a consistent snapshot of the column family, so all pending writes must finish before creating it. The reference counter is initialized to 1, and gets incremented before writing, and decremented afterwards. When the node stops accepting requests, he decrements the counter and waits for it to reach 0.

0.4.3 Client

The provided client API, is located in the public headers, and contains some structures in the `crocks` namespace. The main structure, is the `Cluster` class, defined in `<crocks/cluster.h>` header file.

```
Cluster(const Options& options, const std::string& address);
```

The `Options` class is defined in `<crocks/options.h>` header file, and allows some customization in the client's behavior, while the `address` string is the address of an etcd node, (e.g. `"127.0.0.1:1234"`).

```
struct Options {
  // If true, when a node is detected to be unavailable
  // and etcd is not yet aware, the client updates it.
  bool inform_on_unavailable = true;

  // If true, after a status UNAVAILABLE is received, the client waits
  // until the cluster is healthy again, and then retries the request.
  bool wait_on_unhealthy = true;
};
```

The `Cluster` class provides methods `get()`, `put()`, and `delete()`, for reading, writing and deleting keys respectively.

```
Status Get(const std::string& key, std::string* value);
Status Put(const std::string& key, const std::string\& value);
Status Delete(const std::string& key);
```

The `Status` class is defined in `<crocks/status.h>` header file and contains the composition of the gRPC and RocksDB status classes.

0.4.4 Write batch implementation

Write batches are defined in `<crocks/write_batch.h>`.

```
class WriteBatch {
 public:
  WriteBatch(Cluster* db);
  WriteBatch(Cluster* db, int threshold_low, int threshold_high);
  ~WriteBatch();
```

```
  void Put(const std::string& key, const std::string& value);
  void Delete(const std::string& key);
  void Clear();

  Status Write();
}
```

Listing 5: *The write batch client API*

They are implemented by sending buffers with operations, asynchronously and transparently, while the client is requesting them. gRPC allows one pending write per call, so after each operation, if there is no pending write, the current buffer is sent, unless the buffer has gotten significantly larger (exceeded a configurable threshold), where the call blocks until the buffer is able to be sent.

This leads to a very high performance. However, unlike in RocksDB, write batches are not atomic, because they can fail on one node and succeed on others, and rolling back is not supported. Furthermore, write batches from different clients can reach the nodes in a different order, and leave the database in an inconsistent state. So they should be used carefully, in cases where the included keys are not strictly related, or they are guaranteed to not be overwritten by other clients.

0.4.5 crocksctl

`crocksctl` is a command-line application, that takes a command and after fetching the cluster info from etcd, executes it. The available commands with a brief description, can be seen on the output of the `--help` option, below.

```
Usage: crocksctl [options] command [args]...

A simple command line client for crocks.

Commands:
  get <key>          Get key.
  put <key> <value>  Put key.
  del <key>          Delete key.
  run                Change cluster state to RUNNING.
  migrate            Change cluster state to MIGRATING.
```

```
  health              Check the health of the cluster.
  list                Print every key.
  dump                Print every key—value pair.
  clear               Delete all keys.
  remove <id>         Remove node from the cluster.
  info                Print cluster info.

Options:
  —e, ——etcd <address>  Etcd address [default: localhost:2379].
  —h, ——help            Show this help message and exit.
```

Listing 6: *`crocksctl --help` output*

0.4.6 Recovery from failures

The cluster can safely recover from node failures. During a failure the 'available' field of the node in etcd, is set to false, and the cluster is unresponsive. The etcd update is made by a client, if a request fails with the relevant status and he is configured to do so, by another node, if the failure takes place during a migration, or by the administrator, using the `crocksctl health` command.

When a client request fails, it can either watch the info key on etcd until the cluster is stable again, or return with the relevant status, and let the client handle the situation accordingly, and then wait manually using the `void Cluster::WaitUntilHealthy()` method.

When a node, upon setup, notices an unavailable node in etcd with his own address, he assumes that he must recover from a failure. He checks that his database contains a column family for each shard he is supposed to have, and changes his availability back to true.

In case there is a failure during a migration, after the recovery, the receiving node will request the sending node to continue the migration from where it left off. For that reason, there are some metadata kept in the database. The receiving node keeps the filename of the last SSTable of the shard that was safely received and stored on disk, its largest key, and the number of the next SSTable he needs to request. When he recovers, he makes sure that the column family exists, imports the last received SSTable, if he

hasn't already, and starts a new call for the migration, starting from the next SSTable. The sending node keeps whether he managed to create all SSTables, and if so, their number and each one's largest key.

0.5 Experimental evaluation

For the experiments, we used virtual machines from AWS [3] (Amazon Web Services).

The first benchmarks were done using two VMs, with 1 Gbps network throughput, 1 GB RAM, and an 8 GB disk with a write throughput of 65 MB/sec each. Initially, one was configured as a cluster of one node, and the other as a client. Etcd was also located on the client's VM.

We used a 2 GB database using a fixed number of 16 byte keys, and random 4 KB values. First, we filled the database using sequential keys and the write batch API, and then we performed random writes, random reads, and random reads and writes simultaneously, using a gradually increasing number of threads.

Filling the database needed 70 seconds, so the throughput was 2 GB / 70 sec $\simeq$ 30 MB/sec, which was expected because the files were not overlapping and no compaction was needed, so we had a write amplification of 2 (1 for the WAL and one for the SSTable).

On full utilization, RocksDB reported a write amplification of 3.5, and on random writes we measured 20 MB/sec throughput, which is $\simeq$ 65 / 3.5. Random reads were slightly better, at 26 MB/sec and simultaneous reads and writes were at 9 MB/sec each. When we let compactions finish and tried doing writes in small bursts, they used the full disk throughput (65 MB/sec) as expected. Finally, we tried adding a second node on the client, and the performance on every benchmark doubled.

Then we experimented with better virtual machines, with a cluster size of 1 to 16 nodes. Each node had a 1 Gbps network throughput with an average latency of 450 μsec, and 2, 40 GB SSDs at 200 MB/sec each, which were configured in RAID 0, and acted as a single 80 GB disk with 400 MB/sec throughput. The client had a network throughput of 10 Gbps, and the write amplification was measured at 4, so the maximum insert

[3] `https://aws.amazon.com/`

throughput was 400 / 4 = 100 MB/sec. This value is comparable to the network's 1 Gbps = 125 MB/sec, so the disk was no longer the bottleneck. We used three different byte sizes, small (128 bytes), medium (4 KB), and large (256 KB).

Latency was shown to be independent of the cluster size. With small values, it started at 440 μsec — at the actual network latency for small and medium values with a comparable P_{99} latency [4]. However, the maximum latency was found to be larger by 2–3 orders of magnitude, because RocksDB uses some locks, and sometimes needs to block for a while before taking them.

	128 Bytes	4 KB	256 KB
min	440	480	1500
P_{99}	750	850	6750
max	200000	200000	300000

Table 1: *Latency (μsec)*

With small byte values, writes started at about 19000 IOPS with a cluster of 1 node, and scaled to 240000 IOPS ($\simeq$ 35 MB/sec) with 16 nodes. Medium values had comparable results with 12000 to 130000 ($\simeq$ 500 MB/sec), while in large values the network throughput was the bottleneck. The client had 10 times the throughput of the nodes, so up to 10 nodes, IOPS scaled linearly and then it peaked at about 10 Gbps (1.4 GB/sec).

Reads had the same overall behavior, with a somewhat worse performance (50% on small values, 70% on medium, while on large values the bottleneck was once again the network).

On simultaneous reads and writes, we had the same number of reader and writer threads, each of which had the same performance. The performance reached 10 MB/sec on small values, 200 MB/sec on medium, and over 1 GB/sec on large values.

Finally, writes using write batches were by far the fastest. On small values, the client was unable to generate more than 1100000 values per second ($\simeq$ 145 MB/sec). The performance peaked at 8 nodes and remained steady thereafter. Both medium and large values managed to utilize the full network throughput, by reaching 1.4 GB/sec (278000 and 5730 IOPS respectively).

[4] The upper bound of latencies experienced by 99% of requests.

The relevant graphs can be seen on section 5.2.2.

0.6 Conclusion and future directions

In this thesis, we designed and implemented a high performance, distributed key-value store, that scales linearly as the cluster is getting bigger. The performance depends on the specs of the nodes and the network, as well as the size of the database, and is determinate and easily calculated in advance.

While benchmarking, we noticed a very low latency in most requests, and managed to perform random writes with 1.4 GB/sec (the network's limit), using a cluster of just 10 nodes.

However our system is far from complete. Some of the most important missing feature are:

- **Redundancy and failover:** Currently, a node failure may not cause data loss, but makes the system unresponsive until the node is recovered by the administrator. This problem could be solved by implementing a replication mechanism, so that each shard is located on more than one nodes, and on failure clients are redirected to the one with the replicated shard.
- **Scaling beyond the initial number of shards:** When the initial number of shards per node is for example 10, the cluster can only get 10 times larger. There should be a mechanism, to redistribute the keys in a larger number of shards, and be able to add even more nodes to the cluster.
- **Implementation of the remaining RocksDB features:** RocksDB, has many more features that would be useful, the most important one being transactions. Transactions in RocksDB, implement the 2PC protocol [15], making it possible to implement them in a distributed application. By doing so, we will be able to have truly atomic write batches.

1
Εισαγωγή

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός και η υλοποίηση ενός αποδοτικού, κατανεμημένου συστήματος αποθήκευσης κλειδιού-τιμής, για ροές εργασίας βασισμένες κυρίως στις τυχαίες εγγραφές.

Στην εποχή των τεχνολογιών υπολογιστικού νέφους (cloud computing) και των δεδομένων μεγάλης κλίμακας (big data), οι διαδικτυακές υπηρεσίες συνεχώς επεκτείνονται, εξυπηρετώντας όλο και περισσότερους πελάτες και συλλέγοντας όλο και περισσότερα δεδομένα.

Ο εκθετικά αυξανόμενος όγκος δεδομένων που πρέπει να διαχειρίζεται ένα σύγχρονο κέντρο δεδομένων, έχει οδηγήσει στην δημιουργία νέων βάσεων δεδομένων, που εστιάζουν στην αποδοτικότερη αξιοποίηση των φυσικών πόρων που παρέχει ένα υπολογιστικό σύστημα, καθώς και στην κατανομή των δεδομένων σε περισσότερα υπολογιστικά συστήματα, όταν αναπόφευκτα προκύψει αυτή η ανάγκη.

Οι βάσεις δεδομένων αυτές, συχνά δεν ακολουθούν την σχεσιακή δομή των SQL βάσεων, αφενός επειδή απευθύνονται σε ροές εργασίας που δεν απαιτούν κάτι τέτοιο, και αφετέρου γιατί αυτό τις καθιστά αποδοτικότερες. Η πιο απλή και ταυτόχρονα αποδοτική μορφή NoSQL ή μη σχεσιακής βάσης δεδομένων όπως ονομάζονται, είναι τα key-value stores (συστήματα αποθήκευσης κλειδιού-τιμής). Λειτουργούν σε χαμηλότερο επίπεδο, αποθηκεύοντας απλά τιμές σε κλειδιά, χωρίς να κάνουν κάποια υπόθεση σχετικά με τη δομή και το περιεχόμενο των τιμών αυτών. Συμπεριφέρονται επομένως σαν πίνακες κατακερματισμού, με τη διαφορά ότι συνήθως αποθηκεύουν τα κλειδιά ταξινομημένα, επιτρέποντας έτσι την προσπέλαση όχι μόνο μεμονωμένων, αλλά και εύρους κλειδιών.

Η αποδοτική αποθήκευση τυχαίων δεδομένων στο δίσκο με ταξινομημένο τρόπο, είναι δύσκολη. Αυτό παραδοσιακά υλοποιείται με τη βοήθεια κάποιας δομής δεδομένων όπως τα B-δέντρα, τα οποία ενώ θεωρητικά παρουσιάζουν καλή πολυπλοκότητα, αδυνατούν να αξιοποιήσουν τις δυνατότητες των δίσκων, καθώς απαιτούν την αποθήκευση της κάθε εγγραφής σε διαφορετική θέση [7]. Αυτό για τυχαίες εγγραφές σε μεγάλες βάσεις, προκαλεί συνεχώς αστοχίες κρυφής μνήμης, και μεταφορά ολόκληρων σελίδων στο δίσκο ανεξάρτητα από το μέγεθος της κάθε εγγραφής.

Σε μια προσπάθεια να ξεπεραστούν αυτά τα προβλήματα, έχει προταθεί μια εναλλακτική δομή δεδομένων, τα LSM-δέντρα [2], τα οποία αποθηκεύουν δεδομένα στο δίσκο πραγματοποιώντας μόνο ακολουθιακές εγγραφές. Το πιο γνωστό και ταυτόχρονα αποδοτικό σύστημα αποθήκευσης κλειδιού-τιμής βασισμένο στα LSM-δέντρα, είναι το RocksDB, το οποίο αναπτύσσεται από την εταιρία Facebook, και όπως θα δούμε στην παράγραφο 2.1 παρουσιάζει σημαντικά καλύτερη απόδοση από τις συνηθισμένες μηχανές αποθήκευσης, σε συγκεκριμένες ροές εργασίας.

Όμως το RocksDB είναι απλά μια βιβλιοθήκη και δεν υποστηρίζει δικτυακή επικοινωνία, πόσο μάλλον κατανεμημένη αποθήκευση δεδομένων. Για να επιτευχθεί κάτι τέτοιο, απαιτείται είτε να αντικαταστήσει την υπάρχουσα μηχανή αποθήκευσης σε μια συγκεκριμένη κατανεμημένη βάση δεδομένων, ή να επεκταθεί, κατασκευάζοντας μια νέα βασισμένη σε αυτό. Εμείς θα ακολουθήσουμε τη δεύτερη κατεύθυνση, για μεγαλύτερη ελευθερία, καθώς και για να μην μπαίνουν όρια στην απόδοση, από υπάρχουσες σχεδιαστικές αποφάσεις.

Υπάρχουν διάφορες υπάρχουσες εφαρμογές που είναι αρκετά κοντά στο ζητούμενο.

- Τα Memcached[1] και Redis[2], είναι κατανεμημένα key-value stores, όμως είναι σχεδιασμένα για να αποθηκεύουν δεδομένα στη μνήμη και όχι στο δίσκο.
- Το Couchbase[3] είναι κατανεμημένο key-value store που διατηρεί τα δεδομένα στο δίσκο, αλλά βασίζεται σε B-δέντρα και όχι σε LSM-δέντρα.
- Τα Apache Hbase[4] και Cassandra[5] είναι κατανεμημένες βάσεις δεδομένων βασι-

[1] https://memcached.org/
[2] https://redis.io/
[3] https://www.couchbase.com/
[4] https://hbase.apache.org/
[5] https://cassandra.apache.org/

σμένες στα LSM-δέντρα, αλλά δεν αποτελούν key-value stores. Αντίθετα, χρησιμοποιούν πίνακες, και παραπέμπουν περισσότερο σε SQL βάσεις δεδομένων.

- Ακόμη υπάρχει το CockroachDB[6], το οποίο είναι κάτι περισσότερο από αυτό που ζητάμε καθώς αποτελεί SQL βάση δεδομένων, αλλά στη βάση του είναι ένα κατανεμημένο σύστημα αποθήκευσης κλειδιού-τιμής βασισμένο στο RocksDB. Το βασικό του μειονέκτημα είναι ότι χρησιμοποιεί τη μέθοδο MVCC[7] (Multiversion concurrency control). Δηλαδή όταν γίνει εκ νέου εγγραφή ενός υπάρχοντος κλειδιού, δεν διαγράφεται η παλιά τιμή αλλά προστίθεται με διαφορετική χρονοσφραγίδα (timestamp). Αυτό επιτρέπει στο χρήστη να εκτελεί αναζητήσεις στη βάση, για μια συγκεκριμένη χρονική στιγμή στο παρελθόν, κάτι που μπορεί να φανεί πολύ χρήσιμο, αλλά έχει μεγάλη επίπτωση στην απόδοση.

- Η μόνη εφαρμογή που πληροί όλες τις ζητούμενες προδιαγραφές, είναι το ZippyDB[8], το οποίο αναπτύσσεται επίσης από τη Facebook, χρησιμοποιεί το RocksDB, και δεν αποτελεί τίποτα παραπάνω από ένα απλό, γρήγορο και ασφαλές κατανεμημένο key-value store. Όμως, ενώ το RocksDB είναι ελεύθερο λογισμικό και διατίθεται με τις άδειες GPLv2 και Apache 2.0, το ZippyDB είναι κλειστού κώδικα και δεν είναι διαθέσιμο στο ευρύ κοινό.

Στα πλαίσια αυτής της διπλωματικής εργασίας θα υλοποιήσουμε το crocks, ένα κατανεμημένο και οριζόντια κλιμακώσιμο σύστημα αποθήκευσης κλειδιού-τιμής, που βασίζεται στο RocksDB και υποστηρίζει την προσθαφαίρεση κόμβων με ζωντανή μετανάστευση δεδομένων, καθώς και την ασφαλή επαναφορά της συστοιχίας σε περίπτωση αποτυχίας ενός ή περισσότερων κόμβων. Στόχος μας είναι να επιτύχουμε όσο το δυνατόν καλύτερη απόδοση, ειδικά στις τυχαίες εγγραφές, λαμβάνοντας υπόψιν μας τους περιορισμούς που επιφέρει η ταχύτητα και η καθυστέρηση του δικτύου, αλλά και το ίδιο το RocksDB.

Μια τέτοιου είδους εφαρμογή ιδανικά θα πρέπει να επιτυγχάνει υψηλή διαθεσιμότητα, υλοποιώντας μηχανισμούς υπερεπάρκειας και αυτόματης μετάπτωσης. Δηλαδή τα ίδια δεδομένα θα πρέπει να βρίσκονται ταυτόχρονα σε περισσότερους από έναν κόμβους,

[6] https://www.cockroachlabs.com/
[7] https://en.wikipedia.org/wiki/Multiversion_concurrency_control
[8] https://www.youtube.com/watch?v=DfiN7pG0D0k

έτσι ώστε σε περίπτωση αποτυχίας του ενός, να παίρνει αυτόματα τη θέση του ένας άλλος. Κάτι τέτοιο όμως, το αναβάλλουμε για μελλοντική επέκταση.

2 Υπόβαθρο

Στο κεφάλαιο αυτό περιγράφουμε αναλυτικά τον τρόπο λειτουργίας και την απόδοση του RocksDB. Επίσης κάνουμε μια σύντομη περιγραφή των υπόλοιπων εργαλείων που θα χρησιμοποιήσουμε για την ανάπτυξη της εφαρμογής μας.

2.1 RocksDB

Το RocksDB [1] είναι ένα σύστημα αποθήκευσης κλειδιού-τιμής του οποίου η λειτουργία βασίζεται στα LSM-δέντρα [2] ή log-structured merge-trees, δηλαδή δέντρα συγχώνευσης δομημένα από αρχεία καταγραφής. Είναι ενσωματωμένο, δηλαδή χρησιμοποιείται ως βιβλιοθήκη και όχι επικοινωνώντας με κάποιον διακομιστή. Αναπτύσσεται από τη Facebook, και βασίζεται στο σύστημα αποθήκευσης LevelDB της Google [3]. Όπως και το LevelDB, είναι γραμμένο στην γλώσσα προγραμματισμού C++.

Μπορεί να χρησιμοποιηθεί είτε ως αυτόνομο σύστημα αποθήκευσης (standalone key-value store), είτε ως μηχανή αποθήκευσης (storage engine) σε βάσεις δεδομένων υψηλότερου επιπέδου. Το δεύτερο έχει πραγματοποιηθεί με επιτυχία στις εφαρμογές MongoRocks[1] και MyRocks[2], που αποτελούν εκδόσεις των βάσεων δεδομένων MongoDB[3] και MySQL[4] αντίστοιχα.

[1] http://mongorocks.org/
[2] http://myrocks.io/
[3] https://www.mongodb.com/
[4] https://www.mysql.com/

2.1.1 Δομικά στοιχεία

Πριν αναλύσουμε τον τρόπο λειτουργίας του RocksDB θα περιγράψουμε τα διαφορετικά είδη αρχείων και δομών δεδομένων που χρησιμοποιεί.

Ημερολόγιο προγενέστερης εγγραφής

Το ημερολόγιο προγενέστερης εγγραφής (Write Ahead Log — WAL) είναι ένα αρχείο καταγραφής, στο οποίο προστίθεται οποιαδήποτε εγγραφή πραγματοποιείται στη βάση (put, delete κτλ). Σε περίπτωση αποτυχίας, όταν το σύστημα επανέλθει, όλες οι εγγραφές του WAL επαναλαμβάνονται, και η βάση επανέρχεται στην αρχική κατάσταση.

Οι εγγραφές στο WAL πραγματοποιούνται ατομικά, δηλαδή είτε δεν πραγματοποιούνται καθόλου είτε ολοκληρώνονται πλήρως και τα δεδομένα αποθηκεύονται μόνιμα και με ασφάλεια στον δίσκο, χωρίς να υπάρχει κίνδυνος παραφθοράς.

MemTable

Στη δομή αυτή, η οποία βρίσκεται στη μνήμη, αποθηκεύονται όλες οι εγγραφές παράλληλα με το WAL, αλλά με πιο δομημένο τρόπο και είναι πιο εύκολα προσβάσιμες. Για την αποθήκευση δεδομένων χρησιμοποιεί skip list [4], αλλά εναλλακτικά μπορεί να χρησιμοποιηθεί το standard vector της C++, hash table με απλά συνδεδεμένες λίστες ή skip lists στα κελιά ή cuckoo hash table. Επίσης επιτρέπει στον προγραμματιστή να κατασκευάσει μια δική του υλοποίηση, δηλαδή μια κλάση που υλοποιεί τις αντίστοιχες μεθόδους.

Οι skip lists έχουν την ίδια πολυπλοκότητα μέσης περίπτωσης με τα δυαδικά δέντρα, όπως φαίνεται στον παρακάτω πίνακα. Όμως είναι προτιμότερες καθώς δεν απαιτούν εξωτερικό συγχρονισμό στις εισαγωγές στοιχείων [5].

Επίσης παρά την αυξημένη πολυπλοκότητα τους υπερτερούν έναντι των hash tables καθώς διατηρούν τα στοιχεία ταξινομημένα, κάτι που καθιστά εύκολη τη μεταφορά τους στον δίσκο όπως θα δούμε παρακάτω. Τέλος θα πρέπει να σημειώσουμε ότι το μέγεθος των MemTables δεν μπορεί να ξεπεράσει ένα προκαθορισμένο όριο κάτι που καθιστά την πολυπλοκότητα πρακτικά σταθερή.

	Average	Worst Case
Space	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Πίνακας 2.1: *Πολυπλοκότητα skip list*

SSTables

Τα SSTables (Sorted String Tables — Πίνακες Ταξινομημένων Συμβολοσειρών), αποτελούν τα βασικά δομικά στοιχεία μιας βάσης δεδομένων του RocksDB. Είναι αρχεία που περιέχουν συνεχόμενα ζεύγη κλειδιού-τιμής ταξινομημένα σε αύξουσα σειρά με βάση το κλειδί.

Από τη στιγμή που γράφεται στο δίσκο, ένα SSTable δεν υφίσταται καμία επεξεργασία. Επίσης για να είναι δυνατό να διαβαστεί κάποια καταχώρηση ενός SSTable, μαζί με αυτές αποθηκεύεται και ένα ευρετήριο με αναφορές στα κλειδιά. Τα ευρετήρια αυτά βρίσκονται ταυτόχρονα και στην μνήμη, εφόσον υπάρχει ο απαιτούμενος χώρος, έτσι ώστε να μην χρειάζεται να ανακτώνται κάθε φορά που πρόκειται να διαβαστεί μια τιμή.

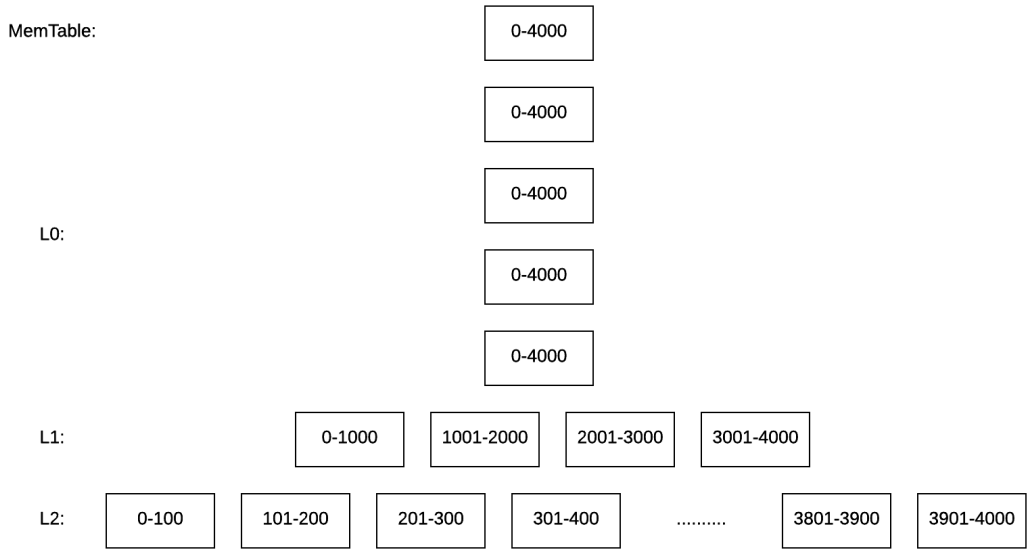
Επίπεδα/Αρχείο `MANIFEST`

Το ίδιο κλειδί μπορεί να βρίσκεται σε περισσότερα από 1 SSTables ή/και στο MemTable. Μία μόνο τιμή όμως είναι έγκυρη ανά πάσα στιγμή, αυτή που καταχωρήθηκε πιο πρόσφατα. Για να γνωρίζουμε ποια είναι αυτή, τα SSTables είναι οργανωμένα σε διαφορετικά επίπεδα, τα οποία απαρτίζουν το LSM-δέντρο. Όσο μικρότερο είναι το επίπεδο (όσο πιο κοντά δηλαδή στην κορυφή του LSM-δέντρου), τόσο πιο πρόσφατα είναι τα κλειδιά και άρα έχουν μεγαλύτερη προτεραιότητα.

Το πρώτο επίπεδο (το επίπεδο 0), περιλαμβάνει SSTables που κατασκευάστηκαν απευθείας από MemTable, και το καθένα καλύπτει όλο το εύρος των κλειδιών. Το καθένα από τα υπόλοιπα επίπεδα, περιλαμβάνει μια σειρά από SSTables, που είναι ταξινομημένα μεταξύ τους και δεν παρουσιάζουν επικάλυψη.

Στον κατάλογο που αποθηκεύονται τα SSTables και το WAL, υπάρχει ακόμα ένα αρχείο που ονομάζεται MANIFEST, και περιέχει πληροφορίες για τη δομή του LSM-δέντρου, όπως τα αρχεία που περιλαμβάνει το κάθε επίπεδο και σε ποιες θέσεις βρίσκονται.

Μια πιθανή μορφή του LSM-δέντρου, για μια βάση δεδομένων του RocksDB που περιλαμβάνει κλειδιά από το “0” έως το “4000” είναι η εξής:



Σχήμα 2.1: *Παράδειγμα LSM-δέντρου*

2.1.2 Λειτουργία

Εγγραφές

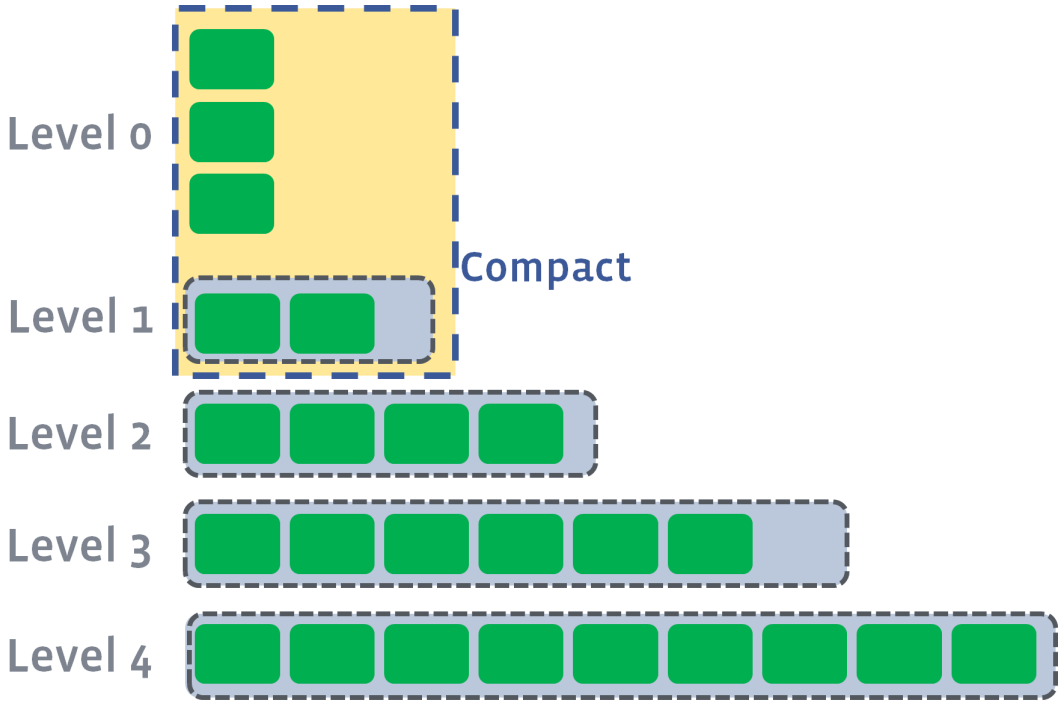
Όταν ζητηθεί από το RocksDB να πραγματοποιήσει κάποια εγγραφή (εισαγωγή ή διαγραφή), αυτή προστίθεται στο τέλος του WAL και στην κατάλληλη θέση του MemTable, και η αντίστοιχη κληθείσα συνάρτηση επιστρέφει. Αν το σχετικό κλειδί υπήρχε ήδη στο MemTable, ή προηγούμενη εγγραφή αντικαθίσταται από τη νέα. Όλες οι υπόλοιπες λειτουργίες που θα περιγράψουμε στη συνέχεια πραγματοποιούνται σε ξεχωριστά νήματα στο παρασκήνιο, και δεν γίνονται αντιληπτές από το χρήστη.

Όταν το μέγεθος του MemTable ξεπεράσει ένα συγκεκριμένο μέγεθος (64 MB από προεπιλογή), μετατρέπεται σε SSTable και γράφεται στο δίσκο στο επίπεδο 0 του LSM-δέντρου. Στη συνέχεια δημιουργούνται νέο MemTable και WAL για τις επόμενες εγ-

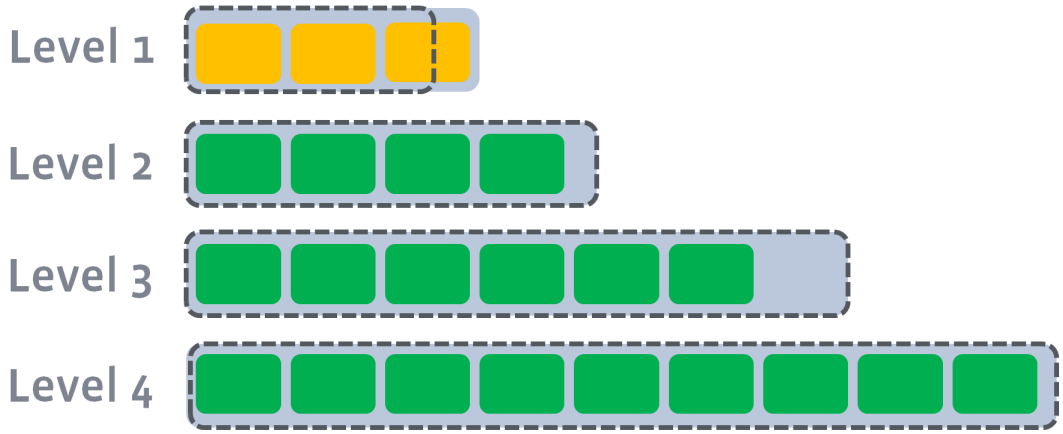
γραφές. Η διαδικασία αυτή επαναλαμβάνεται με το κάθε νέο SSTable να προστίθεται στην κορυφή του επιπέδου 0, ώστε να έχει προτεραιότητα έναντι αυτών που γράφτηκαν νωρίτερα, αφού περιέχει πιο πρόσφατες εγγραφές.

Όταν τα SSTables του επιπέδου 0, ξεπεράσουν ένα όριο (4 από προεπιλογή), πραγματοποιείται η διαδικασία της συμπύκνωσης (compaction). Δηλαδή συγχωνεύονται, δημιουργώντας νέα SSTables και μεταβαίνουν στο επίπεδο 1. Αν δύο η περισσότερα SSTables περιλαμβάνουν το ίδιο κλειδί, κατά τη συγχώνευση διατηρείται μόνο η τιμή που γράφτηκε πιο πρόσφατα. Τα αρχεία που δίνονται ως είσοδος στη συμπύκνωση, με το πέρας της διαδικασίας, παύουν να ανήκουν στο LSM-δέντρο ή τουλάχιστον στην τελευταία έκδοσή του.

Επειδή ενδέχεται να υπάρχει κάποιο στιγμιότυπο της βάσης (βλ. παράγραφο 2.1.4) που τα χρησιμοποιεί, δεν διαγράφονται αμέσως. Αντίθετα έχουν ένα μετρητή αναφορών (reference counter) ο οποίος μειώνεται, και ένα νήμα που αναλαμβάνει το ρόλο του συλλέκτη σκουπιδιών (garbage collector), ελέγχει περιοδικά τα αρχεία και διαγράφει όσα δεν έχουν καμία αναφορά. Όταν το επίπεδο 0 γεμίσει ξανά, όλα τα αρχεία των επιπέδων 0 και 1 συγχωνεύονται και αντικαθιστούν αυτά του επιπέδου 1.

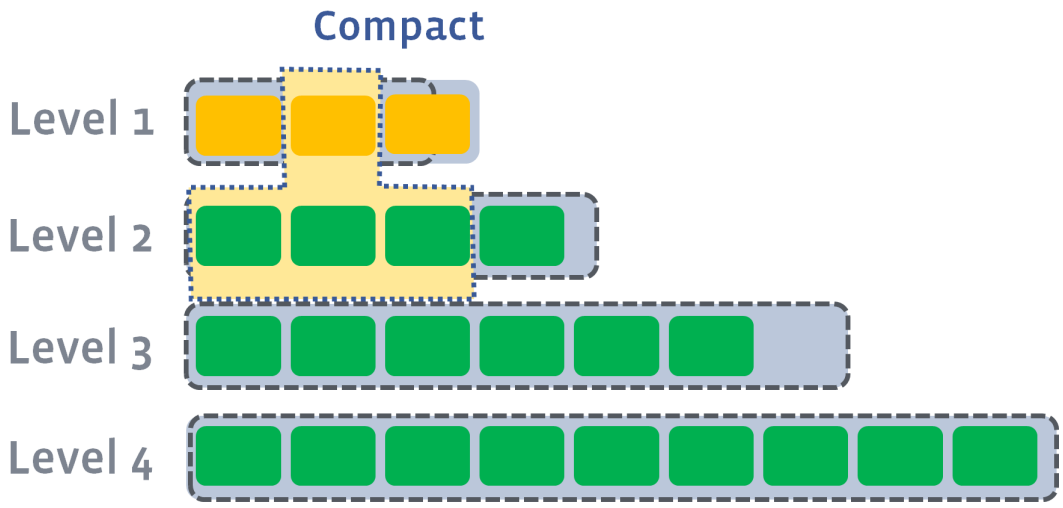


Σχήμα 2.2: *Επιλογή αρχείων για συμπύκνωση του επιπέδου 0*

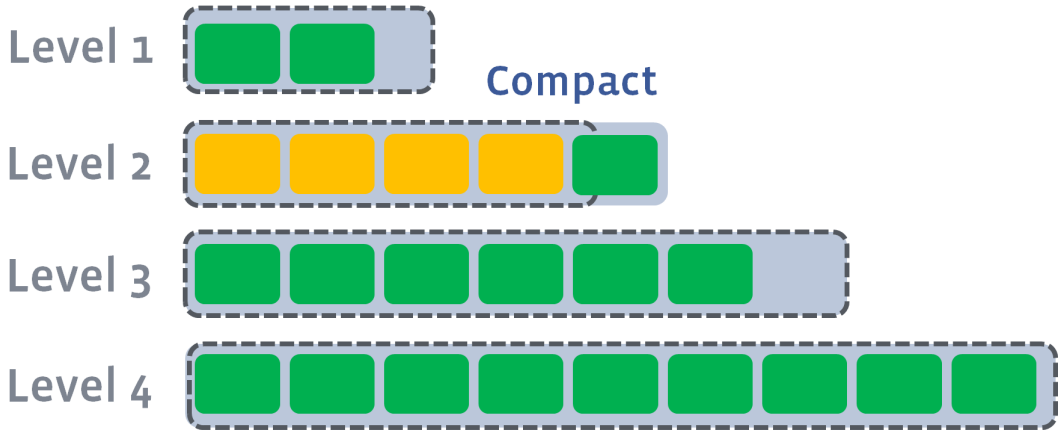


Σχήμα 2.3: *Αποτέλεσμα συμπύκνωσης του επιπέδου 0*

Όταν το επίπεδο 1 ξεπεράσει το όριό του (256 MB από προεπιλογή άρα 4 SSTables), επειδή το επόμενο επίπεδο είναι άδειο, όλα τα αρχεία του θα μεταφερθούν σε αυτό και το ίδιο θα μείνει άδειο. Την επόμενη φορά που θα ξεπεράσει το όριο, ένα αρχείο του θα επιλεγεί για να μεταβεί στο επίπεδο 2, θα συγχωνευτεί με όσα αρχεία του επιπέδου παρουσιάζει επικάλυψη, και αυτά θα αντικατασταθούν από τα αρχεία που παρήγαγε η συγχώνευση.



Σχήμα 2.4: *Επιλογή αρχείου για συμπύκνωση και μετάβαση σε επόμενο επίπεδο*



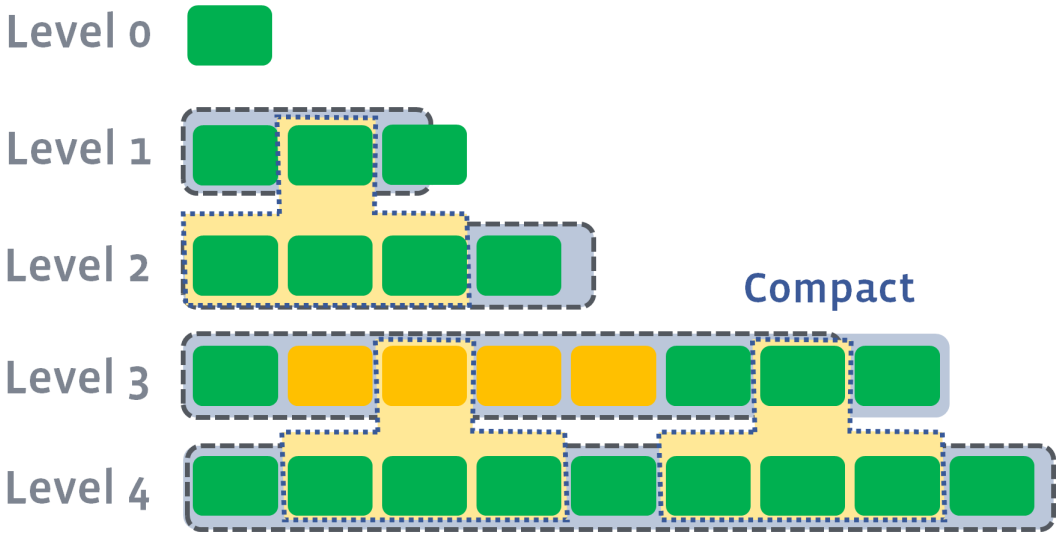
Σχήμα 2.5: *Αποτέλεσμα συμπύκνωσης*

Από το επίπεδο 2 και έπειτα, το όριο είναι ίσο με το όριο του προηγούμενου επιπέδου, πολλαπλασιασμένο επί έναν παράγοντα διακλάδωσης (branching factor), του οποίου η προεπιλεγμένη τιμή είναι 10. Άρα το όριο του επιπέδου n είναι $256 \cdot 10^n$ MB.

Συνοψίζοντας:

- Αν ξεπεραστεί το όριο του επιπέδου 0, συγχωνεύονται τα αρχεία των επιπέδων 0 και 1, στο επίπεδο 1.
- Αν ξεπεραστεί το όριο του τελευταίου επιπέδου, έστω n, όλα τα επίπεδα από 1 έως n κατεβαίνουν 1 επίπεδο.
- Σε όλες τις υπόλοιπες περιπτώσεις, επιλέγεται ένα αρχείο του επιπέδου που ξεπέρασε το όριο, και συγχωνεύεται με όσα αρχεία του επόμενου επιπέδου έχει επικάλυψη.

Επίσης υπάρχει η δυνατότητα να πραγματοποιούνται παράλληλα πολλαπλές συμπυκνώσεις από διαφορετικά νήματα, με την προϋπόθεση αυτές να χρησιμοποιούν διαφορετικά αρχεία.



Σχήμα 2.6: *Παράλληλες συμπυκνώσεις*

Υπάρχουν διάφοροι αλγόριθμοι για την επιλογή του αρχείου που θα μεταβεί στο επόμενο επίπεδο. Θα αναφέρουμε ενδεικτικά δύο από αυτούς. Ο πρώτος, που είναι και ο προεπιλεγμένος, δίνει προτεραιότητα στα αρχεία που περιλαμβάνουν πολλές καταχωρίσεις διαγραφών, με αποτέλεσμα να ανακτάται γρηγορότερα ο χώρος στο δίσκο. Ο δεύτερος, δίνει προτεραιότητα στα αρχεία που έχουν επικάλυψη με τα λιγότερα δυνατά αρχεία του επόμενου επιπέδου, με αποτέλεσμα να δημιουργούνται λιγότερα νέα αρχεία από τη διαδικασία της συμπύκνωσης.

Διαγραφές

Επειδή παλιότερες εκδόσεις ενός κλειδιού μπορεί να βρίσκονται σε πολλά αρχεία, δεν αρκεί να διαγράφεται απλά από το MemTable, γιατί έτσι θα παραμένουν παλιότερες εγγραφές για διαγραμμένα κλειδιά στη βάση. Για αυτόν το λόγο υπάρχει μια ειδική εγγραφή τύπου delete, που αντικαθιστά ενδεχόμενες εγγραφές τύπου put. Οι εγγραφές αυτές ονομάζονται tombstones (ταφόπλακες).

Τα tombstones παραμένουν στη βάση και μεταβαίνουν σε επόμενα επίπεδα αντικαθιστώντας παλαιότερες εγγραφές, μέχρι να φτάσουν στο τελευταίο επίπεδο του LSM-δέντρου όπου και διαγράφονται για πάντα.

Αν κατά την προσπάθεια ανάκτησης της τιμής ενός κλειδιού, βρεθεί ένα σχετικό tombstone, το RocksDB ανακοινώνει ότι το κλειδί αυτό δεν βρέθηκε.

Αναγνώσεις

Όταν ζητηθεί να διαβαστεί η τιμή ενός κλειδιού, το RocksDB αναζητεί το κλειδί αυτό σε όλες τις πιθανές τοποθεσίες με βάση τη σειρά προτεραιότητας, και επιστρέφει την αντίστοιχη τιμή μόλις αυτή βρεθεί. Αρχικά ελέγχει το τρέχον MemTable και μετά τα MemTables που δεν έχουν προλάβει ακόμα να μεταβούν στο δίσκο, αν υπάρχουν. Στη συνέχεια ελέγχει τα SSTables του επιπέδου 0 με τη σειρά, και τέλος το κατάλληλο SSTable του κάθε επιπέδου από το επίπεδο 1 έως το τελευταίο.

Το κατάλληλο SSTable είναι αυτό που περιλαμβάνει το εύρος κλειδιών στο οποίο ανήκει το ζητούμενο κλειδί. Το εύρος των κλειδιών του κάθε SSTable ενός επιπέδου, καθώς και η σχετική θέση μεταξύ τους είναι γνωστά και όλες οι σχετικές πληροφορίες βρίσκονται στη μνήμη. Επομένως για να βρεθεί το κατάλληλο αρχείο του επιπέδου, αρκεί να πραγματοποιηθεί δυαδική αναζήτηση σε αυτό. Επιπλέον, είναι γνωστά και τα αρχεία του επόμενου επιπέδου που παρουσιάζουν επικάλυψη με το κάθε SSTable. Επομένως αν βρεθεί το κατάλληλο αρχείο του επιπέδου 1 και δεν περιέχει το κλειδί, δεν θα πραγματοποιηθεί αναζήτηση και σε όλο το επίπεδο 2, παρά μόνο στα αρχεία που παρουσιάζουν επικάλυψη, κάτι που στη μέση περίπτωση δεν απαιτεί περισσότερες από 3 συγκρίσεις. [5]

Για την ανάκτηση συγκεκριμένου κλειδιού από ένα SSTable, υπάρχουν δείκτες που βοηθούν στην ανεύρεση της κατάλληλης σελίδας για ανάγνωση. Συγκεκριμένα, τα SSTables είναι χωρισμένα σε μπλοκ, και το κάθε SSTable περιέχει ένα ευρετήριο με το μεγαλύτερο κλειδί του κάθε μπλοκ. Επειδή το μέγεθος του μπλοκ είναι από προεπιλογή ίσο με το μέγεθος της σελίδας (4 KB), με δυαδική αναζήτηση στα μπλοκ του SSTable, βρίσκεται και ανακτάται από το λειτουργικό σύστημα, μόνο μία σελίδα του δίσκου, και στη συνέχεια το RocksDB ελέγχει αν το κλειδί βρίσκεται πραγματικά εκεί.

2.1.3 Μετρικές απόδοσης

Για την αξιολόγηση της απόδοσης του RocksDB, θα ορίσουμε ορισμένες μετρικές.

- **Ενίσχυση εγγραφής (write amplification):** Το πλήθος των bytes που θα χρειαστεί να γραφτούν στο δίσκο για κάθε byte που ζητείται να γραφτεί από το χρή-

[5] `https://github.com/facebook/rocksdb/wiki/Indexing-SST-Files-for-Better-Lookup-Performance`

στη.

- **Ενίσχυση ανάγνωσης (read amplification):** Το πλήθος των σελίδων που χρειάζεται να διαβαστούν από το δίσκο για να ανακτηθεί η σελίδα που πραγματικά περιέχει τα ζητούμενα δεδομένα. Εδώ δεν έχει νόημα να αναφερόμαστε σε bytes αφού είμαστε αναγκασμένοι να ανακτήσουμε ολόκληρη σελίδα από τον δίσκο (4 KB σε μια τυπική αρχιτεκτονική), ανεξάρτητα από το πλήθος των bytes που πρόκειται να διαβαστούν.
- **Ενίσχυση χώρου (space amplification):** Το πλήθος των bytes που βρίσκονται στο δίσκο για κάθε byte πραγματικών δεδομένων, σε κάποια χρονική στιγμή.

Στη συνέχεια θα υπολογίσουμε την απόδοση του RocksDB, σε σχέση με τις παραπάνω μετρικές, στις προεπιλεγμένες ρυθμίσεις. Θα πρέπει όμως να αναφέρουμε ότι προσφέρει διαφορετικούς τρόπους οργάνωσης του LSM-δέντρου, με διαφορετικούς συμβιβασμούς ανάμεσα στους τύπους ενίσχυσης.

Ενίσχυση εγγραφής

Υπολογίζουμε αναλυτικά την ενίσχυση εγγραφής που παρατηρείται για να μεταβεί ένα MemTable από τη μνήμη, στο τελευταίο επίπεδο του LSM-δέντρου.

- 1 για την εγγραφή στο WAL. Εδώ υποθέτουμε ότι το λειτουργικό σύστημα θα επιλέξει να γράψει τα δεδομένα στο δίσκο αφού θα έχουν συμπληρώσει τουλάχιστον μία σελίδα και δε θα ανανεώνει την ίδια σελίδα έπειτα από κάθε εγγραφή.
- 1 για τη εγγραφή στο επίπεδο 0.
- 2 για τη μεταφορά SSTable στο επίπεδο 1. Υποθέτουμε ότι έχουμε 4 SSTables στο επίπεδο 0 και 4 στο επίπεδο 1. Θα συγχωνευθούν όλα μαζί, επομένως στη χειρότερη περίπτωση θα δημιουργηθούν 8 αρχεία για να μεταφερθούν 4.
- 7 για κάθε άλλο επίπεδο. Από το επίπεδο 1 και έπειτα, για να μεταβεί ένα αρχείο σε χαμηλότερο επίπεδο θα συγχωνευτεί με όσα αρχεία του επιπέδου αυτού έχει επικάλυψη και θα δημιουργηθούν περίπου άλλα τόσα. Με παράγοντα διακλάδωσης 10 που είναι ο προεπιλεγμένος, διαισθητικά αναμένουμε τα αρχεία

αυτά να είναι περίπου 10. Όμως, λόγω της συμπίεσης και του τρόπου που επιλέγονται τα SSTables για μεταβούν σε επόμενο επίπεδο, στην πράξη αυτό το νούμερο είναι μικρότερο [6].

Άρα εάν έχουμε για παράδειγμα $n = 6$ επίπεδα (256 MB στα επίπεδα 0 και 1 με 2.56 TB στο επίπεδο 5), η ενίσχυση εγγραφής είναι $4 + 7(n - 2) = 32$. Το νούμερο αυτό φαίνεται αρκετά μεγάλο, αλλά για να το αξιολογήσουμε θα πρέπει να λάβουμε υπόψιν μας κάποια πράγματα.

- Αρχικά, θα το συγκρίνουμε με τη ενίσχυση που εμφανίζεται σε μια συμβατική δομή δεδομένων που χρησιμοποιείται για αποθήκευση δεδομένων στο δίσκο, όπως τα B-δέντρα [7]. Υποθέτουμε ότι στα B-δέντρα, τα δεδομένα φτάνουν κατ' ευθείαν στην τελική τους θέση και αγνοούμε τυχόν ανακατανομές των κόμβων, που πραγματοποιούνται για να επιτευχθεί η ισοζύγιση (rebalancing).

  Για να γραφτούν για παράδειγμα 64 bytes, θα πρέπει η αντίστοιχη σελίδα του δίσκου να διαβαστεί, να ανανεωθεί και να γραφτεί ολόκληρη. Αυτό σημαίνει ότι για 64 bytes θα χρειαστεί να γραφτούν 4 KB, άρα θα έχουμε ενίσχυση εγγραφής $4096/64 = 64$. Φυσικά σε αυτήν την περίπτωση, όσο μεγαλώνει το μέγεθος της εγγραφής, τόσο ελαττώνεται η ενίσχυση και το ανάποδο, ενώ στην περίπτωση των LSM-δέντρων η ενίσχυση είναι σταθερή και ανεξάρτητη του μεγέθους της κάθε εγγραφής.

- Επίσης, όλες οι εγγραφές είναι ακολουθιακές, οι οποίες είναι γενικά γρηγορότερες από τις τυχαίες εγγραφές, ειδικά στην περίπτωση που δεν χρησιμοποιείται δίσκος στερεάς κατάστασης (SSD), αλλά σκληρός δίσκος (HDD). Στους σκληρούς δίσκους, οι τυχαίες προσπελάσεις είναι σημαντικά πιο αργές λόγω του χρόνου αναζήτησης (seek time — ο χρόνος που χρειάζεται για να μετακινηθεί ο βραχίονας στον κατάλληλο κύλινδρο), και της καθυστέρησης λόγω περιστροφής (rotational delay — ο χρόνος που χρειάζεται για να περιστραφεί ο κατάλληλος τομέας κάτω από την κεφαλή) [8].

- Παρότι η ενίσχυση εγγραφής ρίχνει τη συνολική ταχύτητα με την οποία μπορούν να αποθηκεύονται δεδομένα στο δίσκο, αυτό δεν γίνεται απαραίτητα αντιληπτό από το χρήστη.

Οι συμπυκνώσεις πραγματοποιούνται στο παρασκήνιο και μόνο εφόσον έχουν ξεπεραστεί τα όρια των επιπέδων, ενώ οι εγγραφές δεν καθυστερούν παρά μόνο εάν ξεπεραστεί ένα προαιρετικό όριο στο πλήθος των αρχείων του επιπέδου 0. Άρα αν ο χρήστης δεν προσπαθεί να γράφει με τέτοιο ρυθμό ώστε να φτάνει το δίσκο στα όριά του, αντιλαμβάνεται πολύ μεγαλύτερη ταχύτητα εγγραφών, ενδεχομένως και ίση με την πραγματική ταχύτητα του δίσκου.

- Θα πρέπει τέλος να επισημάνουμε, ότι η ενίσχυση εγγραφής μπορεί να μειωθεί πολύ απλά, ελαττώνοντας τον παράγοντα διακλάδωσης. Κάτι τέτοιο, θα έχει όμως σαν αποτέλεσμα να αυξηθεί τόσο η ενίσχυση χώρου όσο και η ενίσχυση ανάγνωσης. Οι προεπιλεγμένες ρυθμίσεις του RocksDB, έχουν επιλεγεί με δεδομένο ότι χρησιμοποιούνται γρήγορα αποθηκευτικά μέσα, όπως δίσκοι SSD, τα οποία έχουν μεγάλη ταχύτητα εγγραφής, όμως ο ίδιος ο αποθηκευτικός χώρος κοστίζει. Επομένως, είναι προτιμότερο να ελαττωθεί η ταχύτητά τους μέσω της ενίσχυσης εγγραφής, παρά τα αποθηκευμένα δεδομένα να καταλαμβάνουν περισσότερο χώρο από όσο χρειάζεται.

Ενίσχυση ανάγνωσης

Αν έχουμε 4 SSTables στο επίπεδο 0, και 5 ακόμη επίπεδα, η ενίσχυση ανάγνωσης είναι 4 + 5 = 9. Αυτό φυσικά προϋποθέτει το ευρετήριο για όλα τα κλειδιά σε όλα τα επίπεδα να βρίσκεται στη μνήμη και επομένως να γίνεται η ανάκτηση μόνο της σωστής σελίδας του κάθε επιπέδου από το δίσκο.

Με ενίσχυση ανάγνωσης 9, για να ανακτηθεί ένα κλειδί που βρίσκεται στο τελευταίο επίπεδο θα πρέπει να ελεγχθούν 9 διαφορετικές σελίδες πριν βρεθεί η σωστή. Αυτό το πρόβλημα παρακάμπτεται με τη χρήση των φίλτρων Bloom [9].

Τα φίλτρα Bloom λειτουργούν ως εξής. Έχουμε πίνακα από bits των m στοιχείων αρχικοποιημένων στο 0, και k διαφορετικές συναρτήσεις κατακερματισμού που αντιστοιχίζουν συμβολοσειρές σε κάποια από τις θέσεις του πίνακα. Κάθε κλειδί που προστίθεται, περνάει και από τις k συναρτήσεις και τα bits των αντίστοιχων θέσεων παίρνουν την τιμή 1. Για να εξετάσουμε αν ένα κλειδί βρίσκεται στον πίνακα, ελέγχουμε τα περιεχόμενα του πίνακα στις αντίστοιχες k θέσεις. Αν έστω και ένα βρεθεί ίσο με 0, είναι βέβαιο ότι το κλειδί δεν έχει προστεθεί, ενώ αν βρεθούν και τα k ίσα με 1, το κλειδί

μάλλον έχει προστεθεί, όμως δεν μπορούμε να αποφανθούμε με βεβαιότητα, καθώς ενδέχεται οι τιμές αυτές να έχουν τεθεί από διαφορετικά κλειδιά.

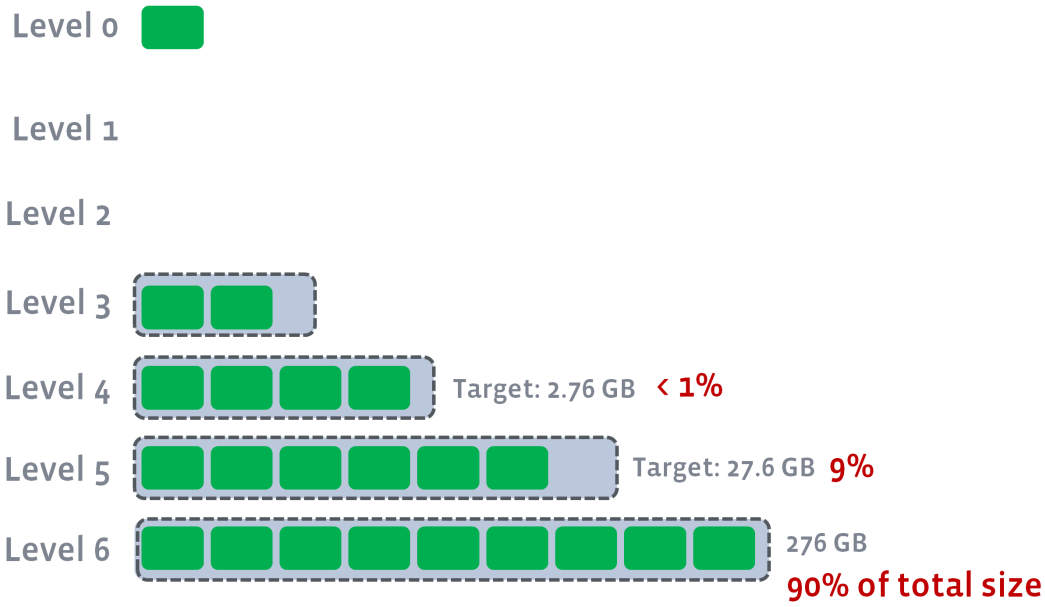
Γενικά τα φίλτρα Bloom έχουν το πρόβλημα ότι δεν υποστηρίζουν διαγραφές κάτι που αυξάνει σημαντικά τα εσφαλμένα θετικά αποτελέσματα ή απαιτεί την περιοδική ανακατασκευή τους. Αυτό όμως δεν ισχύει στην περίπτωση του RocksDB, καθώς τα SSTables είναι αμετάβλητα.

Υπολογίζεται ότι για πίνακα των 10 bits επί το πλήθος των κλειδιών που πρόκειται να προστεθούν, και 6 συναρτήσεις κατακερματισμού, η πιθανότητα εσφαλμένα θετικών αποτελεσμάτων είναι περίπου 1% [10]. Άρα στο παράδειγμά μας έχουμε 9% πιθανότητα συνολικά ανά ανάγνωση, και η ενίσχυση πέφτει στο 1.09.

Ενίσχυση χώρου

Στη χειρότερη περίπτωση, το τελευταίο επίπεδο μιας βάσης περιλαμβάνει όλα τα κλειδιά της, ενώ τα υπόλοιπα επίπεδα περιέχουν μόνο ανανεωμένες τιμές σε υπάρχοντα κλειδιά. Έστω S ο χώρος που καταλαμβάνει το τελευταίο επίπεδο. Με παράγοντα διακλάδωσης ρυθμισμένο στο 10, το προτελευταίο επίπεδο θα καταλαμβάνει S/10, το αμέσως προηγούμενο S/100 κοκ. Άρα ενίσχυση χώρου θα είναι κάτι λιγότερο από 1.111…

Φυσικά αυτό προϋποθέτει όλα τα επίπεδα να είναι κοντά στα όριά τους. Διαφορετικά, στη χειρότερη περίπτωση, το τελευταίο με το προτελευταίο επίπεδο θα έχουν το ίδιο περίπου μέγεθος, και η ενίσχυση χώρου θα είναι μεγαλύτερη από 2. Για να εξασφαλιστεί ότι το κάθε επίπεδο θα είναι δεκαπλάσιο του προηγούμενου, υπάρχει η επιλογή να καθορίζονται τα όρια δυναμικά, με βάση το μέγεθος του μεγαλύτερου επιπέδου. Η βάση αρχίζει να γεμίζει από το μεγαλύτερο δυνατό επίπεδο (το επίπεδο 6 από προεπιλογή). Αν S το μέγεθός του, το όριό του είναι S, το όριο του προτελευταίου επιπέδου S/10 κτλ, μέχρι κάποιο επίπεδο να έχει όριο l, μικρότερο από το ελάχιστο (256 MB από προεπιλογή), δηλαδή $256/10 \text{ MB} < l < 256 \text{ MB}$. Τα υπόλοιπα επίπεδα παραμένουν άδεια.



Σχήμα 2.7: *Δομή LSM-δέντρου με δυναμικά όρια επιπέδων*

Τέλος παρατηρούμε ότι τα SSTables αποθηκεύονται στο δίσκο συμπιεσμένα. Είναι δύσκολο να υπολογιστεί το πλεονέκτημα που παρέχει η συμπίεση στην ενίσχυση χώρου καθώς εξαρτάται από τον αλγόριθμο συμπίεσης που χρησιμοποιείται (οι διαθέσιμοι είναι οι LZ, Snappy, zlib και Zstandard), καθώς και από τη μορφή των δεδομένων. Σύμφωνα με το Facebook, ο χώρος που καταλαμβάνει ένα SSTable μπορεί να φτάσει μέχρι και το 25% των αρχικών δεδομένων [11].

2.1.4 Χρήσιμες δυνατότητες

Το RocksDB παρέχει πολλές δυνατότητες και συνεχώς προστίθενται νέες. Θα αναφέρουμε μόνο αυτές που φάνηκαν χρήσιμες κατά την υλοποίηση της παρούσας διπλωματικής εργασίας.

Column Families

Τα column families επιτρέπουν την αποθήκευση κλειδιών, ενδεχομένως και κοινών, σε διαφορετικές "στήλες". Τα διαφορετικά column families διατηρούν διαφορετικά MemTables και LSM-δέντρα αλλά μοιράζονται το WAL. Αυτό έχει ως αποτέλεσμα να αποτελούν διαφορετικές ουσιαστικά βάσεις δεδομένων, αλλά έχουν το πλεονέκτημα

ότι υποστηρίζουν ατομικές ενέργειες πάνω σε περισσότερες από μια οικογένειες.

Είναι ιδιαίτερα χρήσιμα, εάν υπάρχει η επιθυμία να διαγράφονται συγκεκριμένες ομάδες κλειδιών. Εφόσον έχει προβλεφθεί να ανήκουν όλα στο ίδιο column family, διαγράφοντας το, διαγράφονται όλα τα κλειδιά χωρίς κάποια επιβάρυνση. Διαφορετικά θα έπρεπε να διαγραφούν ένα ένα, κάτι που θα γέμιζε τη βάση με tombstones και θα είχε σημαντική επίπτωση στην απόδοση.

Write Batches

Το write batch, είναι μια ομάδα εγγραφών που εκτελούνται ατομικά. Συγκεκριμένα η εγγραφή στο WAL πραγματοποιείται ατομικά, και επομένως σε περίπτωση αποτυχίας είτε θα πραγματοποιηθούν όλες οι εγγραφές είτε καμία. Επίσης παρουσιάζουν σημαντικό πλεονέκτημα στην ταχύτητα εισαγωγής.

Στιγμιότυπα

Υπάρχει η δυνατότητα να διατηρείται ένα στιγμιότυπο (snapshot) της βάσης, όπως ήταν σε μια συγκεκριμένη χρονική στιγμή. Το γεγονός ότι τα SSTables είναι αμετάβλητα καθιστά τη διαδικασία αυτή πολύ απλή. Αρκεί τα τρέχοντα MemTables να περάσουν στο δίσκο, να δημιουργηθεί ένα αντίγραφο της δομής των LSM-δέντρων, και ο μετρητής αναφορών όλων των SSTables να αυξηθεί ώστε να μην διαγραφούν όταν πάψουν να ανήκουν στο κύριο LSM-δέντρο.

Επαναλήπτες

Οι επαναλήπτες (iterators) επιτρέπουν την προσπέλαση ενός εύρους κλειδιών που ανήκουν σε ένα Column Family (ενδεχομένως και ολόκληρου του Column Family) σε αύξουσα ή φθίνουσα σειρά σε ένα στιγμιότυπο της βάσης.

Δημιουργία/εισαγωγή SSTable

Το RocksDB, επιτρέπει τη δημιουργία SSTable επιλέγοντας τα ζεύγη κλειδιού-τιμής που θα τα απαρτίζουν, καθώς και την προσθήκη SSTable σε ένα υπάρχον LSM-δέντρο[6].

[6] `https://github.com/facebook/rocksdb/wiki/Creating-and-Ingesting-SST-files`

Η εισαγωγή ενός SSTable μπορεί να πραγματοποιηθεί με δύο τρόπους. Ο πρώτος είναι να προστεθεί στην κατάλληλη θέση ώστε οι τιμές που περιλαμβάνει να έχουν την υψηλότερη προτεραιότητα. Αν έχει επικάλυψη με τα κλειδιά που βρίσκονται στο MemTable, αναγκαστικά πρέπει το MemTable να μεταφερθεί στο δίσκο και στη συνέχεια το SSTable να προστεθεί στην κορυφή του επιπέδου 0. Αν δεν υπάρχει τέτοια επικάλυψη, προστίθεται στο χαμηλότερο δυνατό επίπεδο έτσι ώστε να μην υπάρχει επικάλυψη με κανένα SSTable που βρίσκεται από πάνω του.

Ο δεύτερος τρόπος είναι να προστεθεί σε ένα ξεχωριστό επίπεδο που βρίσκεται χαμηλότερα από όλα τα άλλα, και είναι δεσμευμένο για τέτοιου είδους εισαγωγές. Έτσι έχει την χαμηλότερη δυνατή προτεραιότητα, και κλειδιά που περιλαμβάνει και υπήρχαν ήδη στη βάση, είναι σαν να μην προστέθηκαν καθόλου.

2.2 Protocol Buffers

Το protobuf[7] (συντόμευση για Protobuf Buffers), είναι μια βιβλιοθήκη που επιτρέπει την γρήγορη σειριοποίηση και αποσειριοποίηση δομημένων δεδομένων, με στόχο το να καταλαμβάνουν όσο το δυνατόν λιγότερο χώρο.

Η δομή των δεδομένων ορίζεται από τον προγραμματιστή στη γλώσσα των protocol buffers, και με τη χρήση ενός μεταγλωττιστή παράγεται κώδικας για όποια γλώσσα προγραμματισμού επιλεγεί. Οι διαθέσιμες γλώσσες είναι οι εξής:

- C++
- Java
- Python
- Objective-C
- C#
- JavaNano
- JavaScript

[7] `https://developers.google.com/protocol-buffers/`

- Ruby
- Go
- PHP
- Dart

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}

message SearchResponse {
  repeated Result results = 1;
}

message Result {
  string url = 1;
  string title = 2;
  repeated string snippets = 3;
}
```

Listing 2.1: *Παράδειγμα ορισμού protobuf μηνυμάτων*

Ο μεταγλωττιστής για κάθε τύπο μηνύματος, ορίζει μία κλάση με τα πεδία να αντιστοιχούν στον κατάλληλο τύπο ανάλογα τη γλώσσα προγραμματισμού. Για παράδειγμα το `repeated string snippets`, στη C++ αντιστοιχεί σε διάνυσμα από συμβολοσειρές (`std::vector<std::string>`) ενώ στην Python σε λίστα από συμβολοσειρές (`List(str)`). Επίσης ορίζονται ορισμένες χρήσιμες μέθοδοι για την επεξεργασία και την ανάκτηση των πεδίων (getters/setters), την σειριοποίηση κλπ.

2.3 gRPC

Το gRPC[8] είναι μία βιβλιοθήκη που επιτρέπει την υλοποίηση κλήσεων απομακρυσμένων διαδικασιών (Remote Procedure Call — RPC). Στηρίζεται στην ανταλλαγή protobuf μηνυμάτων με τη χρήση του δικτυακού πρωτοκόλλου HTTP/2.

[8] `https://grpc.io/`

Με το gRPC, η γλώσσα των protocol buffers επεκτείνεται για να επιτρέψει τον ορισμό διαδικασιών, οι οποίες λαμβάνουν ως είσοδο και δίνουν ως έξοδο protobuf μηνύματα. Τα μηνύματα μπορεί να είναι μεμονωμένα ή και ροή μηνυμάτων (stream). Ο ορισμός έχει την εξής μορφή:

```
service RPC {
  rpc Search(SearchRequest) returns (stream SearchResponse) {}
}
```

Listing 2.2: *Παράδειγμα ορισμού RPC*

Σε αντιστοιχία με τα protocol buffers, ένας μεταγλωττιστής αναλαμβάνει την παραγωγή κώδικα που προσφέρει ένα API για την υλοποίηση των διακομιστών που παρέχουν τις διαδικασίες, και των πελατών που τις καλούν.

Το API αυτό έχει δύο μορφές, το σύγχρονο και το ασύγχρονο. Με το σύγχρονο, οι κλήσεις συναρτήσεων που απαιτούν δικτυακή επικοινωνία μπλοκάρουν μέχρι αυτή να ολοκληρωθεί, με αποτέλεσμα να χρειάζονται ξεχωριστά νήματα για να πραγματοποιηθούν περισσότερες από μία κλήσεις παράλληλα.

```
SearchRequest request;
SearchResponse response;
request.set_query("query");
Search(request, &response);
```

Listing 2.3: *Παράδειγμα σύγχρονου API*

Με το ασύγχρονο, οι κλήσεις αυτές παίρνουν σαν όρισμα μια ουρά και ένα αναγνωριστικό, και επιστρέφουν αμέσως. Μόλις κάποια ολοκληρωθεί, το αναγνωριστικό της μπαίνει στην ουρά από το gRPC. Επομένως αρκεί ένα νήμα που να αφαιρεί αναγνωριστικά από την ουρά. Κάθε φορά που συμβαίνει αυτό, είναι βέβαιο ότι η αντίστοιχη απάντηση έχει φτάσει.

```
SearchRequest request;
SearchResponse response;
CompletionQueue queue;
int tag = 0;
request.set_query("query");
AsyncSearch(request, &response, &queue, tag);
int got_tag;
queue.Next(&got_tag);  // Blocks
```

```
assert(got_tag == 0);
// response is populated with the answer
```

Listing 2.4: *Παράδειγμα ασύγχρονου API*

2.4 etcd

Το etcd[9] είναι ένα αξιόπιστο κατανεμημένο σύστημα αποθήκευσης κλειδιού-τιμής, που χρησιμοποιείται για την αποθήκευση των κρίσιμων δεδομένων ενός κατανεμημένου συστήματος. Παρέχει ένα απλό και καλώς ορισμένο gRPC API, και είναι ασφαλές και γρήγορο. Για να εξασφαλίσει τη συνέπεια των δεδομένων που αποθηκεύει, χρησιμοποιεί τον αλγόριθμο συναίνεσης (consensus algorithm) Raft [12]. Έχει σχεδιαστεί ώστε να μην επηρεάζεται η ορθότητα του από αποτυχίες μελών, ηγέτη, πλειοψηφίας ή δικτύου και επιτρέπει την προσθαφαίρεση μελών στη συστοιχία του.

Εκτός από τις βασικές λειτουργίες ενός συστήματος αποθήκευσης κλειδιού-τιμής (get, put, delete), το etcd παρέχει πολλές άλλες δυνατότητες. Από αυτές θα μας απασχολήσουν η παρακολούθηση κλειδιών και οι συναλλαγές.

Κάνοντας χρήση της παρακολούθησης, ο πελάτης μπορεί να ενημερώνεται για τις αλλαγές ενός ή περισσότερων κλειδιών ή/και εύρους κλειδιών, χωρίς να απαιτείται από τον ίδιο να πραγματοποιεί περιοδικά αιτήματα. Πρόκειται για μια κλήση απομακρυσμένης διαδικασίας του gRPC, που σαν είσοδο παίρνει ένα protobuf μήνυμα που περιλαμβάνει τα ζητούμενα κλειδιά, και επιστρέφει ροή από μηνύματα με τα κλειδιά που ανανεώθηκαν και τις νέες τους τιμές.

Επίσης το etcd υποστηρίζει συναλλαγές. Συγκεκριμένα επιτρέπει την πραγματοποίηση κάποιων εγγραφών εφόσον πληρούνται ορισμένες συνθήκες ή/και κάποιων άλλων εφόσον δεν πληρούνται. Οι συνθήκες μπορεί να είναι ένα κλειδί να έχει μια συγκεκριμένη τιμή, να μην έχει ανανεωθεί μετά από μια συγκεκριμένη χρονική στιγμή κτλ. Πρακτικά αυτό σημαίνει ότι με το etcd, έχουμε τη δυνατότητα να διαβάζουμε την τιμή ενός κλειδιού, να την ανανεώνουμε και να την ξαναγράφουμε, και όλη αυτή η διαδικασία να πραγματοποιείται ατομικά, χωρίς να υπάρχει ο κίνδυνος να παρέμβει άλλος πελάτης. Σε περίπτωση αποτυχίας επαναλαμβάνουμε όλη τη διαδικασία μέχρι

[9] https://github.com/coreos/etcd

να πετύχει.

3 Σχεδίαση

Ο στόχος αυτής της διπλωματικής εργασίας είναι να επεκτείνουμε το RocksDB, αναπτύσσοντας μια εφαρμογή η οποία θα επιτρέπει την κατανεμημένη αποθήκευση δεδομένων σε μία συστοιχία κόμβων, και θα επιτυγχάνει όσο το δυνατόν υψηλότερη ταχύτητα εγγραφών, καθώς και οριζόντια κλιμάκωση χωρίς περιορισμούς. Δηλαδή αν με n κόμβους καταφέρνουμε να έχουμε x συνολική ταχύτητα εγγραφών, να είναι δυνατόν να διπλασιαστούν οι κόμβοι με την ταχύτητα να ανεβαίνει στο $2x$.

3.1 Κατακερματισμός δεδομένων

Θα χρειαστεί να χωρίσουμε τη βάση δεδομένων, δηλαδή το σύνολο των πιθανών κλειδιών, σε λογικά τμήματα έτσι ώστε να μπορεί να μοιραστεί ισάξια στους διαφορετικούς κόμβους. Επίσης θα πρέπει να προνοήσουμε ώστε να είναι εύκολο να περνάνε τμήματα της βάσης από τον έναν κόμβο στον άλλο όταν αυξάνεται ή μειώνεται το πλήθος τους, σύμφωνα με τις ανάγκες της εφαρμογής. Αυτό σημαίνει ότι ένα μεγάλο ποσοστό της βάσης ενός κόμβου θα πρέπει να διαγραφεί.

Αρχικά παρατηρούμε ότι στο RocksDB, η διαγραφή πολλών κλειδιών δεν είναι εύκολη διαδικασία. Αν αποφασίσουμε να διαγράψουμε τα μισά κλειδιά μιας βάσης, περιμένουμε ο χώρος που καταλαμβάνει να πέσει στο μισό. Αυτό που θα γίνει όμως στην πραγματικότητα, είναι να μείνει ως έχει και να επιβαρυνθεί επιπλέον με tombstones. Φυσικά στην πορεία αυτό θα διορθωθεί όταν μετά από πολλές συμπυκνώσεις, φτάσουν όλα τα tombstones στο τελευταίο επίπεδο του LSM-δέντρου και απελευθερωθεί

ο χώρος. Αυτό απαιτεί πολλούς πόρους και χρόνο κατά τον οποίο η απόδοση του συστήματός θα είναι περιορισμένη.

Για να ξεπεράσουμε αυτό το πρόβλημα θα χρησιμοποιήσουμε τα column families που παρέχει το RocksDB, τα οποία όπως έχουμε δει μπορούν να δημιουργούνται και να διαγράφονται χωρίς επιβάρυνση. Χωρίζουμε τη βάση σε ένα σύνολο τμημάτων στα οποία θα αναφερόμαστε ως shards (θραύσματα). Ένα shard είναι το ελάχιστο σύνολο ζευγών κλειδιού-τιμής που μπορεί να μεταφερθεί από έναν κόμβο σε έναν άλλο, και αποθηκεύεται σε ένα δικό του ξεχωριστό column family.

Η χρήση ξεχωριστών column families έχει ένα ακόμη πλεονέκτημα. Το γεγονός ότι χρησιμοποιούν διαφορετικά LSM-δέντρα, κάνει το κάθε shard να απαιτεί λιγότερα επίπεδα από ότι αν ο κάθε κόμβος διατηρούσε όλα τα κλειδιά του στο ίδιο column family. Αυτό οδηγεί σε χαμηλότερη ενίσχυση εγγραφής και επομένως μεγαλύτερη ταχύτητα. Από την άλλη, τα column families απαιτούν υπολογιστικούς πόρους, και κυρίως μνήμη, καθώς χρησιμοποιούν διαφορετικές δομές (MemTables, caches κτλ.). Επομένως το αρχικό πλήθος των shards ανά κόμβο, θα πρέπει να επιλέγεται προσεκτικά.

Για να χωρίσουμε τη βάση σε shards, επιλέγουμε να χρησιμοποιήσουμε μια συνάρτηση κατακερματισμού. Έστω `hash()` η συνάρτηση κατακερματισμού και n το πλήθος των shards με αναγνωριστικά 0…n-1. Αντιστοιχίζουμε το κλειδί "key" στο shard με αναγνωριστικό `hash(key) modulo n`. Έτσι μπορούμε να εγγυηθούμε ότι όλα τα shards θα καταλαμβάνουν περίπου τον ίδιο χώρο, κάτι που θα ήταν αδύνατο αν χρησιμοποιούσαμε κάποια άλλη μέθοδο, όπως για παράδειγμα αν αναθέταμε διαφορετικές περιοχές κλειδιών σε διαφορετικά shards (π.χ. a–b στο shard 0, c–d στο shard 1 κτλ.).

Για τις ανάγκες του συστήματός μας, η συνάρτηση κατακερματισμού δεν είναι απαραίτητο να είναι κρυπτογραφική, δηλαδή να είναι δύσκολο να βρεθεί η είσοδος που δόθηκε στη συνάρτηση και παρήγαγε μια συγκεκριμένη έξοδο. Αρκεί να είναι γρήγορη και να παρουσιάζει ομοιόμορφη κατανομή, δηλαδή η κάθε είσοδος να έχει την ίδια πιθανότητα να παράγει οποιαδήποτε τιμή.

Η συνάρτηση κατακερματισμού που επιλέχθηκε είναι μια παραλλαγή της `MurmurHash`[1], που χρησιμοποιείται εσωτερικά από το RocksDB[2].

[1] `https://github.com/aappleby/smhasher`
[2] `https://github.com/facebook/rocksdb/blob/master/util/hash.cc`

3.2 Αρχιτεκτονική

Για να πετύχουμε την ζητούμενη κλιμακωσιμότητα, θα πρέπει να περιορίσουμε την επικοινωνία μεταξύ των διαφορετικών κόμβων στο ελάχιστο, ώστε να μην αποτελεί σημείο συμφόρησης (bottleneck) όταν αυξάνονται οι κόμβοι.

Επομένως δεν υπάρχει κόμβος με κάποιον ιδιαίτερο ρόλο (π.χ. που να συγχρονίζει λειτουργίες πάνω στους υπόλοιπους κόμβους). Όλοι οι κόμβοι είναι ισάξιοι και ο πελάτης επικοινωνεί απευθείας με αυτόν που είναι υπεύθυνος για το αντίστοιχο shard, και όχι με κάποιον διαμεσολαβητή (proxy). Φυσικά αν για τις ανάγκες κάποιας συγκεκριμένης εφαρμογής, χρειάζεται να υπάρχει τέτοιος διαμεσολαβητής, μπορεί να υλοποιηθεί ένας "πελάτης" που να αναλαμβάνει αυτόν το ρόλο, και να επικοινωνούν μαζί του οι πραγματικοί πελάτες τις εφαρμογής.

Σε μια συστοιχία etcd αποθηκεύουμε τις απολύτως απαραίτητες πληροφορίες που πρέπει να έχουν τόσο οι κόμβοι, όσο και οι πελάτες, για την ορθή λειτουργία της εφαρμογής μας. Κατά βάση χρησιμοποιούμε το etcd για να γνωρίζουμε ανά πάσα στιγμή τους κόμβους που συμμετέχουν στη συστοιχία, τη διεύθυνση στην οποία ακούει ο καθένας και σε ποιον ανήκει το κάθε shard. Οι κόμβοι παρακολουθούν το etcd για αυτές τις πληροφορίες (βλ. παράγραφο 2.4), επομένως σε περίπτωση που πραγματοποιηθεί κάποια αλλαγή, λαμβάνουν άμεσα την τελευταία έκδοση.

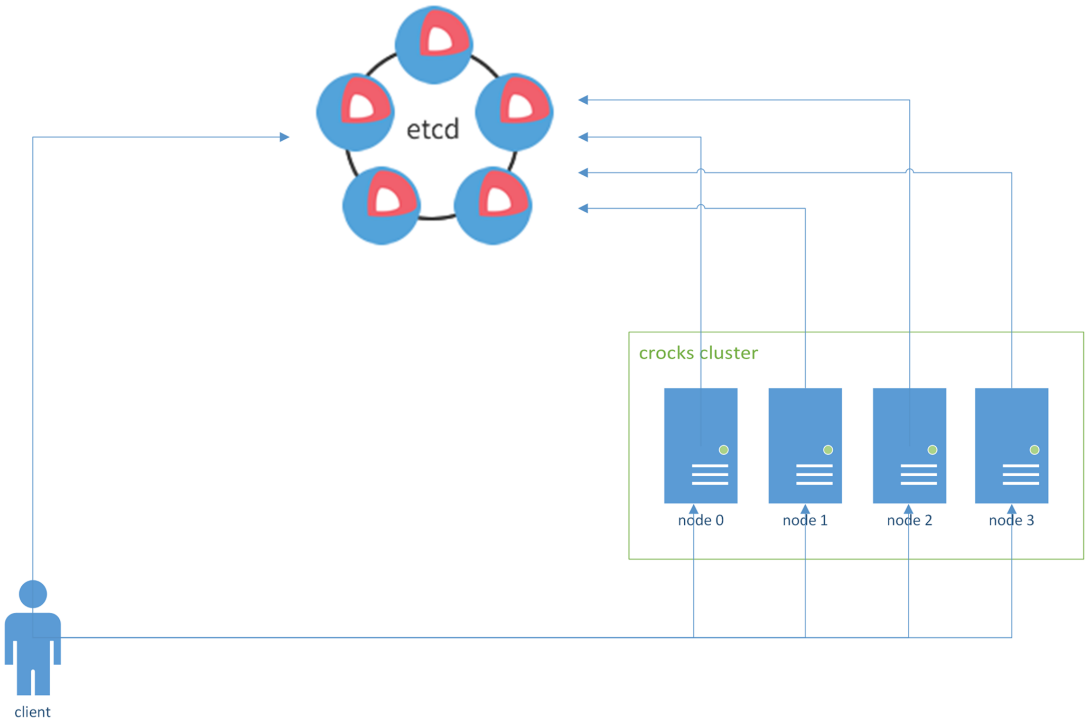
Οι πελάτες επίσης ενημερώνονται μέσω του etcd, άλλα δεν το παρακολουθούν συνεχώς για να μην επιβαρύνουν το δίκτυο με περιττή κίνηση. Αντίθετα λαμβάνουν μία φορά τις πληροφορίες, και τις ζητούν εκ νέου αν χρειαστεί, δηλαδή αν εκτελέσουν κάποιο αίτημα για ένα shard σε έναν κόμβο, και εκείνος απαντήσει ότι δεν είναι ο ίδιος υπεύθυνος.

Επειδή η συστοιχία etcd έχει περιορισμένο μέγεθος (3–9 κόμβους), και η δική μας συστοιχία θέλουμε να μεγαλώνει χωρίς όριο, είναι αδύνατον να διατηρούμε έναν κόμβο etcd ανά κόμβο. Αντίθετα χρησιμοποιούμε μια απομακρυσμένη συστοιχία etcd με την οποία επικοινωνούν όλοι οι κόμβοι.[3]

Επίσης για να μην χρειαστεί να υλοποιήσουμε μηχανισμό συνεχούς ενημέρωσης των κόμβων για την κατάσταση της συστοιχίας etcd, ώστε σε περίπτωση αποτυχίας κάποιου δικού της κόμβου να δημιουργείται νέα σύνδεση προς τον επόμενο διαθέσιμο,

[3] `https://coreos.com/os/docs/latest/cluster-architectures.html`

θεωρούμε ότι επικοινωνούν με μια πύλη etcd (etcd gateway)[4]. Η πύλη λειτουργεί ως διαμεσολαβητής, και αναλαμβάνει εκείνη να ενημερώνεται για την κατάσταση του etcd. Υποθέτοντας ότι ο διαχειριστής του δικτύου έχει φροντίσει ώστε η πύλη etcd να είναι μόνιμα διαθέσιμη, χρησιμοποιώντας κάποιο μηχανισμό αυτόματης μετάπτωσης (failover), αντιμετωπίζουμε το etcd ως ένα "μαύρο κουτί" που θα λειτουργεί πάντα σωστά.

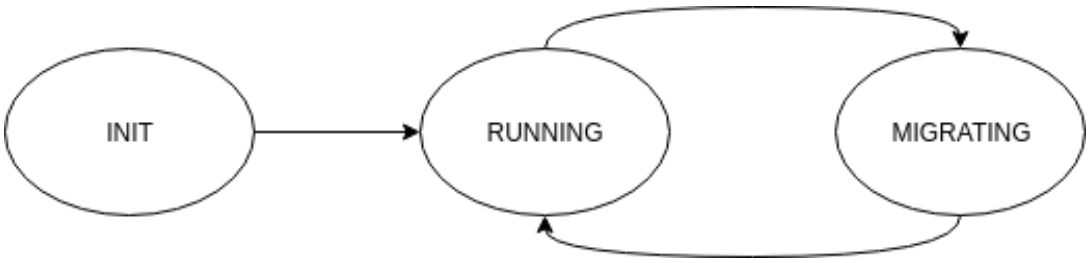


Σχήμα 3.1: *Αρχιτεκτονική*

3.3 Καταστάσεις συστοιχίας

Η συστοιχία περνάει από τα στάδια της αρχικοποίησης (`INIT`), της εκτέλεσης (`RUNNING`) και της μετανάστευσης (`MIGRATING`).

[4] `https://coreos.com/etcd/docs/latest/op-guide/gateway.html`



Σχήμα 3.2: *Καταστάσεις συστοιχίας*

3.3.1 Κατάσταση `INIT`

Αρχικά η συστοιχία βρίσκεται στην κατάσταση `INIT`, όπου αρχικοποιούνται οι κόμβοι που θα αποτελούν την πρώτη έκδοση της συστοιχίας. Ο κάθε κόμβος, ανακοινώνει τη διεύθυνση στην οποία ακούει στο etcd. Ο διαχειριστής διαλέγει το πλήθος των shards που επιθυμεί και τα μοιράζει στους κόμβους.

Όταν συνδεθεί ο πρώτος πελάτης ή όταν το ζητήσει ο διαχειριστής, η συστοιχία μεταβαίνει στην κατάσταση `RUNNING`, και αρχίζουν να εξυπηρετούνται αιτήματα πελατών.

3.3.2 Κατάσταση `RUNNING`

Ο πελάτης πριν εκτελέσει το πρώτο του αίτημα, λαμβάνει από το etcd τις πληροφορίες της συστοιχίας. Στη συνέχεια, βρίσκει το shard που αντιστοιχεί στο κλειδί, και το στέλνει στον κόμβο που αποτελεί τον υπεύθυνό του. Για παράδειγμα, ένα αίτημα για εγγραφή ενός ζεύγους κλειδιού-τιμής (put), μοιάζει κάπως έτσι:

```
shard_id = hash(key) mod num_shards
shard = shards[shard_id]
node = shard.master
node.put(key, value)
```

Listing 3.1: *Put*

Επειδή ενδέχεται ο υπεύθυνος να έχει αλλάξει, ο κόμβος που θα λάβει ένα αίτημα ελέγχει ότι πράγματι είναι ο ίδιος ο υπεύθυνος. Αν είναι, εκτελεί το αίτημα και προωθεί ό,τι του επέστρεψε το RocksDB, ενώ αν δεν είναι στέλνει ένα status που παραπέμπει τον πελάτη να ζητήσει τις ανανεωμένες πληροφορίες από το etcd.

Σε αυτό το στάδιο ο διαχειριστής μπορεί να προσθέσει ή να αφαιρέσει κόμβους. Κάθε νέος κόμβος προσθέτει μια καταχώρηση για τον εαυτό του στο etcd, ενώ για να αφαιρεθεί κάποιος, ο διαχειριστής ενημερώνει σχετικά τις πληροφορίες του etcd. Στη συνέχεια ανακατανέμει τα shards στους κόμβους που πρόκειται να παραμείνουν στη συστοιχία, αλλάζει την κατάσταση σε `MIGRATING`, και στέλνει στο etcd τις ανανεωμένες πληροφορίες. Η ανακατανομή γίνεται ως εξής:

Αρχικά υπολογίζεται το πλήθος των shards που θα έχει ο κάθε κόμβος στο τέλος της διαδικασίας ώστε να είναι κατανεμημένα δίκαια. Έστω s το πλήθος των shards και n το πλήθος των κόμβων που θα παραμείνουν. Οι πρώτοι $s \mod n$ κόμβοι παίρνουν $s/n + 1$ shards και οι υπόλοιποι s/n.

Στη συνέχεια υπολογίζεται πόσα shards πρέπει να πάρει ή να δώσει ο κάθε κόμβος για φτάσει στο στόχο και τέλος υπολογίζονται οι ακριβείς μετακινήσεις που θα γίνουν, με τον ακόλουθο αλγόριθμο.

```
i = 0
for each node in nodes:
  while node.shards_to_take > 0:
    shard = shards[i]
    if shard.master.shards_to_give > 0:
      shard.migrate = true
      shard.from = shard.master
      shard.to = node
      shard.master.shards_to_give −= 1
      node.shards_to_take −= 1
    i += 1
```

Listing 3.2: *Αλγόριθμος ανακατανομής των shards*

3.3.3 Κατάσταση `MIGRATING`

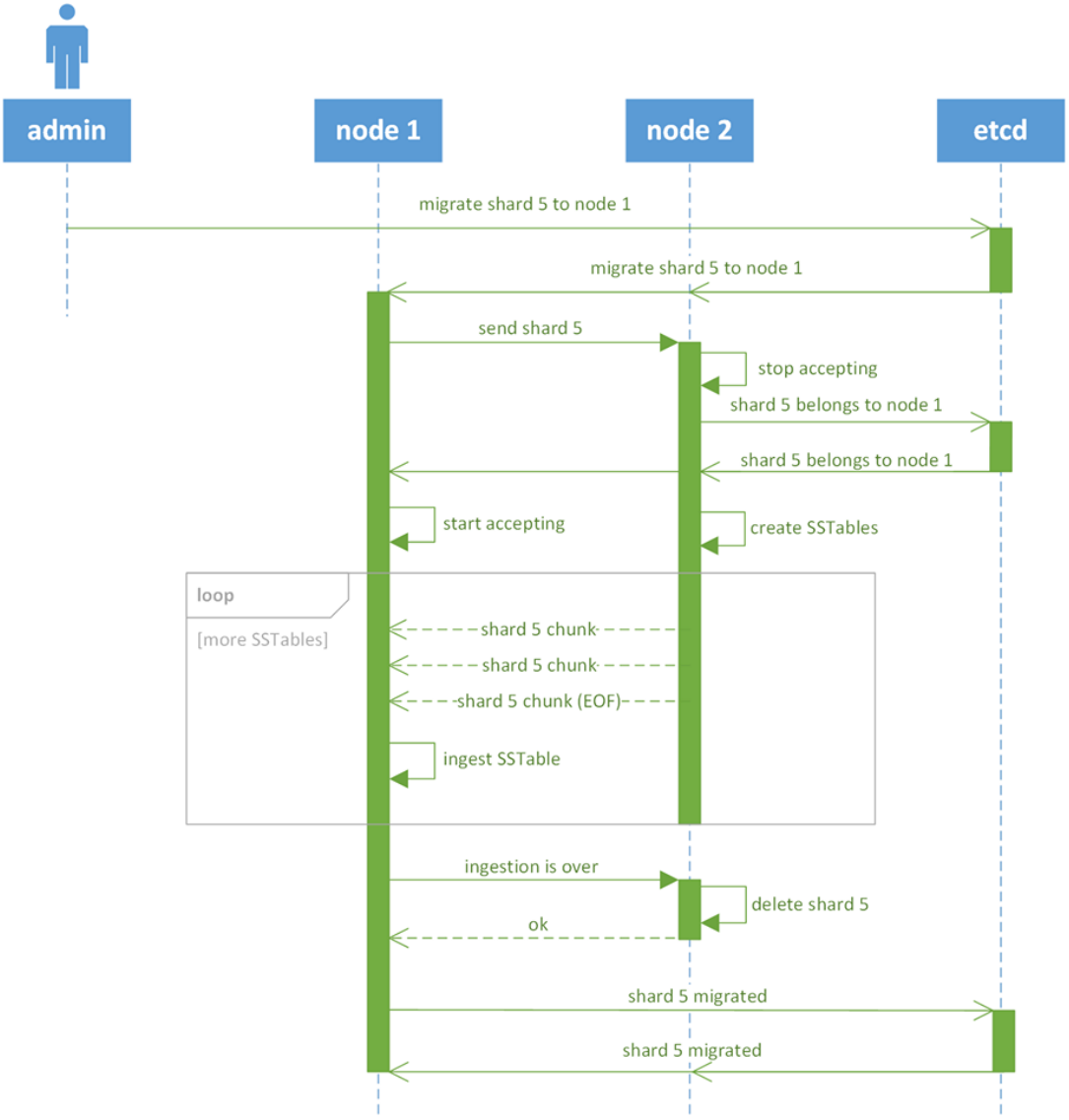
Στην κατάσταση `MIGRATING`, λαμβάνουν χώρα οι μεταναστεύσεις (migrations) των shards από τον έναν κόμβο στον άλλο. Μόλις οι κόμβοι ενημερωθούν για την αλλαγή αυτή στην κατάσταση της συστοιχίας, ελέγχουν αν πρέπει να λάβουν κάποια shard και αρχίζουν να τα ζητάνε ένα ένα. Ο παλιός υπεύθυνος ενός shard, όταν αυτό του ζητηθεί, κάνει τα εξής:

- Σταματάει να δέχεται αιτήματα για το συγκεκριμένο shard.
- Ορίζει ως υπεύθυνο του shard τον νέο.
- Ενημερώνει το etcd.
- Περιμένει να τελειώσουν τα αιτήματα που είχαν ήδη αρχίσει.
- Ετοιμάζει τα SSTables που θα στείλει. Συγκεκριμένα, συμπυκνώνει όλο το shard σε μια σειρά ταξινομημένων αρχείων, με τη βοήθεια ενός επαναλήπτη. Αυτή η διαδικασία είναι απαραίτητη καθώς ο νέος υπεύθυνος θα εισάγει τα SSTables στη βάση του LSM-δέντρου (βλ. παράγραφο 2.1.4), και επειδή η εισαγωγή γίνεται σε ένα συγκεκριμένο επίπεδο, τα αρχεία δεν γίνεται να έχουν επικάλυψη.

Στο μεταξύ ο νέος ενημερώνεται από το etcd, και αρχίζει να δέχεται αιτήματα. Υπάρχει επομένως ένα μικρό διάστημα που αρνούνται και οι δύο. Αυτό είναι απαραίτητο για να μην υπάρχουν συνθήκες συναγωνισμού (race conditions) σχετικά με τη σειρά εκτέλεσης των εγγραφών.

Ο παλιός υπεύθυνος, αρχίζει να στέλνει τα SSTables σε αύξουσα σειρά, ενώ ο νέος ένα ένα τα λαμβάνει και τα εισάγει στη βάση. Η εισαγωγή πραγματοποιείται στο τελευταίο επίπεδο του LSM-δέντρου ώστε να μην αντικατασταθούν μεταγενέστερες εγγραφές. Εάν το RocksDB δεν παρείχε αυτήν τη δυνατότητα, θα έπρεπε να υλοποιηθεί κάποιος περίπλοκος μηχανισμός, οπού οι νέες εγγραφές θα πραγματοποιούνταν ξεχωριστά και στο τέλος της εισαγωγής θα συγχωνεύονταν στο column family του shard. Κάτι τέτοιο θα είχε μεγάλη επιβάρυνση και θα προκαλούσε περαιτέρω συνθήκες συναγωνισμού.

Όταν εισαχθεί και το τελευταίο SSTable, ο νέος υπεύθυνος ενημερώνει τον παλιό για να διαγράψει το shard με ασφάλεια, καθώς και το etcd. Αν δεν υπάρχει άλλο shard υπό μετανάστευση, θέτει ξανά την κατάσταση της συστοιχίας σε `RUNNING`.



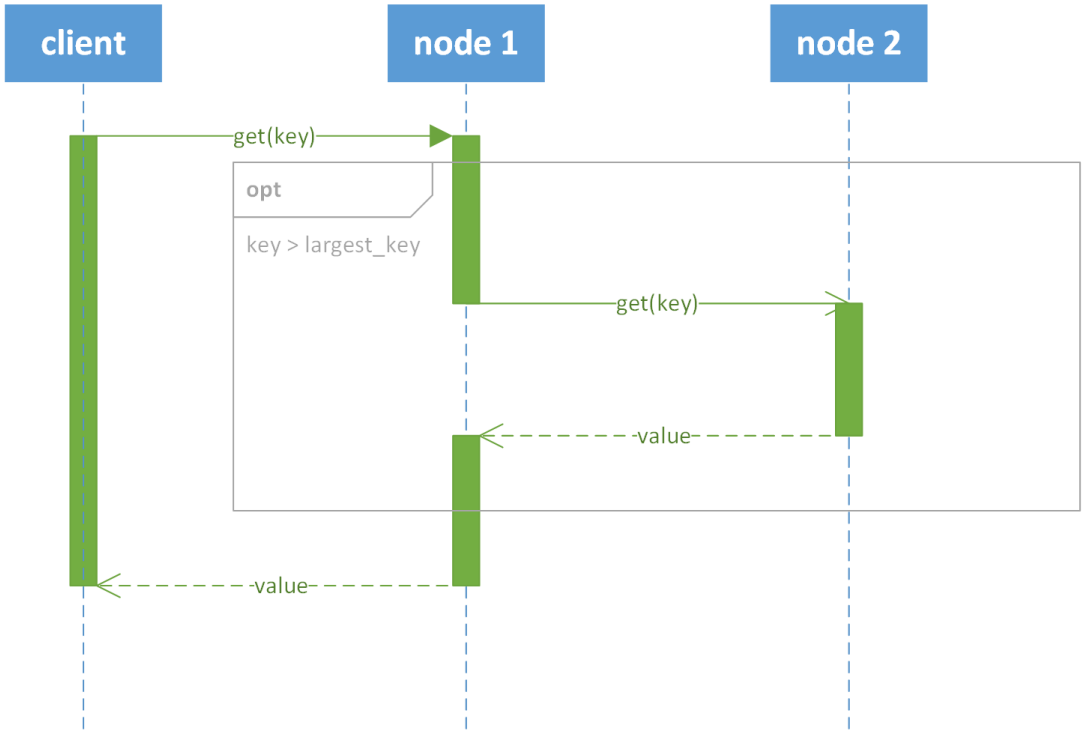
Σχήμα 3.3: *Ακολουθιακό διάγραμμα μετανάστευσης*

Ενώ λαμβάνει ένα shard, ένας κόμβος εξυπηρετεί αιτήματα χωρίς να καταλαβαίνει κάποια διαφορά ο πελάτης (εκτός από μια αύξηση στην καθυστέρηση λόγω αυξημένης κίνησης στο δίκτυο).

Τα SSTables που στέλνονται είναι ταξινομημένα τόσο στα περιεχόμενα τους, όσο και μεταξύ τους, και ανά πάσα στιγμή είναι γνωστό το μεγαλύτερο από τα κλειδιά που έχουν εισαχθεί στη βάση (το μεγαλύτερο κλειδί του τελευταίου SSTable).

Επομένως, αιτήματα για εγγραφές ή διαγραφές στο νέο shard, εκτελούνται κανονικά. Για να πραγματοποιηθεί κάποια ανάγνωση, ο κόμβος ελέγχει αν το ζητούμενο κλειδί βρίσκεται στη βάση. Αν βρεθεί ή αν βρίσκεται στο εύρος των κλειδιών που έχουν λη-

φθεί, απαντάει. Διαφορετικά ενδέχεται να το έχει ο παλιός, οπότε το ζητάει και προωθεί την απάντησή του. Μέχρι να φτάσει στον παλιό το αίτημα, υπάρχει μια μικρή πιθανότητα να έχει ολοκληρωθεί η μετανάστευση και εκείνος να έχει διαγράψει όλο το shard. Σε αυτήν την περίπτωση ο νέος ελέγχει ξανά τη βάση και απαντάει.



Σχήμα 3.4: *Get by proxy*

Σημειώνουμε ότι υπάρχει ένα σφάλμα στον παραπάνω σχεδιασμό. Έστω ότι ο νέος υπεύθυνος κόμβος, με το που δημιουργήσει το column family για το νέο shard, και πριν προλάβει να εισάγει παλιότερα SSTables, λάβει ένα αίτημα για διαγραφή κάποιου κλειδιού. Επειδή το κλειδί αυτό δε βρίσκεται πουθενά στη βάση, η διαγραφή θα αγνοηθεί και δε δημιουργήσει tombstone. Όταν τελικά πραγματοποιηθεί η εισαγωγή, εάν το κλειδί αυτό υπάρχει στα παλιά δεδομένα, θα παραμείνει στη βάση παρόλο που έχει ζητηθεί η διαγραφή του.

Στο RocksDB υπάρχει η επιλογή να διατηρούνται τα tombstones που έχουν δημιουργηθεί από μια χρονική στιγμή και έπειτα στη βάση, μέχρι ο χρήστης να ανανεώσει την τιμή αυτής της χρονικής στιγμής[5]. Ιδανικά θα θέλαμε να είναι δυνατόν να απενεργοποιούμε τελείως αυτή τη λειτουργία για το συγκεκριμένο column family, αμέσως

[5] `https://github.com/facebook/rocksdb/pull/2999`

μόλις ολοκληρωθεί η μετανάστευση. Η δυνατότητα για τη δυναμική αλλαγή αυτής της ρύθμισης προβλέπεται να υλοποιηθεί στο μέλλον, επομένως επιλέξαμε προς το παρόν να αγνοήσουμε το παραπάνω πρόβλημα.

4 Υλοποίηση

Η εφαρμογή crocks αποτελείται από τρία μέρη:

- Το εκτελέσιμο `crocks` που αποτελεί τον διακομιστή (server), δηλαδή έναν κόμβο της συστοιχίας.
- Τη βιβλιοθήκη `libcrocks` που παρέχεται είτε σε μορφή δυναμικής διασύνδεσης (`libcrocks.so`), είτε σε μορφή στατικής διασύνδεσης (`libcrocks.a`), και τα σχετικά αρχεία κεφαλίδας (header files) που παρέχουν μια Διεπαφή Προγραμματισμού Εφαρμογής (Application Programming Interface — API) για την υλοποίηση πελατών.
- Το εκτελέσιμο `crocksctl`, ένα εργαλείο που πραγματοποιεί αιτήματα προς τη συστοιχία για εγγραφές ή αναγνώσεις, και διάφορες διαχειριστικές εργασίες.

4.1 Κοινόχρηστα δεδομένα

Οι πληροφορίες που αφορούν τη συστοιχία, αποθηκεύονται στην συστοιχία etcd, σε ένα σειριοποιημένο protobuf μήνυμα τύπου `ClusterInfo`, στο κλειδί "info". Οι κόμβοι παρακολουθούν αυτό το κλειδί, οπότε σε περίπτωση που γίνει κάποια αλλαγή, το etcd στέλνει σε όλους την τελευταία έκδοση.

Ο πλήρης ορισμός του μηνύματος είναι ο εξής:

```
message NodeInfo {
  string address = 1;
  int32 id = 2;
  int32 num_shards = 3;
  bool available = 4;
  bool remove = 5;
}

message ShardInfo {
  int32 master = 1;
  bool migrating = 2;
  int32 from = 3;
  int32 to = 4;
}

message ClusterInfo {
  enum State {
    INIT = 0;
    RUNNING = 1;
    MIGRATING = 2;
  }
  State state = 1;
  int32 num_nodes = 2;
  repeated NodeInfo nodes = 3;
  repeated ShardInfo shards = 4;
}
```

Listing 4.1: *Ορισμός ClusterInfo*

Για να πραγματοποιηθεί η οποιαδήποτε ενημέρωση, χρησιμοποιείται μια συναλλαγή. Δηλαδή ανακτάται η τελευταία έκδοση του `ClusterInfo`, ενημερώνεται τοπικά, και στέλνεται στο etcd για να πραγματοποιηθεί μόνο εφόσον δεν έχει ανανεωθεί από την τελευταία ανάκτηση. Αν δεν ικανοποιείται αυτή η προϋπόθεση, η συναλλαγή αποτυγχάνει και η διαδικασία επαναλαμβάνεται όσες φορές χρειαστεί μέχρι να πετύχει.

Παραδείγματα τέτοιων συναλλαγών είναι η προσθήκη ενός νέου κόμβου, ή το αίτημα για αφαίρεση ενός υπάρχοντος κόμβου, όπου ο διαχειριστής θέτει το πεδίο `remove` του κόμβου σε true.

Κατά τη διαδικασία της μετανάστευσης ενός shard που ανήκει στον κόμβο 1, και πρόκειται να δοθεί στον κόμβο 2, το αντίστοιχο `ShardInfo` μήνυμα περνάει από τα εξής στάδια:

- Αρχικά.

  ```
  master: 1
  migrating: false
  ```

- Ο διαχειριστής ανακατανέμει τα shards.

  ```
  master: 1
  migrating: true
  from: 1
  to: 2
  ```

- Ο κόμβος 1 ζητάει το shard από τον κόμβο 2 και εκείνος του το παραχωρεί.

  ```
  master: 2
  migrating: true
  from: 1
  to: 2
  ```

- Ο κόμβος 2 αφού το λάβει και το προσθέσει, ανακοινώνει το τέλος της μετανάστευσης.

  ```
  master: 2
  migrating: false
  ```

4.2 Διακομιστής

Ο διακομιστής χρησιμοποιεί το ασύγχρονο API που παρέχει το gRPC, και εξυπηρετεί αιτήματα χρησιμοποιώντας ένα προκαθορισμένο πλήθος νημάτων, επιλεγμένο από το διαχειριστή. Εάν επιλέγαμε το σύγχρονο API η υλοποίηση θα ήταν πολύ ευκολότερη αλλά όπως αναφέρθηκε στην παράγραφο 2.3, αυτό θα απαιτούσε ένα ξεχωριστό νήμα

για κάθε διαφορετικό αίτημα, ενδεχομένως μπλοκαρισμένο περιμένοντας κάποιο μήνυμα από το δίκτυο, με αποτέλεσμα να χρησιμοποιούνται πόροι που θα μπορούσε να αξιοποιήσει το RocksDB, και να πέφτει η συνολική απόδοση του συστήματος.

Για κάθε διαφορετικό τύπο αιτήματος, ορίζουμε μία αντίστοιχη κλάση, κάθε μια από τις οποίες υλοποιεί μια μέθοδο `Proceed`, που προχωράει την εξέλιξη του αιτήματος όσο είναι δυνατόν, μέχρι να χρειαστεί περαιτέρω δικτυακή επικοινωνία. Το κάθε νήμα έχει μια ουρά και αρχικοποιεί ένα στιγμιότυπο για την κάθε μια από τις κλάσεις.

Θα αναφέρουμε ενδεικτικά τη διαδικασία της εξυπηρέτησης ενός αιτήματος εισαγωγής (put). Οι αντίστοιχοι ορισμοί του RPC και των μηνυμάτων είναι οι εξής:

```
rpc Put(KeyValue) returns (Response) {}

message KeyValue {
  bytes key = 1;
  bytes value = 2;
}

message Response {
  int32 status = 1;
}
```

Listing 4.2: *Protobuf ορισμοί για αίτηματα εισαγωγής*

Η κλάση που αντιστοιχεί στα αιτήματα εισαγωγής, δηλώνει στον constructor την πρόθεση της να εξυπηρετήσει κάποιο αίτημα, καλώντας την αντίστοιχη μέθοδο της κλάσης `AsyncService` του gRPC. Θέτει ως αναγνωριστικό, τη διεύθυνση μνήμης της μεθόδου `Proceed` του συγκεκριμένου στιγμιότυπου, και δίνει ακόμη τη διεύθυνση μνήμης ενός στιγμιότυπου του protobuf μηνύματος `KeyValue`, περιμένοντας από το gRPC να τοποθετήσει εκεί το αίτημα. Μόλις αυτό φτάσει στον κόμβο, και αντιγραφεί στην τελική του θέση, το gRPC προσθέτει στην ουρά το αναγνωριστικό. Το νήμα που διαβάζει από την ουρά, το αφαιρεί ως δείκτη τύπου `void*`, το κάνει cast σε συνάρτηση τύπου `Proceed`, και την καλεί.

Στη συνάρτηση `Proceed` δημιουργείται νέο στιγμιότυπο της κλάσης για να εξυπηρετήσει το επόμενο αίτημα. Διαβάζεται το μήνυμα, καλείται η συνάρτηση `Put` του RocksDB, και στη συνέχεια ζητείται από το gRPC να στείλει στον πελάτη ένα μήνυμα τύπου `Response`, με την τιμή που επέστρεψε το RocksDB. Ως τότε η συνάρτηση επι-

στρέφει, για να κληθεί εκ νέου και να συνεχίσει από εκεί που είχε μείνει, όταν το gRPC τοποθετήσει ξανά το αναγνωριστικό στην ουρά. Για να γνωρίζει από που πρέπει να συνεχίσει, οι κλάσεις αυτές διατηρούν ένα `enum`[1] που ανά πάσα στιγμή υποδηλώνει την κατάσταση στην οποία βρίσκεται η εξέλιξη του αιτήματος. Στην περίπτωση των Put, είναι ορισμένο ως "`enum CallStatus { REQUEST, FINISH };`". Κάθε φορά που καλείται η `Proceed` αλλάζει η κατάσταση.

Το `CallStatus` σε αυτήν την περίπτωση ήταν απλό επειδή η αντίστοιχη διαδικασία είναι απλή. Στην αντίστοιχη κλάση για τις μεταναστεύσεις, το `CallStatus` είναι ορισμένο ως "`enum CallStatus { REQUEST, READ, WRITE, DONE, FINISH };`" καθώς η διαδικασία λαμβάνει τόσο ως είσοδο όσο και ως έξοδο ροή μηνυμάτων και περνάει από περισσότερα στάδια μέχρι να ολοκληρωθεί (βλ. παράγραφο 3.3.3).

Κάθε κλάση υλοποιεί ακόμη τη μέθοδο `OnDone`, η οποία καλείται αυτόματα μετά την ολοκλήρωση της κλήσης, και αναλαμβάνει να αποδεσμεύσει την μνήμη που καταλαμβάνουν ορισμένες εσωτερικές δομές δεδομένων, καθώς και το το ίδιο το στιγμιότυπο (`delete this`).

4.2.1 Ταυτοχρονισμός

Νήματα

Οι διακομιστές εξυπηρετούν αιτήματα χρησιμοποιώντας ένα συγκεκριμένο πλήθος νημάτων, επιλεγμένο από τον διαχειριστή.

Οι μεταναστεύσεις χρησιμοποιούν αρκετή επικοινωνία με τον δίσκο (IO), κυρίως λόγω της εγγραφής των SSTables, τόσο στον κόμβο που στέλνει όσο και σε αυτόν που λαμβάνει το shard. Για το λόγο αυτό, πραγματοποιούνται σε ξεχωριστό νήμα, ώστε να μην μπλοκάρουν τα νήματα που εξυπηρετούν αιτήματα πελατών, περιμένοντας την ολοκλήρωση του IO.

Οι δομές δεδομένων που δεν διαθέτουν ασφάλεια νήματος (thread-safety), (protocol buffers, `std::vector`, `std::unordered_map` κτλ.) προστατεύονται από κλειδώματα, με ό,τι επίπτωση στην απόδοση αυτό συνεπάγεται. Όμως, αν είναι διαθέσιμη η έκδοση C++17 της C++, υπάρχει η επιλογή να χρησιμοποιούνται κοινόχρηστα κλειδώματα,

[1] Τα `enum` (enumerations) στη C++ είναι μεταβλητές που παίρνουν τιμές από μια προκαθορισμένη λίστα.

(`std::shared_mutex`). Σε αυτήν την περίπτωση, όσο δεν γίνεται κάποια αλλαγή στο etcd (πρακτικά όσο δεν πραγματοποιούνται μεταναστεύσεις), και η πρόσβαση στις δομές δεδομένων είναι μόνο για ανάγνωση, όλα τα νήματα παίρνουν τα κλειδώματα ως αναγνώστες ταυτόχρονα, χωρίς να μπλοκάρουν.

Συνθήκες συναγωνισμού κατά τη μετανάστευση

Για να σταλθεί ένα shard, προσπελάζονται όλα τα κλειδιά του σε αύξουσα σειρά, με τη βοήθεια ενός επαναλήπτη (iterator), και δημιουργείται μια σειρά από SSTables. Επειδή οι επαναλήπτες λειτουργούν πάνω σε ένα συνεπές στιγμιότυπο της βάσης που δημιουργείται κατά την κατασκευή τους, πρέπει να εξασφαλιστεί ότι μετά τη δημιουργία κάποιου, δεν θα τροποποιηθεί περαιτέρω το αντίστοιχο shard.

Αυτό δημιουργεί πιθανές συνθήκες συναγωνισμού. Πριν την κατασκευή του επαναλήπτη απαγορεύουμε μελλοντικές εγγραφές αλλά κάποια αιτήματα μπορεί να βρίσκονται ήδη σε εξέλιξη σε διαφορετικό νήμα. Για αυτόν το λόγο τα shards έχουν ένα μετρητή αναφορών αρχικοποιημένο στο 1. Πριν πραγματοποιηθεί κάποια εγγραφή στο shard, ο μετρητής αυξάνεται και στη συνέχεια μειώνεται. Αν έχουν απαγορευτεί οι εγγραφές (θέτοντας ατομικά μια αντίστοιχη boolean μεταβλητή σε `true`), η αύξηση αποτυγχάνει και ο πελάτης ενημερώνεται σχετικά.

Επομένως, πριν τη μετανάστευση απαγορεύουμε τις εγγραφές, μειώνουμε τον μετρητή και περιμένουμε να μηδενιστεί.

Επίσης για να μην διαγραφεί κάποιο shard κατά τη διάρκεια μιας ανάγνωσης που προορίζεται για το νέο υπεύθυνο, τα shards δεν διαγράφονται ρητά αμέσως μόλις σταλθούν, αλλά αφαιρούνται από την αντίστοιχη δομή δεδομένων στην οποία αποθηκεύονται ως `std::shared_ptr` (`std::unordered_map<int, std::shared_ptr<Shard>>`). Τα `shared_ptr` είναι δομή που παρέχει η C++, που διαγράφει το αντίστοιχο αντικείμενο μόλις χαθούν όλες οι αναφορές σε αυτό. Επομένως αν ο παλιός υπεύθυνος προλάβει να πάρει μια αναφορά στο αντίστοιχο shard, είναι βέβαιος ότι θα προλάβει να διαβάσει από αυτό πριν διαγραφεί.

```
Usage: crocks [options]

Start a crocks server.

Options:
  -p, --path <path>      RocksDB database path.
  -o, --options <path>   RocksDB options file path.
  -H, --host <hostname>  Node hostname [default: localhost].
  -P, --port <port>      Listening port [default: chosen by OS].
  -e, --etcd <address>   Etcd address [default: localhost:2379].
  -t, --threads <int>    Number of serving threads [default: 2].
  -s, --shards <int>     Number of initial shards [default: 10].
  -d, --daemon           Daemonize process.
  -v, --version          Show version and exit.
  -h, --help             Show this help message and exit.
```

Listing 4.3: *Έξοδος της εντολής `crocks --help`*

4.3 Πελάτης

Όλες οι κλάσεις και οι συναρτήσεις του παρεχόμενου API, βρίσκονται στον χώρο ονομάτων (namespace) `crocks`. Η βασική οντότητα που επιτρέπει την υλοποίηση πελατών είναι η κλάση `Cluster`, η οποία είναι ορισμένη στο αρχείο κεφαλίδας `<crocks/cluster.h>` με τον εξής constructor:

```
Cluster(const Options& options, const std::string& address);
```

Η κλάση `Options`, που βρίσκεται στο αρχείο `<crocks/options.h>`, επιτρέπει την παραμετροποίηση της συμπεριφοράς του πελάτη. Προς το παρόν περιέχει δύο ρυθμίσεις: Εάν θα ενημερώνεται το etcd σε περίπτωση που κάποιος κόμβος βρεθεί να είναι μη διαθέσιμος, και εάν οι κλήσεις προς κάποιον μη διαθέσιμο κόμβο θα μπλοκάρουν μέχρι να επανέλθει ή θα επιστρέφουν άμεσα με το αντίστοιχο `status`.

```
struct Options {
  // If true, when a node is detected to be unavailable
  // and etcd is not yet aware, the client updates it.
  bool inform_on_unavailable = true;
```

```
  // If true, after a status UNAVAILABLE is received, the client waits
  // until the cluster is healthy again, and then retries the request.
  bool wait_on_unhealthy = true;
};
```

Η συμβολοσειρά `address`, είναι η διεύθυνση στην οποία ακούει κάποιος κόμβος της συστοιχίας etcd, ή η πύλη etcd (π.χ. `"127.0.0.1:1234"`).

Η κλάση `Cluster`, παρέχει τις μεθόδους `Get()`, `Put()`, και `Delete()`, για την ανάγνωση, την εγγραφή, και τη διαγραφή κλειδιών αντίστοιχα.

```
Status Get(const std::string& key, std::string* value);
Status Put(const std::string& key, const std::string\& value);
Status Delete(const std::string& key);
```

Τέλος η κλάση `Status`, είναι ορισμένη στο αρχείο `<crocks/status.h>`, και περιλαμβάνει το status που επέστρεψε το gRPC κατά την εκτέλεση της αντίστοιχης κλήσης (`grpc_code()`), καθώς και το status που επέστρεψε το RocksDB, αν η κλήση ολοκληρώθηκε επιτυχώς (`rocksdb_code()`).

Ακολουθεί ένα παράδειγμα μιας απλής υλοποίησης πελάτη.

```
#include <assert.h>
#include <string>
#include <crocks/cluster.h>
#include <crocks/status.h>

int main() {
  crocks::Cluster* db = crocks::DBOpen("localhost:2379");
  crocks::Status s;
  std::string val;

  s = db->Put("key", "value");
  assert(s.ok());

  s = db->Get("key", &val);
  assert(s.ok());
  assert(val == "value");

  s = db->Delete("key");
  assert(s.ok());
```

```
  s = db->Get("key", &val);
  assert(s.IsNotFound());

  delete db;

  return 0;
}
```

Listing 4.4: *Παράδειγμα υλοποίησης πελάτη*

Το παραπάνω πρόγραμμα υποθέτει ότι υπάρχει μια συστοιχία `crocks`, τις οποίας οι πληροφορίες βρίσκονται σε μια συστοιχία etcd, και κάποιος κόμβος της τελευταίας δέχεται αιτήματα από πελάτες στη διεύθυνση `localhost:2379`. Η εκτέλεση του προγράμματος θα πρέπει να τερματίσει επιτυχώς, χωρίς έξοδο.

```
$ g++ test_crocks.cc -o test_crocks -lcrocks
$ ./test_crocks
```

4.4 Υλοποίηση των write batches

Ως προς την υλοποίηση των write batches, υπάρχει ένα σημαντικό πρόβλημα. Στο RocksDB τα write batches, όπως έχουμε αναφέρει στην παράγραφο 2.1.4, εκτελούνται ατομικά, κάτι που είναι δύσκολο να εγγυηθούμε στη δική μας κατανεμημένη εφαρμογή. Αν ζητηθεί από τον κάθε κόμβο να δεσμεύσει το δικό του μέρος ενός write batch, και σε κάποιον αυτή η διαδικασία αποτύχει, είναι αδύνατο να επαναφέρουμε τα ήδη δεσμευμένα δεδομένα των άλλων κόμβων.

Επίσης, ιδανικά θα πρέπει να εξασφαλίσουμε ότι οι δεσμεύσεις διαφορετικών write batches από διαφορετικούς πελάτες, θα εκτελεστούν με την ίδια σειρά στον κάθε κόμβο, ανεξάρτητα από τη σειρά με την οποία θα φτάσουν.

Επιλέξαμε σε πρώτη φάση να τα υλοποιήσουμε χωρίς να λάβουμε υπόψη μας αυτές τις εγγυήσεις, ώστε να κερδίσουμε τουλάχιστον την αυξημένη απόδοση, και να τις αναβάλλουμε για μελλοντική επέκταση (βλ. παράγραφο 6). Έτσι, αν κάποιος πελάτης επιθυμεί να ομαδοποιήσει ορισμένες εγγραφές οι οποίες δεν έχουν κάποια σαφή σειρά με την οποία πρέπει να πραγματοποιηθούν, ή είναι βέβαιος ότι δεν σκοπεύει κάποιος

άλλος να γράψει στα ίδια κλειδιά, είναι ελεύθερος να το κάνει αρκεί να γνωρίζει τους περιορισμούς.

```
rpc Batch(stream BatchBuffer) returns (stream Response) {}

message BatchUpdate {
  enum Operation {
    PUT = 0;
    DELETE = 1;
    SINGLE_DELETE = 2;
    MERGE = 3;
    CLEAR = 4;
  }
  Operation op = 1;
  bytes key = 2;
  bytes value = 3;
}

message BatchBuffer {
  repeated BatchUpdate updates = 1;
}

message Response {
  int32 status = 1;
}
```

Listing 4.5: *Protobuf ορισμοί για τα write batches*

```
class WriteBatch {
 public:
  WriteBatch(Cluster* db);
  WriteBatch(Cluster* db, int threshold_low, int threshold_high);
  ~WriteBatch();

  void Put(const std::string& key, const std::string& value);
  void Delete(const std::string& key);
  void Clear();

  Status Write();
}
```

Listing 4.6: *Το API των write batches που παρέχεται στους πελάτες*

Ο κάθε κόμβος λαμβάνει ροή από protobuf μηνύματα τύπου `BatchBuffer` κάθε ένα από τα οποία περιέχει εγγραφές για κάποιο από τα shards που του ανήκουν. Στην αρχή δημιουργεί ένα νέο write batch. Μόλις λάβει το πρώτο μήνυμα για κάποιο shard, προσπαθεί να αυξήσει το μετρητή αναφορών για το συγκεκριμένο. Αν τα καταφέρει, αν δηλαδή το shard του ανήκει, και δεν έχουν απαγορευτεί οι εγγραφές επειδή πρόκειται να μεταναστεύσει σε άλλο κόμβο, στέλνει μήνυμα με status 0. Διαφορετικά στέλνει ένα διαφορετικό status, και ο πελάτης στέλνει ξανά το ίδιο μήνυμα στον νέο υπεύθυνο. Στη συνέχεια προσθέτει όλες τις εγγραφές του μηνύματος στο write batch, και επαναλαμβάνει την ίδια διαδικασία μέχρι ο πελάτης να ανακοινώσει ότι είναι έτοιμος για τη δέσμευση, την οποία και πραγματοποιεί.

Για να μεγιστοποιηθεί η απόδοση ο πελάτης χρησιμοποιεί το ασύγχρονο gRPC API. Δημιουργεί ένα `BatchBuffer` μήνυμα για κάθε shard, και αρχίζει να τα γεμίζει. Υπάρχουν δύο κατώφλια στο μέγεθος των μηνυμάτων αυτών, τα οποία εξετάζονται μετά από κάθε προσθήκη. Αν δεν έχει ξεπεραστεί το πρώτο, δεν γίνεται κάποια άλλη ενέργεια.

Αν το μήνυμα βρίσκεται ανάμεσα στα δύο κατώφλια, γίνεται μια προσπάθεια να σταλεί. Συγκεκριμένα, αν δεν υπάρχει ακόμα ενεργή κλήση προς τον αντίστοιχο κόμβο, ο πελάτης ξεκινάει μια νέα, στέλνει το μήνυμα χωρίς να μπλοκάρει, και δημιουργεί το επόμενο `BatchBuffer` προτού επιστρέψει από την κληθείσα μέθοδο. Εάν υπάρχει κλήση και έχει σταλεί το πρώτο μήνυμα για το συγκεκριμένο shard, δεν μπορεί να

στείλει το επόμενο αν δε λάβει επιβεβαίωση για το ότι ο κόμβος είναι όντως ο σωστός υπεύθυνος. Επομένως ελέγχει την ουρά του gRPC για το σχετικό αναγνωριστικό χωρίς να μπλοκάρει, και μόνο αν βρεθεί συνεχίζει. Διαφορετικά επιστρέφει χωρίς να στείλει το μήνυμα. Επίσης, επειδή το gRPC δεν επιτρέπει να βρίσκονται ταυτόχρονα δύο εκκρεμή μηνύματα στο δίκτυο για την ίδια κλήση, ελέγχει και για το αναγνωριστικό της προηγούμενης αποστολής.

Τέλος, εάν έχει ξεπεραστεί το μεγάλο κατώφλι, ο πελάτης περιμένει στην ουρά για όλα τα απαραίτητα εκκρεμή αναγνωριστικά, και η αντίστοιχη μέθοδος επιστρέφει μόνο αφού σταλεί το μήνυμα.

Τα κατώφλια αυτά είναι παραμετροποίησιμα και αν επιλεγούν κατάλληλα, είναι δυνατόν να μην μπλοκάρει ποτέ καμία κλήση κάποιας μεθόδου που προσθέτει εγγραφές στο write batch, αλλά μόνο η κλήση της `Write()`.

Όταν πια κληθεί η `Write()`, ο πελάτης στέλνει όσα τρέχοντα μηνύματα περιέχουν εγγραφές, μπλοκάροντας αναγκαστικά όπου είναι απαραίτητο (π.χ. σε διαδοχικά μηνύματα για διαφορετικό shard στον ίδιο κόμβο), και ανακοινώνει το τέλος των μηνυμάτων με τη χρήση της συνάρτησης `WritesDone()` του gRPC. Στη συνέχεια περιμένει για τα status, και επιστρέφει το πρώτο μη επιτυχές που θα συναντήσει, ή ένα θετικό σε περίπτωση που η δέσμευση πετύχει σε όλους τους κόμβους.

4.5 Διαχειριστικό εργαλείο `crocksctl`

Το `crocksctl` είναι μια εφαρμογή γραμμής εντολών, η οποία παίρνει σαν όρισμα μια εντολή, και αφού λάβει από το etcd τις πληροφορίες της συστοιχίας, την εκτελεί. Οι διαθέσιμες εντολές είναι οι εξής:

- **get, put, delete:** Πραγματοποιούν το αντίστοιχο αίτημα προς τη συστοιχία, και τυπώνουν το status που έλαβαν, καθώς και την τιμή στην περίπτωση της get.
- **run, migrate:** Θέτουν το πεδίο `state` του `ClusterInfo` σε `RUNNING` ή `MIGRATING` αντίστοιχα. Η εντολή migrate επίσης εκτελεί και την ανακατανομή των shards.

- **health:** Ελέγχει την κατάσταση του κάθε κόμβου και ενημερώνει το etcd για ενδεχόμενες αποτυχίες.

  Πιο συγκεκριμένα, πραγματοποιεί ένα αίτημα τύπου ping, δηλαδή ένα RPC με κενό μήνυμα τόσο στην είσοδο όσο και στην έξοδο, προς κάθε κόμβο της συστοιχίας. Τα αιτήματα μπορεί είτε να επιτύχουν είτε να αποτύχουν με gRPC status `UNAVAILABLE`. Σε κάθε περίπτωση το αποτέλεσμα τυπώνεται στην γραμμή εντολών, και αν διαφωνεί με την αναγραφόμενη τιμή στο etcd, αυτό ενημερώνεται.

```
info = etcd.get_info()
for each node in info.nodes:
    status = node.ping()
    if status == UNAVAILABLE:
        print(node, ": DOWN")
        if info.is_available(node):
            etcd.update(node, available=false)
    else:
        print(node, ": OK")
        if info.is_unavailable(node):
            etcd.update(node, available=true)
```

Listing 4.7: *Αλγόριθμος ελέγχου υγείας της συστοιχίας*

- **list, dump:** Οι εντολές αυτές τυπώνουν όλα τα κλειδιά που περιέχονται στην βάση σε αύξουσα σειρά, ενώ η εντολή dump τυπώνει και τις αντίστοιχες τιμές.

  Υλοποιούνται με τη χρήση επαναληπτών, οι οποίοι λειτουργούν ως εξής: Όλοι οι κόμβοι παίρνουν επαναλήπτη σε καθένα από τα column families, τους συγχωνεύουν με τη χρήση ενός σωρού (heap), και στέλνουν, όποτε ζητηθεί, τα επόμενα ζεύγη κλειδιού-τιμής. Ο πελάτης από την πλευρά του διατηρεί ένα buffer όπου τοποθετεί τα ζεύγη αυτά, επίσης συγχωνεύοντας τα. Ακόμη διατηρεί μετρητές για να γνωρίζει πόσα έχει διαθέσιμα από τον κάθε κόμβο, και όταν ξεπεράσουν ένα κατώφλι ζητάει τα επόμενα. Έτσι ελαχιστοποιεί τις πιθανότητες να αναγκαστεί να μπλοκάρει περιμένοντας μηνύματα.

  Η υλοποίηση των επαναληπτών έχει δύο προβλήματα. Αρχικά, όπως και στην περίπτωση των write batches, υπάρχει συνθήκη συναγωνισμού σχετικά με τη

σειρά με την οποία θα φτάσουν τα αιτήματα στους διαφορετικούς κόμβους. Είναι για παράδειγμα δυνατό, να πραγματοποιηθούν δύο εγγραφές σε διαφορετικούς κόμβους με μια συγκεκριμένη σειρά, και το στιγμιότυπο της βάσης που θα χρησιμοποιεί ο επαναλήπτης στον ένα κόμβο να περιλαμβάνει την μεταγενέστερη εγγραφή, ενώ ο επαναλήπτης στον άλλο να μην περιλαμβάνει τη προγενέστερη.

Επίσης, οι επαναλήπτες δεν λειτουργούν σωστά κατά τη διάρκεια της μετανάστευσης. Δηλαδή θα έπρεπε να λαμβάνουν και από τους δύο κόμβους τις τιμές για το υπό μετανάστευση shard, δίνοντας προτεραιότητα στον νέο υπεύθυνο αλλά κάτι τέτοιο δεν έχει ακόμα υλοποιηθεί.

Επομένως, οι επαναλήπτες προς το παρόν χρησιμοποιούνται περισσότερο για να βοηθήσουν στην αποσφαλμάτωση (debugging), παρά για να υλοποιήσουν όντως αιτήματα για λήψη εύρους κλειδιών.

- **clear:** Διαγράφει όλα τα κλειδιά της βάσης. Δημιουργεί έναν επαναλήπτη και διαγράφει το κάθε κλειδί, ομαδοποιώντας τις διαγραφές σε write batches. Όπως και παραπάνω, η εντολή αυτή προορίζεται μόνο για πειραματική χρήση.

- **remove:** Θέτει το πεδίο `remove` του ζητούμενου κόμβου στις πληροφορίες του etcd, σε true. Φυσικά για να αρχίσει να στέλνει τα shards του, πρέπει πρώτα να κληθεί η εντολή migrate ώστε με την ανακατανομή, να ανατεθούν σε διαφορετικούς κόμβους.

- **info:** Τυπώνει όλες τις πληροφορίες της συστοιχίας σε μια ευανάγνωστη μορφή. Στο ακόλουθο παράδειγμα υπήρχε μια αρχική συστοιχία τεσσάρων κόμβων και 40 shard (10 στον καθένα). Στη συνέχεια προστέθηκε ένας πέμπτος στον οποίο ανατέθηκαν 8 shards (2 από τον καθένα), και πραγματοποιήθηκαν οι μεταναστεύσεις. Τέλος προστέθηκε ένας έκτος κόμβος και ζητήθηκε η αφαίρεση του τρίτου, χωρίς όμως να ζητηθεί ακόμα να ξεκινήσουν οι μεταναστεύσεις.

```
state: RUNNING
nodes: 6
shards: 40
node 0:
  address: 10.94.250.138:50050
  shards: 2—9 (8)
```

```
    node 1:
      address: 10.94.250.138:50051
      shards: 12—19 (8)
    node 2:
      address: 10.94.250.138:50052
      shards: 22—29 (8)
      remove: true
    node 3:
      address: 10.94.250.138:50053
      shards: 32—39 (8)
    node 4:
      address: 10.94.250.138:50054
      shards: 0—1,10—11,20—21,30—31 (8)
    node 5:
      address: 10.94.250.138:50055
```

Listing 4.8: *Παράδειγμα εξόδου της εντολής `crocksctl info`*

```
Usage: crocksctl [options] command [args]...

A simple command line client for crocks.

Commands:
  get <key>          Get key.
  put <key> <value>  Put key.
  del <key>          Delete key.
  run                Change cluster state to RUNNING.
  migrate            Change cluster state to MIGRATING.
  health             Check the health of the cluster.
  list               Print every key.
  dump               Print every key—value pair.
  clear              Delete all keys.
  remove <id>        Remove node from the cluster.
  info               Print cluster info.

Options:
  —e, ——etcd <address>  Etcd address [default: localhost:2379].
  —h, ——help            Show this help message and exit.
```

Listing 4.9: *Έξοδος της εντολής `crocksctl --help`*

4.6 Επαναφορά από αποτυχίες

Η συστοιχία μπορεί να επανέρχεται από αποτυχίες κόμβων, δεδομένου ότι ο δίσκος είναι στην κατάσταση που τον άφησαν.[2]

4.6.1 Εντοπισμός αποτυχιών

Σε περίπτωση αποτυχίας κάποιου κόμβου, το etcd ενημερώνεται σχετικά, και το πεδίο `available` στην καταχώρηση για αυτόν τον κόμβο, παίρνει την τιμή `false`. Η ενημέρωση αυτή, πραγματοποιείται από τον πρώτο που θα το εντοπίσει (πελάτη, κόμβο ή διαχειριστή).

- Αν ένας πελάτης εκτελέσει ένα αίτημα και ενημερωθεί από το gRPC ότι ο κόμβος δεν είναι διαθέσιμος, ελέγχει το etcd. Αν έχει αλλάξει ο υπεύθυνος του shard, συνεχίζει κανονικά με τον νέο υπεύθυνο. Αν δεν έχει αλλάξει αλλά φαίνεται διαθέσιμος, ενημερώνει το etcd ότι πλέον είναι μη διαθέσιμος. Σε περίπτωση που γίνεται κάποιου είδους παρακολούθηση (monitoring) της συστοιχίας από τον διαχειριστή, η ενημέρωση αυτή είναι καλό να απενεργοποιηθεί για να μην επιβαρύνεται άσκοπα το etcd.
- Αν ένας κόμβος αποτύχει κατά τη διαδικασία της μετανάστευσης (είτε ενώ στέλνει είτε ενώ λαμβάνει), ενημερώνει το etcd ο άλλος κόμβος.
- Ο διαχειριστής μπορεί με τη χρήση της εντολής `health` του `cockroachctl`, να ελέγξει την κατάσταση του κάθε κόμβου και να ενημερώσει το etcd για ενδεχόμενες αποτυχίες (βλ. παράγραφο 4.5).

4.6.2 Συμπεριφορά πελατών κατά την αποτυχία

Αφού λάβει status `UNAVAILABLE` από το gRPC κάποιος πελάτης, είναι αναγκασμένος να περιμένει να επανέλθει η συστοιχία σε υγιή κατάσταση. Για να το πετύχει αυτό,

[2] Αν αποτύχει όλο το μηχάνημα και όχι απλά η διεργασία, και το RocksDB δεν έχει ρυθμιστεί ώστε να συγχρονίζει το αρχείο WAL με το δίσκο (fsync) μετά από κάθε εγγραφή, ενδέχεται μετά την επαναφορά να έχουν χαθεί εγγραφές που έχουν ανακοινωθεί ως επιτυχείς στους πελάτες.

ξεκινάει να παρακολουθεί το etcd για αλλαγές. Κάθε φορά που λαμβάνει κάποια ενημέρωση, ελέγχει αν είναι διαθέσιμοι όλοι οι κόμβοι και μόλις συμβεί αυτό, επαναλαμβάνει το αρχικό αίτημα.

Εναλλακτικά μπορεί να επιλέξει να επιστρέφει το status ως έχει, ώστε να χειριστεί την κατάσταση όπως επιθυμεί προτού αρχίσει την παρακολούθηση, χρησιμοποιώντας τη μέθοδο `void WaitUntilHealthy()` της κλάσης `Cluster`.

4.6.3 Επαναφορά κόμβων

Μόλις ένας κόμβος αρχικοποιηθεί, λαμβάνει τις πληροφορίες για τη συστοιχία από το etcd. Αν σε αυτές υπάρχει ένας καταχωρημένος κόμβος με τη διεύθυνση του, ο οποίος φαίνεται μη διαθέσιμος, υποθέτει ότι πρόκειται για τον ίδιο και ξεκινάει τη διαδικασίας της επαναφοράς. Αφού ανοίξει τη βάση και το RocksDB εκτελέσει την επαναφορά της, ελέγχει ότι τα shards που θα έπρεπε να του ανήκουν υπάρχουν τοπικά, και επαναφέρει τον εαυτό του σε διαθέσιμο, ενημερώνοντας το etcd. Σε διαφορετική περίπτωση τερματίζει με το αντίστοιχο μήνυμα.

Σε περίπτωση αποτυχίας κατά τη διάρκεια μιας μετανάστευσης, είτε αποτύχει ο κόμβος που στέλνει το shard είτε αυτός που το λαμβάνει, θα αναλάβει να επαναφέρει τη μετανάστευση ο δεύτερος, ζητώντας να ξεκινήσει από το τελευταίο SSTable που έλαβε. Επομένως, ως είσοδος στο RPC της μετανάστευσης, εκτός από το αναγνωριστικό του shard, δίνεται και το αναγνωριστικό του πρώτου SSTable που πρέπει να σταλεί.

```
message MigrateRequest {
  int32 shard = 1;
  // First SST to send. Specify a number higher than 0 to resume.
  int32 start_from = 2;
}
```

Στο 'default' column family του RocksDB (ξεχωριστά από τα column families των shards) αποθηκεύεται οτιδήποτε χρειάζεται να "θυμάται" ένας κόμβος, ώστε σε περίπτωση που επανέλθει από αποτυχία, να μπορέσει να ολοκληρωθεί η διαδικασία της μετανάστευσης ομαλά.

Έστω ότι ο κόμβος 2 επανέρχεται από κάποια αποτυχία, ελέγχει το etcd και βλέπει ότι πρέπει να λάβει το shard 5 από τον κόμβο 1.

Αν το shard 5 ανήκει στον 1 (`shard.from == shard.master`), ζητάει να πραγματοποιηθεί η μετανάστευση κανονικά. Το column family ενδέχεται να υπάρχει ήδη στην περίπτωση που πριν από την αποτυχία, είχε ξεκινήσει η μετανάστευση και είχε προλάβει να το φτιάξει. Αν δεν υπάρχει το φτιάχνει.

Αν το shard ανήκει στον ίδιο, σημαίνει πως είχε προχωρήσει η διαδικασία και πρέπει να συνεχίσει από εκεί που είχε μείνει, δηλαδή από το τελευταίο SSTable που έλαβε.

Μόλις φτάσει το τελευταίο κομμάτι ενός SSTable και είναι έτοιμο στο δίσκο, αποθηκεύεται στη βάση το όνομα του αρχείου (`shard_5_filename`), το μεγαλύτερο κλειδί που περιλαμβάνει (`shard_5_largest_key`), και το αναγνωριστικό του επόμενου SSTable που θα ζητηθεί (`shard_5_next_num`). Η εγγραφή αυτή πραγματοποιείται ατομικά, με τη χρήση ενός write batch, και στη συνέχεια το SSTable προστίθεται στη βάση.

Αν η αποτυχία συμβεί πριν προλάβει να ολοκληρωθεί αυτή η εγγραφή, ο κόμβος συμπεριφέρεται σαν να μην έλαβε ποτέ το αρχείο, και όταν επανέλθει το ζητάει ξανά. Αν συμβεί μετά από την εγγραφή, αλλά πριν πραγματοποιηθεί η εισαγωγή του αρχείου στο LSM-δέντρο, θα δει ότι το αρχείο `shard_5_filename` βρίσκεται ακόμα στον δίσκο, θα το εισάγει και θα συνεχίσει τη μετανάστευση από το `shard_5_next_num`. Αν συμβεί μετά από την εισαγωγή, δε θα βρει αρχείο, θα υποθέσει ότι είχε προλάβει να το εισάγει πριν την αποτυχία, και θα συνεχίσει πάλι από το `shard_5_next_num`.

Παραθέτουμε συνοπτικά τον αλγόριθμο σε ψευδοκώδικα:

```
for each shard I have to take:
  if shard does not belong to me:
    EnsureColumnFamily()
    RequestMigration(shard, start_from=0)
  else:
    if shard_5_next_num not found:
      RequestMigration(shard, start_from=0)
    else:
      if FileExists(shard_5_filename):
        Ingest(shard_5_filename)
      RequestMigration(shard, start_from=shard_5_next_num)
```

Ο κόμβος που στέλνει το shard, δε χρειάζεται να θυμάται σε πιο στάδιο της μετανάστευσης βρισκόταν πριν την αποτυχία, αφού θα αναλάβει να συνεχίσει τη διαδικασία

ο άλλος κόμβος. Αυτό που θυμάται είναι αν πρόλαβε να ετοιμάσει τα SSTables και αν ναι, πόσα ήταν και ποιο είναι το μεγαλύτερο κλειδί του καθενός. Επίσης, φροντίζει να μην διαγράψει κάποιο αρχείο πριν σιγουρευτεί ότι δεν θα χρειαστεί να το στείλει ξανά.

5 Πειραματική αποτίμηση

5.1 Αναμενόμενη απόδοση

Λόγω της ενίσχυσης εγγραφής, το RocksDB δεν μπορεί να αξιοποιήσει όλη την ταχύτητα εγγραφών που παρέχει ο δίσκος. Συγκεκριμένα, υπάρχει μια μέγιστη ταχύτητα εγγραφών πραγματικών δεδομένων που μπορούν να αποθηκευτούν, που ισούται με τη μέση ταχύτητα των *ακολουθιακών* (sequential) εγγραφών του δίσκου [1], προς την ενίσχυση εγγραφής.

Ας υποθέσουμε για παράδειγμα ότι έχουμε δίσκους με δυνατότητα εγγραφής 400 MB/sec και μια βάση δεδομένων μεγέθους 2.5 TB. Αυτό σε έναν κόμβο απαιτεί 6 επίπεδα, και η ενίσχυση εγγραφής είναι περίπου 32 (βλ. παράγραφο 2.1.3). Σε αυτήν την περίπτωση, η μέγιστη ταχύτητα εγγραφής πραγματικών δεδομένων είναι $400/32 = 12.5$ MB/sec. Αν οι κόμβοι αυξηθούν σε 1000, ο καθένας θα διατηρεί τα 2.5 GB της βάσης, κάτι που απαιτεί μόνο 3 επίπεδα. Άρα η ενίσχυση εγγραφής θα είναι 11, και η συνολική ταχύτητα εγγραφών $1000 * (400/11) \simeq 36.36$ GB/sec.

Τα παραπάνω προϋποθέτουν ότι ο κάθε κόμβος διατηρεί μόνο ένα shard, άρα χρησιμοποιεί ένα column family. Με περισσότερα shards στον κάθε κόμβο, το κάθε column family θα καταλάμβανε λιγότερο χώρο, με αποτέλεσμα η ενίσχυση εγγραφής να είναι χαμηλότερη και η ταχύτητα αυξημένη.

[1] Το RocksDB πραγματοποιεί μόνο ακολουθιακές εγγραφές αφού γράφει αρχεία και δεν τα επεξεργάζεται ξανά, και οι ακολουθιακές εγγραφές είναι εν γένει αρκετά πιο γρήγορες από τις τυχαίες.

5.2 Benchmarks

Για την εκτέλεση των μετρήσεων χρησιμοποιήσαμε εικονικές μηχανές που τις παρείχε η υπηρεσία AWS [2] (Amazon Web Services) της Amazon.

Αρχικά πειραματιστήκαμε με δύο μικρές εικονικές μηχανές, με ταχύτητα δικτύου 1 Gbps, δηλαδή 128 MB/sec. Οι εικονικές μηχανές διέθεταν έναν επεξεργαστή, 1 GB RAM, και δίσκο 8 GB με ταχύτητα εγγραφών περίπου 65 MB/sec. Επομένως σε κάθε περίπτωση (ακόμα και αν δεν υπήρχε καθόλου ενίσχυση εγγραφής), σημείο συμφόρησης αποτελούσε ο δίσκος.

Η μία εικονική μηχανή ανέλαβε το ρόλο του διακομιστή, δηλαδή μιας συστοιχίας του ενός κόμβου, και η δεύτερη το ρόλο του πελάτη, καθώς και του etcd.

Χρησιμοποιήθηκαν κλειδιά των 16 byte και τιμές των 4096 bytes (4 KB). Τα κλειδιά ήταν συγκεκριμένα, και τόσα στο πλήθος ώστε το μέγεθος της βάσης να μην ξεπερνάει τα 2 GB. Αυτό ήταν απαραίτητο καθώς ο διαθέσιμος χώρος ήταν περίπου 5 GB και η ενίσχυση χώρου μεγαλύτερη από 2, καθώς για τέτοια μεγέθη χρησιμοποιούνται μόνο δύο επίπεδα στο LSM-δέντρο τα οποία καταλαμβάνουν περίπου τον ίδιο χώρο.

Ο κόμβος είχε 5 shards, άρα 5 column families, μία σχετικά χαμηλή τιμή που επιλέχθηκε λόγω της περιορισμένης μνήμης των μηχανημάτων. Το κάθε ένα καταλάμβανε 2 GB / 5 = 410 MB, και η ενίσχυση εγγραφής μετρήθηκε κοντά στο 3.5. Η μέτρηση αυτή πραγματοποιείται αυτόματα στον κάθε κόμβο χρησιμοποιώντας το `EventListener` API του RocksDB [3]. Συγκεκριμένα, ο κόμβος διατηρεί δύο μετρητές: το πλήθος των byte που ζητήθηκε να γραφτούν, και το πλήθος των bytes που γράφτηκαν. Κάθε φορά που μεταφέρεται ένα MemTable στο δίσκο ή ολοκληρώνεται μία συμπύκνωση, καλείται η αντίστοιχη συνάρτηση που ανανεώνει αυτούς τους μετρητές και υπολογίζει και τυπώνει το λόγο τους, δηλαδή την ενίσχυση εγγραφής.

Για κάθε αίτημα επιλέγαμε έναν τυχαίο αριθμό από τους διαθέσιμους, τον οποίο κωδικοποιούσαμε σε μια συμβολοσειρά με ASCII χαρακτήρες, και το κατάλληλο πλήθος μηδενικών για πρόθεμα. Για παράδειγμα `"000000000491827"`.

Οι τιμές επιλέγονταν τυχαία από έναν πίνακα χαρακτήρων μεγέθους 1 MB. (`char blob_[1024 * 1024]`). Δηλαδή με N το μέγεθος της κάθε τιμής, η γεννήτρια τιμών

[2] `https://aws.amazon.com/`

[3] `https://github.com/facebook/rocksdb/wiki/EventListener`

επέλεγε ένα τυχαίο δείκτη στο εύρος $[0, 1024^2 - N)$ και επέστρεφε N bytes ξεκινώντας από αυτόν το δείκτη.

```
std::string NextValue() {
  // Random index for blob_ small enough to not overflow
  int start = rand() % (kBlobSize − value_size_);
  return std::string(&blob_[start], value_size_);
}
```

Listing 5.1: *Ανάκτηση τυχαίας τιμής*

Εκτελέσαμε 4 διαφορετικά benchmarks: εγγραφές, αναγνώσεις, ταυτόχρονες εγγραφές και αναγνώσεις και εγγραφές με τη χρήση του write batch.

- Αρχικά η βάση γέμισε με τη χρήση write batch. Αυτό χρειάστηκε περίπου 70 sec, άρα η ταχύτητα ήταν 2 GB / 70 sec $\simeq$ 30 MB/sec. Αυτό ήταν αναμενόμενο, καθώς τα αρχεία δεν είχαν επικάλυψη, και δε χρειάστηκε να πραγματοποιηθεί καμία συμπύκνωση. Η ενίσχυση εγγραφής ήταν 2 (1 για το WAL και ένα για το SSTable), και παρατηρήσαμε τη μισή ταχύτητα από την πραγματική του δίσκου.[4]
- Στη συνέχεια, με γεμάτη πια βάση, ο πελάτης πραγματοποιούσε τυχαίες εγγραφές ή αναγνώσεις, τη μία μετά την άλλη για λίγα λεπτά. Αρχικά με 1 νήμα και έπειτα με περισσότερα ώστε να φανεί η συμπεριφορά με περισσότερους πελάτες. Με ένα νήμα, κατάφερνε να πραγματοποιεί περίπου 2000 εγγραφές ανά δευτερόλεπτο (8 MB/sec) ενώ με περισσότερα νήματα έφτασε στις 5200 (20 MB/sec), κάτι που ήταν αναμενόμενο δεδομένης της ενίσχυσης εγγραφής.
- Στις αναγνώσεις οι τιμές ήταν αντίστοιχες, ξεκινώντας από τα 5 MB/sec με ένα νήμα και φτάνοντας μέχρι τα 26 MB/sec.
- Έπειτα κάποια νήματα εκτελούσαν εγγραφές και κάποια αναγνώσεις ταυτόχρονα, όπου παρατηρήθηκε απόδοση από 3 έως 9 MB/sec το καθένα.
- Τέλος αφήσαμε να ολοκληρωθούν οι συμπυκνώσεις, και εκτελέσαμε τυχαίες εγγραφές για 1 δευτερόλεπτο, ώστε να μην προλάβουν να χρειαστούν άλλες συμπυκνώσεις, όπου όπως αναμενόταν, πραγματοποιήθηκαν με την ταχύτητα του δίσκου (65 MB/sec).

[4]Για να γεμίσει μια βάση με ήδη ταξινομημένα δεδομένα γρηγορότερα, το RocksDB υποστηρίζει την προσωρινή απενεργοποίηση του WAL, ώστε η ενίσχυση εγγραφής να είναι 1 και να αξιοποιούνται όλες οι δυνατότητες του δίσκου.

Στη συνέχεια δοκιμάσαμε και με έναν δεύτερο κόμβο στην εικονική μηχανή του πελάτη. Αυτό κανονικά δε θα έδινε ακριβείς μετρήσεις καθώς χάνεται η καθυστέρηση του δικτύου, αλλά δεδομένου ότι σημείο συμφόρησης ήταν ο δίσκος, η συμπεριφορά δεν επηρεάστηκε ιδιαίτερα. Το μέγεθος της βάσης διπλασιάστηκε, ώστε να διατηρεί ο κάθε κόμβος 2 GB, και παρατηρήσαμε την απόδοση σε όλα τα benchmarks, να διπλασιάζεται.

Στη συνέχεια πραγματοποιήθηκαν μετρήσεις με καλύτερα μηχανήματα και μεγαλύτερα μεγέθη συστοιχίας. Χρησιμοποιήθηκαν δύο είδη εικονικών μηχανών, ένα για τους κόμβους και ένα για τους πελάτες με τα εξής χαρακτηριστικά.

	CPUs	RAM	Εύρος ζώνης	Δίσκοι
Διακομιστές	4 (2.5 GHz)	15 GB	1 Gbps	2×40 GB SSDs
Πελάτες	8	16 GB	10 Gbps	

Πίνακας 5.1: *Χαρακτηριστικά εικονικών μηχανών*

Οι δίσκοι των εικονικών μηχανών των κόμβων, είχαν μέση ταχύτητα εγγραφών περίπου 200 MB/sec και ρυθμίστηκαν σε RAID 0, επομένως συμπεριφέρονταν σαν ένας δίσκος χωρητικότητας 80 GB και συνολικής ταχύτητας εγγραφών 400 MB/sec. Η μέση καθυστέρησή τους σε εγγραφές των 4 KB, μετρήθηκε με τη χρήση του εργαλείου `ioping`[5] στα 275 μsec, ενώ η μέση καθυστέρηση του δικτύου μετρήθηκε στα 460 μsec.

Επειδή χρειάστηκε να παρθούν πολλές μετρήσεις και δεν υπήρχε αρκετός χρόνος για να γεμίζει ο δίσκος πριν από κάθε μια, αρκεστήκαμε σε 20 GB ανά κόμβο. Ο κάθε κόμβος είχε 20 shards, δηλαδή 20 column families, άρα 1 GB ανά column family. Η επιλογή αυτή έγινε επειδή η διαθέσιμη υπολογιστική ισχύς ήταν πολύ μεγάλη και το σύστημα μπορούσε να αντεπεξέλθει. Επομένως το LSM-δέντρο είχε την ακόλουθη μορφή.

Επίπεδο	Μέγεθος	Αρχεία
0	256 MB	4
1	128 MB	2
2	1 GB	10

[5] `https://github.com/koct9i/ioping`

Για την ενίσχυση εγγραφής έχουμε:

- 1 για το WAL.
- 1 για το επίπεδο 0.
- Για τη συμπύκνωση των επιπέδων 0 και 1 στο επίπεδο 1, αναμένουμε 4 + 2 αρχεία να παράγουν περίπου 5, άρα η ενίσχυση είναι κάτι παραπάνω από 1.
- Από τα 5 αρχεία του επιπέδου 1, κάποια θα μεταβούν στο επίπεδο 2. Το καθένα θα έχει επικάλυψη με 2 αρχεία και η συμπύκνωση θα παράγει 2 νέα. Άρα η ενίσχυση θα είναι στο 2.

Συνολικά η ενίσχυση φαίνεται να είναι κοντά στο 5. Όμως επειδή κάποιες τιμές ανανεώνονταν πριν προλάβουν να γραφτούν ξανά, και λόγω της συμπίεσης, μετρήσαμε ενίσχυση εγγραφής περίπου 4. Επομένως η μέγιστη ταχύτητα σε συνεχόμενες εγγραφές ήταν περίπου 400 / 4 = 100 MB/sec, και ο δίσκος δεν αποτελούσε μεγαλύτερο σημείο συμφόρησης από ότι το δίκτυο.

Ο πελάτης όπως και προηγουμένως, ήταν ένας, και χρησιμοποιούσε μεταβλητό πλήθος νημάτων, το καθένα από τα οποία εκτελούσε διαδοχικά αιτήματα για ένα συγκεκριμένο χρονικό διάστημα. Το μέγεθος της συστοιχίας ήταν επίσης μεταβλητό και κυμαινόταν από 1 έως 16 κόμβους. Χρησιμοποιήθηκαν κλειδιά των 16 byte, και 3 διαφορετικά μεγέθη τιμών — 128 bytes, 4 KB και 256 KB.

5.2.1 Καθυστέρηση

Η καθυστέρηση του κάθε ξεχωριστού αιτήματος φάνηκε να είναι ανεξάρτητη του μεγέθους της συστοιχίας. Παραθέτουμε ενδεικτικά την καθυστέρηση που μετρήθηκε στις τυχαίες εγγραφές. Με τον όρο "P_n" εννοούμε ότι το n% των αιτημάτων είχαν μικρότερη καθυστέρηση από την αναγραφόμενη. [6]

[6] `https://en.wikipedia.org/wiki/Percentile`

min	440
P_{50}	500
P_{90}	550
P_{95}	600
P_{99}	750
P_{999}	3000
P_{9999}	25000
P_{99999}	100000
max	200000
mean	530

Πίνακας 5.2: *Καθυστερήσεις στις τιμές των 128 bytes (μsec)*

min	480
P_{50}	550
P_{90}	610
P_{95}	670
P_{99}	850
P_{999}	3700
P_{9999}	40000
P_{99999}	140000
max	200000
mean	580

Πίνακας 5.3: *Καθυστερήσεις στις τιμές των 4 KB (μsec)*

min	1500
P_{50}	2000
P_{90}	2200
P_{95}	3000
P_{99}	6750
P_{999}	23000
P_{9999}	250000
max	300000
mean	2200

Πίνακας 5.4: *Καθυστερήσεις στις τιμές των 256 KB (μsec)*

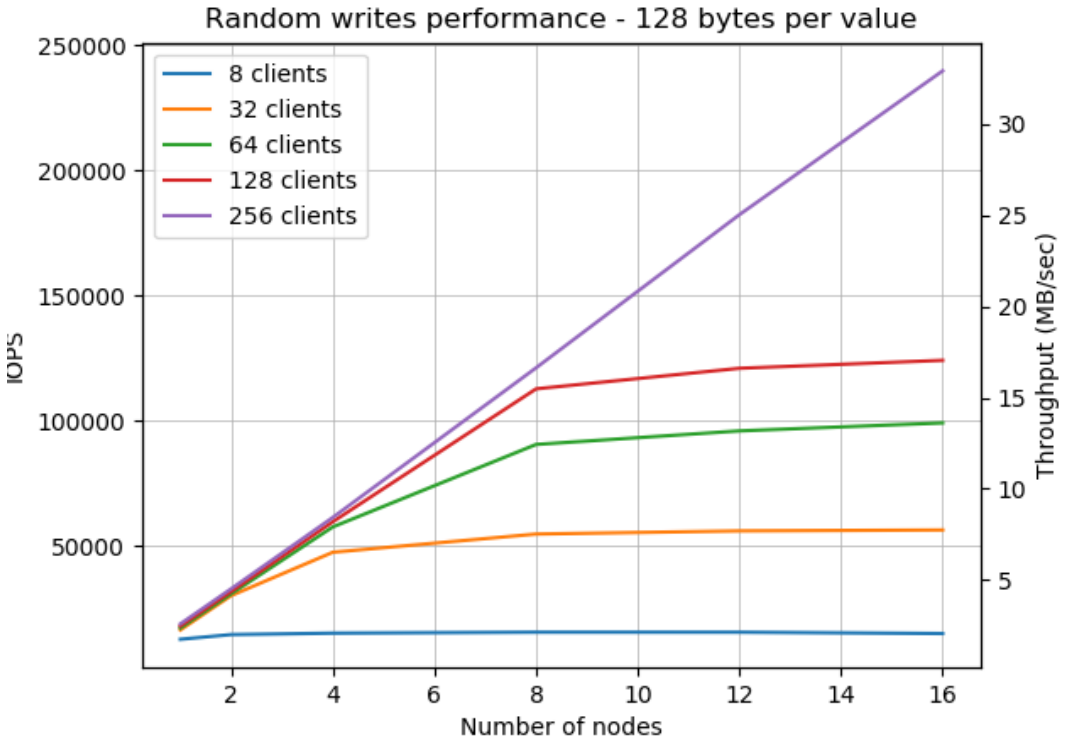
Παρατηρούμε ότι η συνήθης καθυστέρηση ήταν πολύ χαμηλή — χαμηλότερη από το άθροισμα των καθυστερήσεων του δικτύου και του του δίσκου, τουλάχιστον για μικρές τιμές. Υπενθυμίζουμε ότι το RocksDB δεν έχει ρυθμιστεί να συγχρονίζεται με το δίσκο σε κάθε νέα εγγραφή, αλλά μόνο να μεταφέρει το αίτημα στο λειτουργικό σύστημα. Η μέγιστη καθυστέρηση όμως, είναι μεγαλύτερη κατά 2 με 3 τάξεις μεγέθους. Ο λόγος είναι ότι το RocksDB χρησιμοποιεί ορισμένα κλειδώματα, που αν και περιορισμένα, ορισμένες φορές καθυστερεί να τα πάρει, με αποτέλεσμα να αδυνατεί να εξυπηρετήσει έγκαιρα το αίτημα.

Ένας άλλος λόγος θα μπορούσε να είναι ότι φτάνουν SSTables στο επίπεδο 0, με τέτοιο ρυθμό που η ταχύτητα του δίσκου δεν επαρκεί για να προλάβουν να μεταβούν σε επόμενα επίπεδα. Το RocksDB αντιμετωπίζει αυτές τις καταστάσεις με τον εξής τρόπο: Ας υποθέσουμε ότι έχει ρυθμιστεί να ξεκινάει τις συμπυκνώσεις όταν μαζευτούν 4 αρχεία στο επίπεδο 0, και ότι υπάρχουν δύο ακόμη κατώφλια στα 8 και 12 αρχεία αντίστοιχα. Αν τα αρχεία φτάσουν τα 8, αρχίζει να μπαίνει ένα όριο στο πόσο γρήγορα γίνονται οι εισαγωγές, δηλαδή προστίθεται μια τεχνητή καθυστέρηση. Αν παρ' όλα αυτά εξακολουθήσουν να αυξάνονται τα αρχεία, στα 12 σταματάνε τελείως οι εγγραφές και αρχίζουν ξανά όταν λιγοστέψουν. Σε αυτήν την περίπτωση η καθυστέρηση θα ήταν ακόμη πιο αυξημένη, όμως έχουμε ρυθμίσει την τεχνητή καθυστέρηση κατάλληλα, ώστε να μην ξεπερνιέται ποτέ το μεγάλο κατώφλι.

5.2.2 IOPS

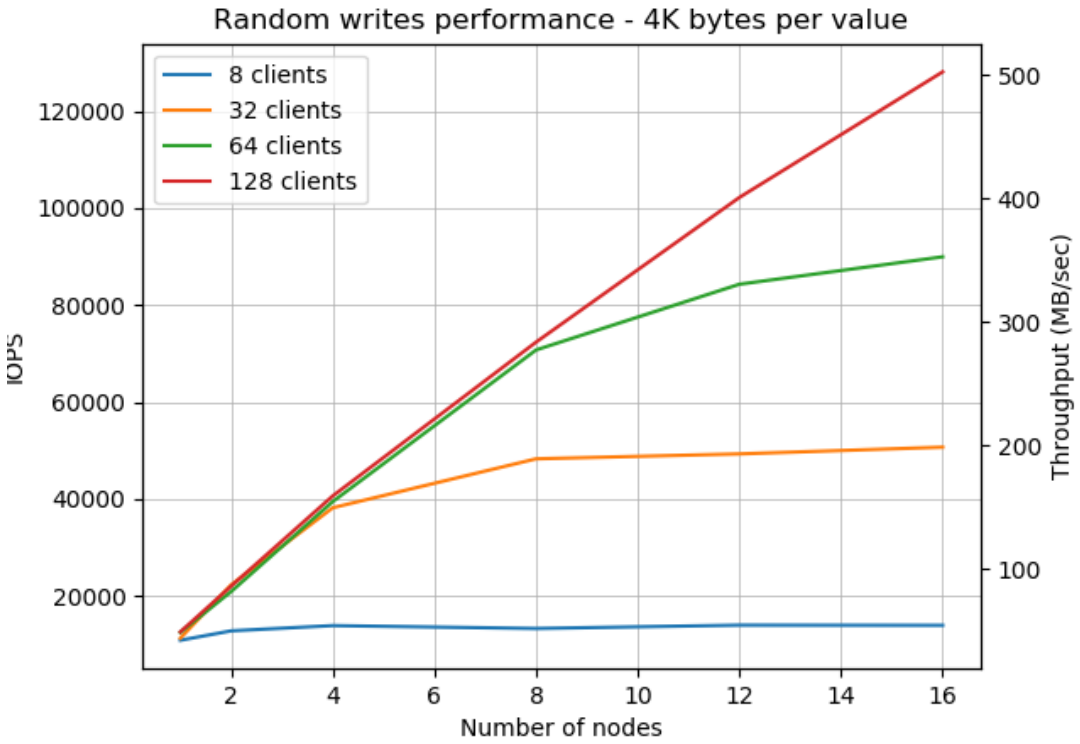
Στις ακόλουθες γραφικές παραστάσεις βλέπουμε στον αριστερό άξονα των y το πλήθος των αιτημάτων που ολοκληρώνονται ανά δευτερόλεπτο, και στον δεξιό την ταχύτητα σε MB/sec στην οποία αντιστοιχούν αυτά τα αιτήματα, ενώ η κάθε καμπύλη αντιστοιχεί στο εκάστοτε πλήθος νημάτων του πελάτη.

Εγγραφές

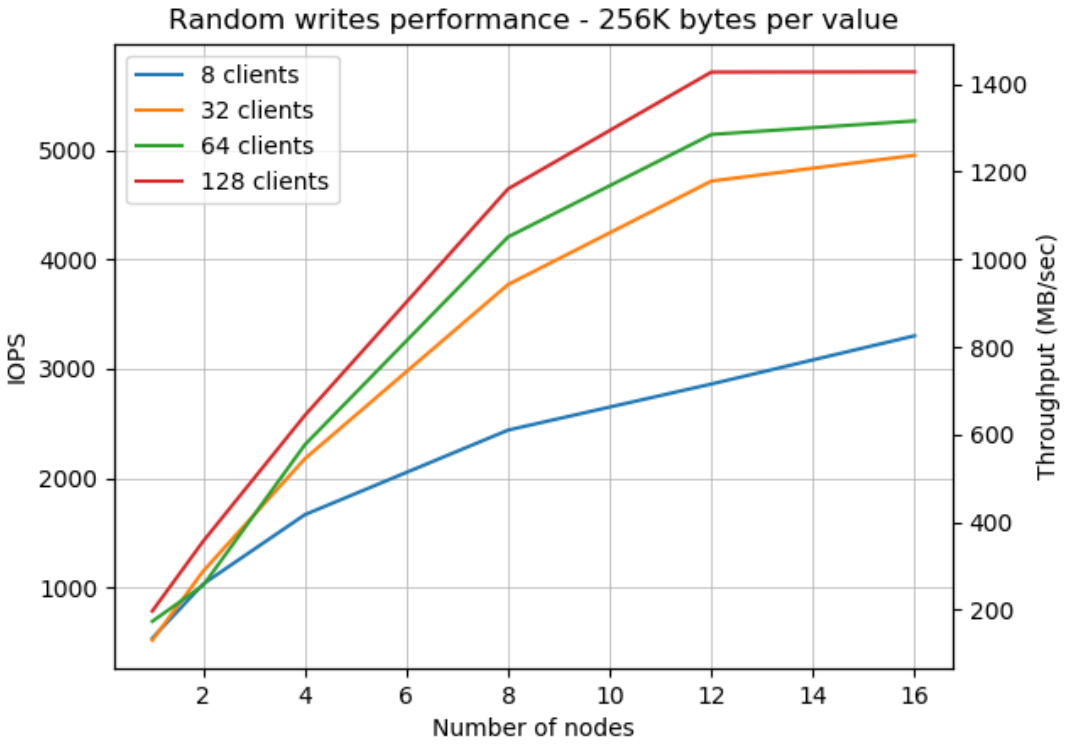


Εάν δεν υπήρχε καθόλου καθυστέρηση από το RocksDB, με καθυστέρηση 460 μsec στο δίκτυο, ένας πελάτης θα μπορούσε να εκτελεί $1 \cdot 10^6/460 \simeq 2174$ διαδοχικά αιτήματα ανά δευτερόλεπτο. Στην περίπτωση του ενός νήματος που εκτελεί αιτήματα των 128 bytes, μετρήσαμε 2030 ops/sec που δεν απέχει πολύ από το ιδανικό.

Αντίστοιχα με 256 πελάτες, το μέγιστο είναι περίπου $256 \cdot 2174 \simeq 550000$ αιτήματα ανά δευτερόλεπτο και παρατηρήσαμε 240000, μία αρκετά ικανοποιητική τιμή δεδομένου ότι κάποια αιτήματα είχαν αυξημένη καθυστέρηση (της τάξης των 0.2 δευτερολέπτων όπως είδαμε παραπάνω).



Στην περίπτωση των 4 KB, η συμπεριφορά είναι η ίδια. Λόγω της αυξημένης κίνησης στο δίκτυο, αναπόφευκτα η καθυστέρηση ανέβηκε, με αποτέλεσμα τα αιτήματα ανά δευτερόλεπτο να πέσουν περίπου στο μισό, αλλά η κλιμάκωση παρέμεινε γραμμική με τη συνολική ταχύτητα να φτάνει τα 500 MB/sec στη συστοιχία των 16 κόμβων.

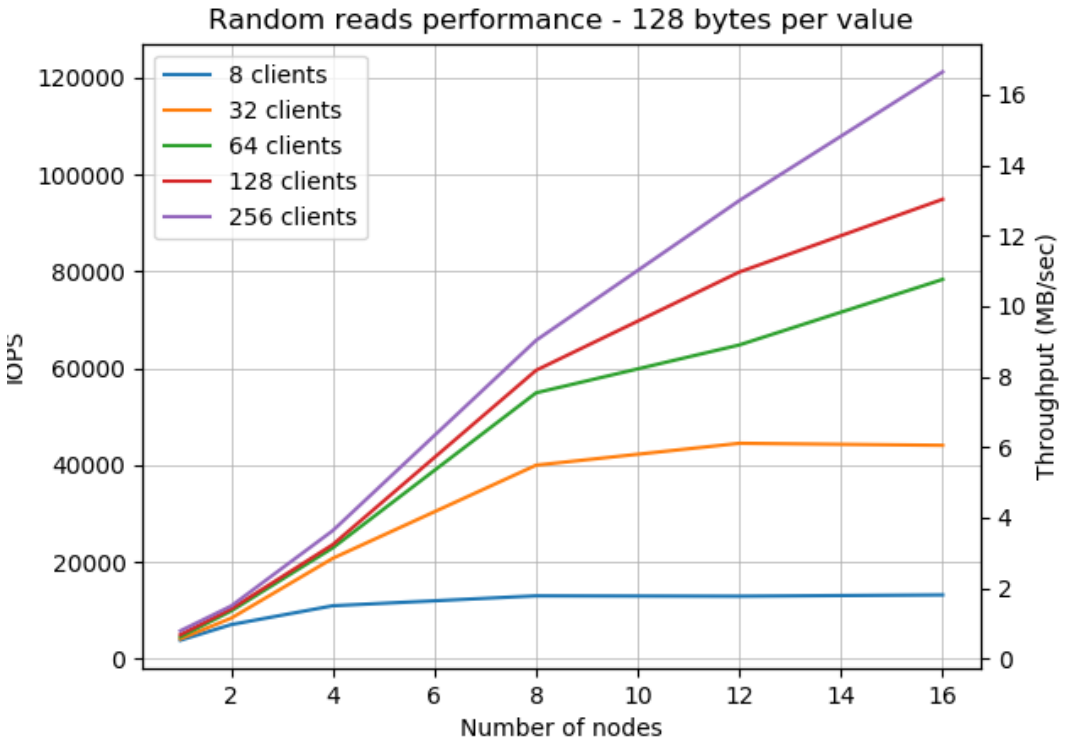


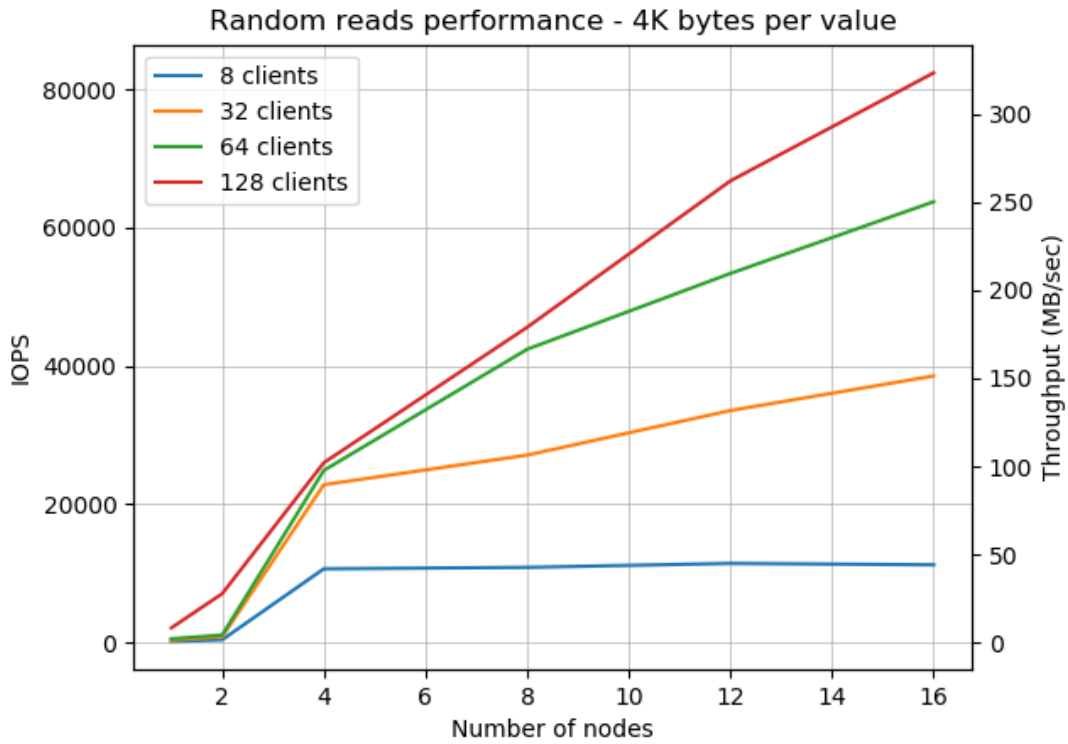
Τέλος στην περίπτωση με τις τιμές των 256 KB, φαίνεται ότι σημείο συμφόρησης ήταν το εύρος ζώνης του δικτύου. Δηλαδή από 1 έως 10 κόμβους είχαμε ταχύτητα ίση με το εύρος ζώνης των κόμβων επί το πλήθος τους, και από εκεί και έπειτα άρχισε να αποτελεί σημείο συμφόρησης το εύρος ζώνης του πελάτη, το οποίο ήταν δεκαπλάσιο (10 έναντι 1 Gbps).

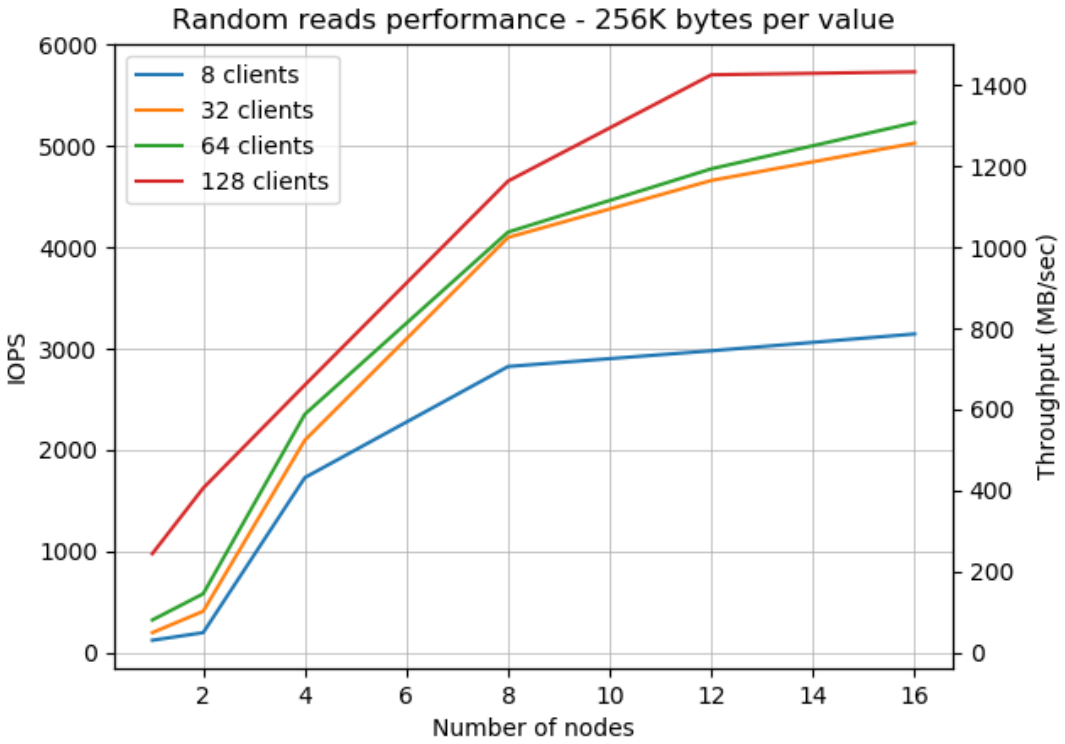
Το σημαντικό είναι ότι σε κάθε περίπτωση, η απόδοση αυξάνεται γραμμικά με την αύξηση του μεγέθους της συστοιχίας.

Αναγνώσεις

Ακολουθούν οι γραφικές παραστάσεις για τα benchmarks των τυχαίων αναγνώσεων.


Random reads performance - 128 bytes per value
8 clients
32 clients
64 clients
128 clients
256 clients
IOPS
Throughput (MB/sec)
Number of nodes
0
20000
40000
60000
80000
100000
120000
0
2
4
6
8
10
12
14
16
2
4
6
8
10
12
14
16


Random reads performance - 4K bytes per value
8 clients
32 clients
64 clients
128 clients
IOPS
Throughput (MB/sec)
Number of nodes
0
20000
40000
60000
80000
0
50
100
150
200
250
300
2
4
6
8
10
12
14
16

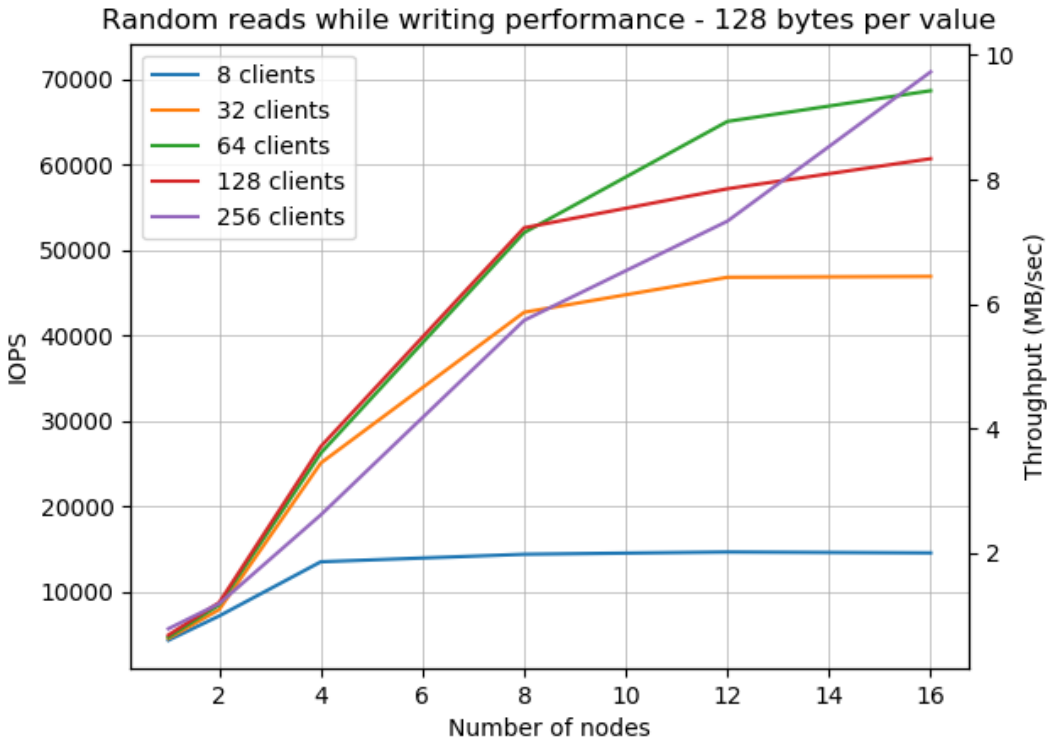


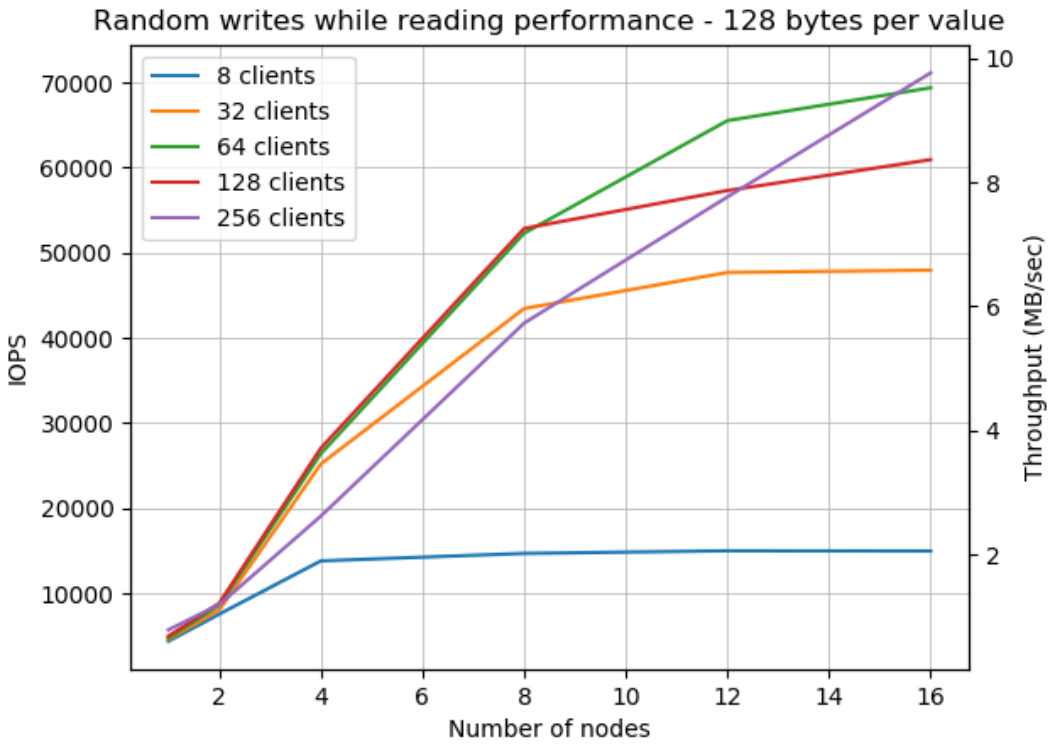
Παρατηρούμε ότι στις αναγνώσεις η συμπεριφορά της συστοιχίας είναι ακριβώς η ίδια με αυτή των εγγραφών, απλά όπως ήταν αναμενόμενο η απόδοση είναι κάπως χαμηλότερη.

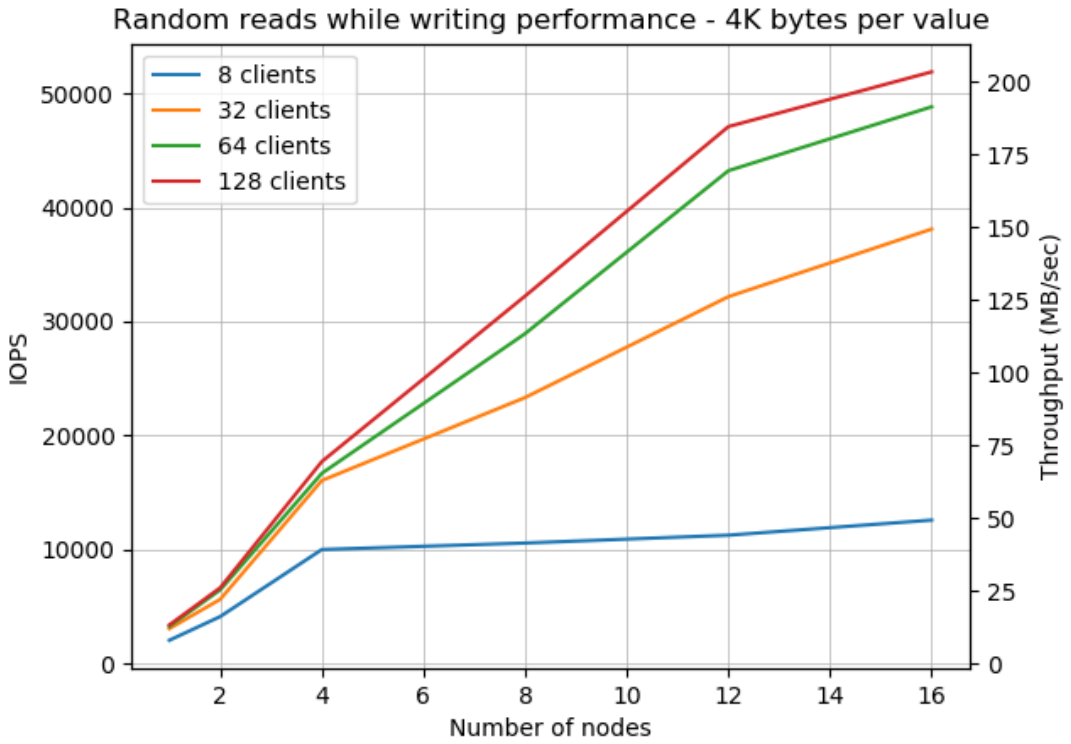
Ταυτόχρονες εγγραφές και αναγνώσεις

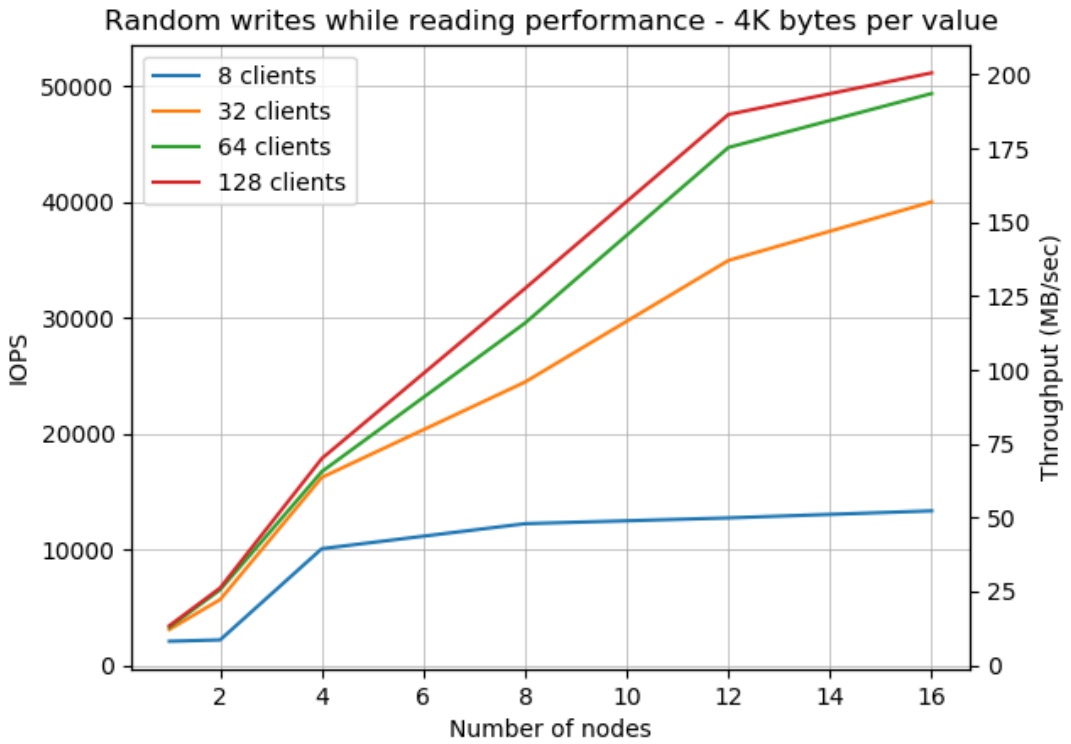
Ακολουθούν οι γραφικές παραστάσεις για τις ταυτόχρονες εγγραφές και αναγνώσεις.

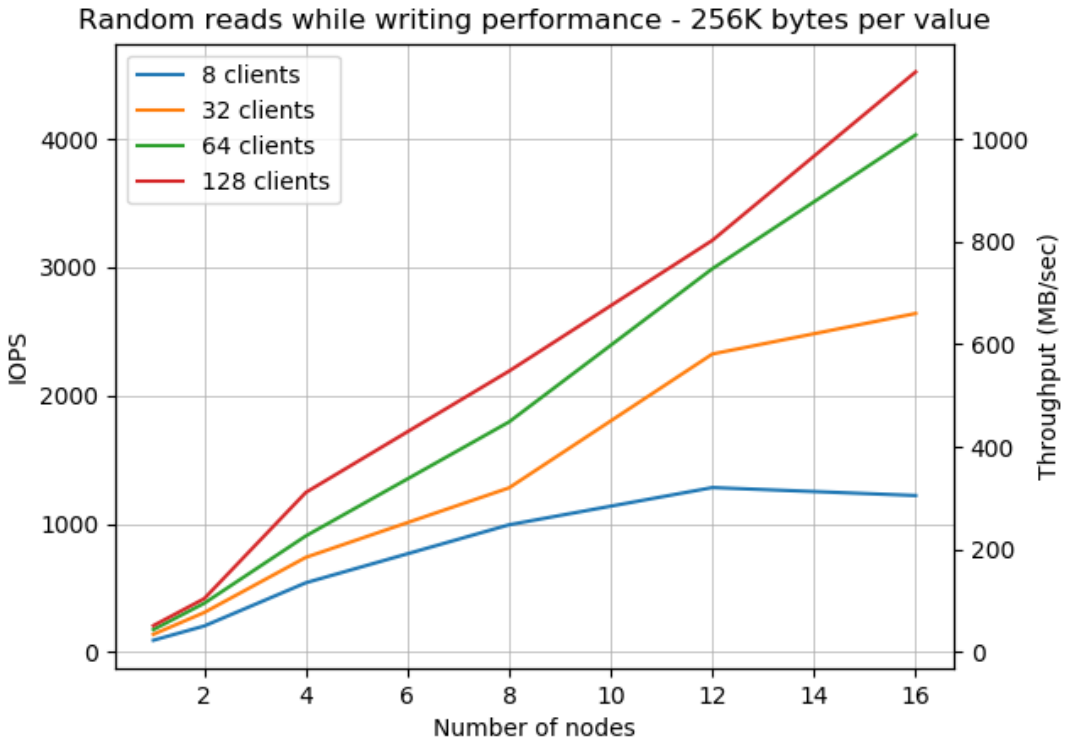
Σε όλες τις μετρήσεις, ο αριθμός των νημάτων που πραγματοποιούσαν εγγραφές ήταν ο ίδιος με αυτών που πραγματοποιούσαν αναγνώσεις. Αν και θα είχε ενδιαφέρον να βλέπαμε τη συμπεριφορά με διαφορετικό λόγο νημάτων, κάτι τέτοιο δεν ήταν εφικτό λόγω έλλειψης χρόνου.

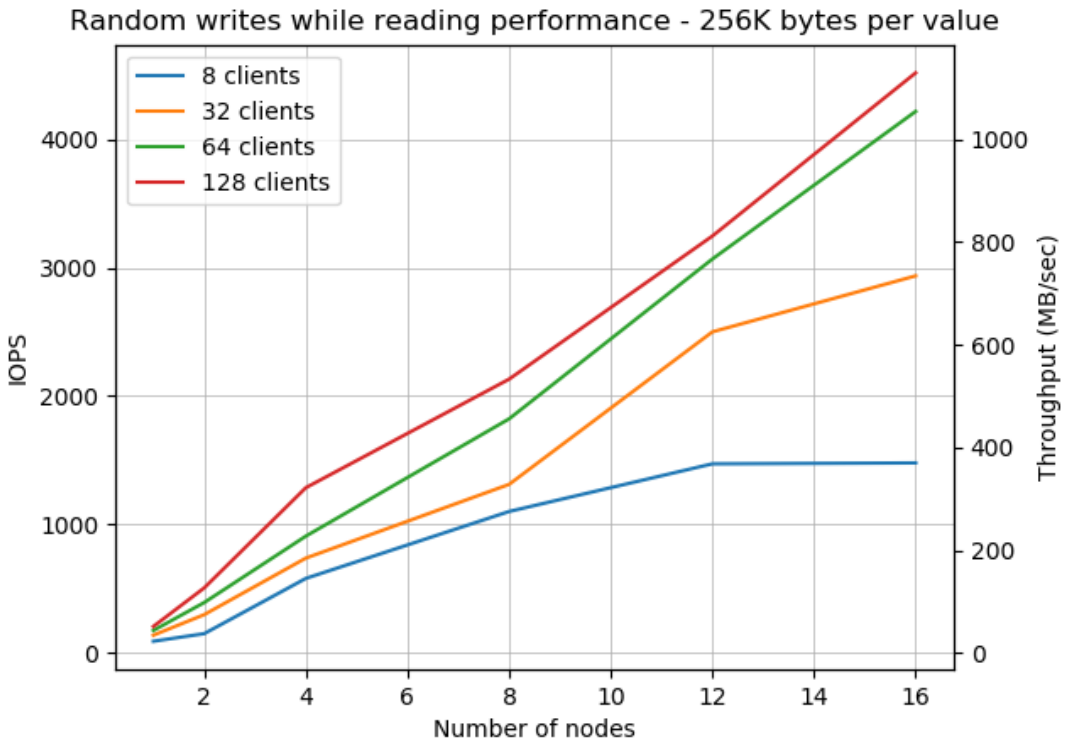

Random reads while writing performance - 128 bytes per value
8 clients
32 clients
64 clients
128 clients
256 clients
IOPS
Throughput (MB/sec)
Number of nodes
70000
60000
50000
40000
30000
20000
10000
10
8
6
4
2
2
4
6
8
10
12
14
16


Random writes while reading performance - 128 bytes per value
8 clients
32 clients
64 clients
128 clients
256 clients
IOPS
Throughput (MB/sec)
Number of nodes
70000
60000
50000
40000
30000
20000
10000
10
8
6
4
2
2
4
6
8
10
12
14
16


Random reads while writing performance - 4K bytes per value
8 clients
32 clients
64 clients
128 clients
IOPS
Throughput (MB/sec)
Number of nodes
0
10000
20000
30000
40000
50000
0
25
50
75
100
125
150
175
200
2
4
6
8
10
12
14
16


Random writes while reading performance - 4K bytes per value
8 clients
32 clients
64 clients
128 clients
IOPS
Throughput (MB/sec)
Number of nodes
0
10000
20000
30000
40000
50000
0
25
50
75
100
125
150
175
200
2
4
6
8
10
12
14
16



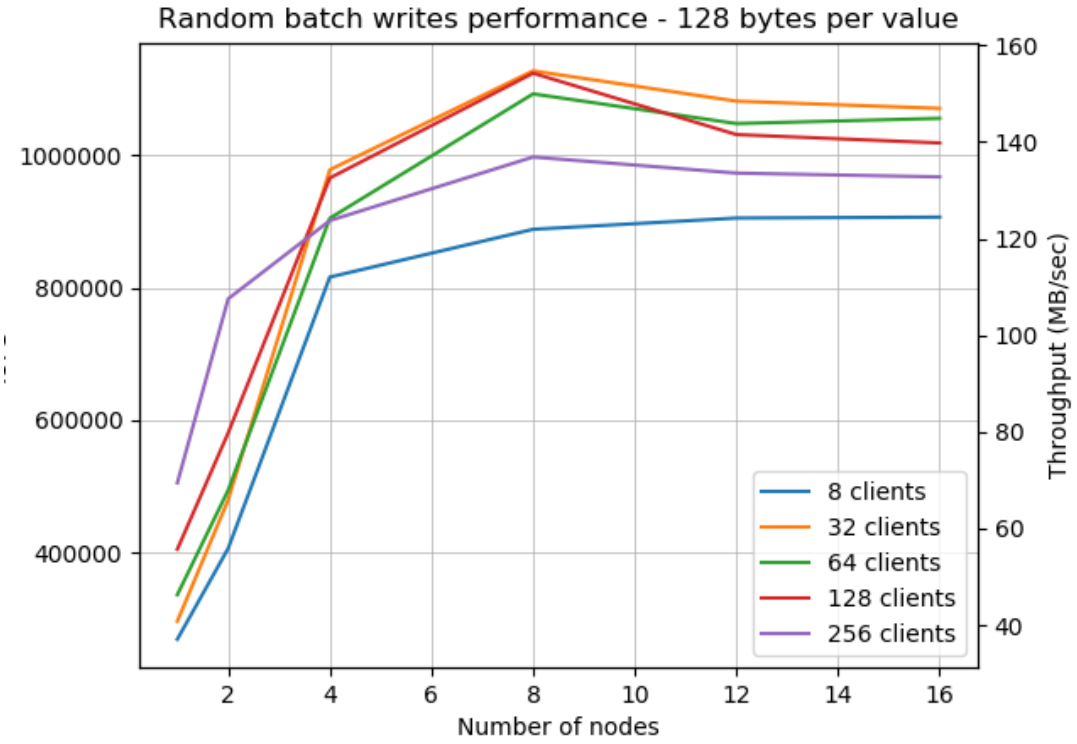


Παρατηρούμε ότι τόσο οι εγγραφές όσο και οι αναγνώσεις πραγματοποιήθηκαν με

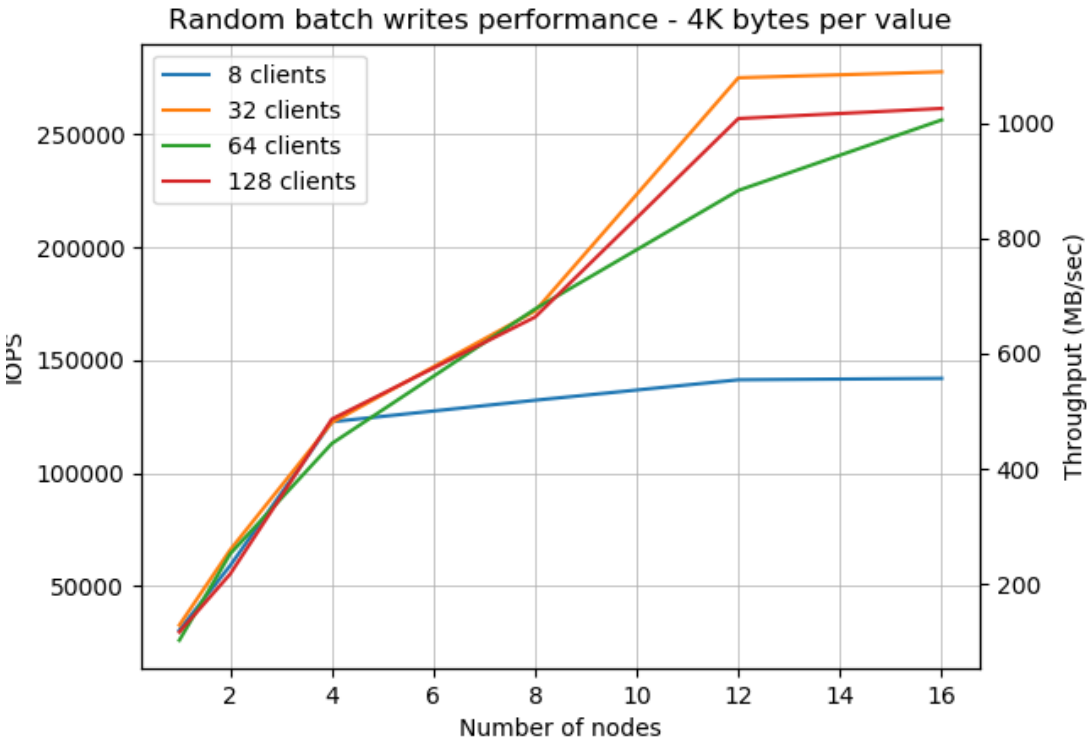
την ίδια περίπου ταχύτητα. Στην περίπτωση των 128 bytes και 4 KB, η ταχύτητα ήταν χαμηλότερη από αυτή των μεμονωμένων εγγραφών ή αναγνώσεων, ενώ στην περίπτωση των 256 KB, και οι δύο κατάφεραν να αξιοποιήσουν στο μέγιστο την ταχύτητα του δικτύου.

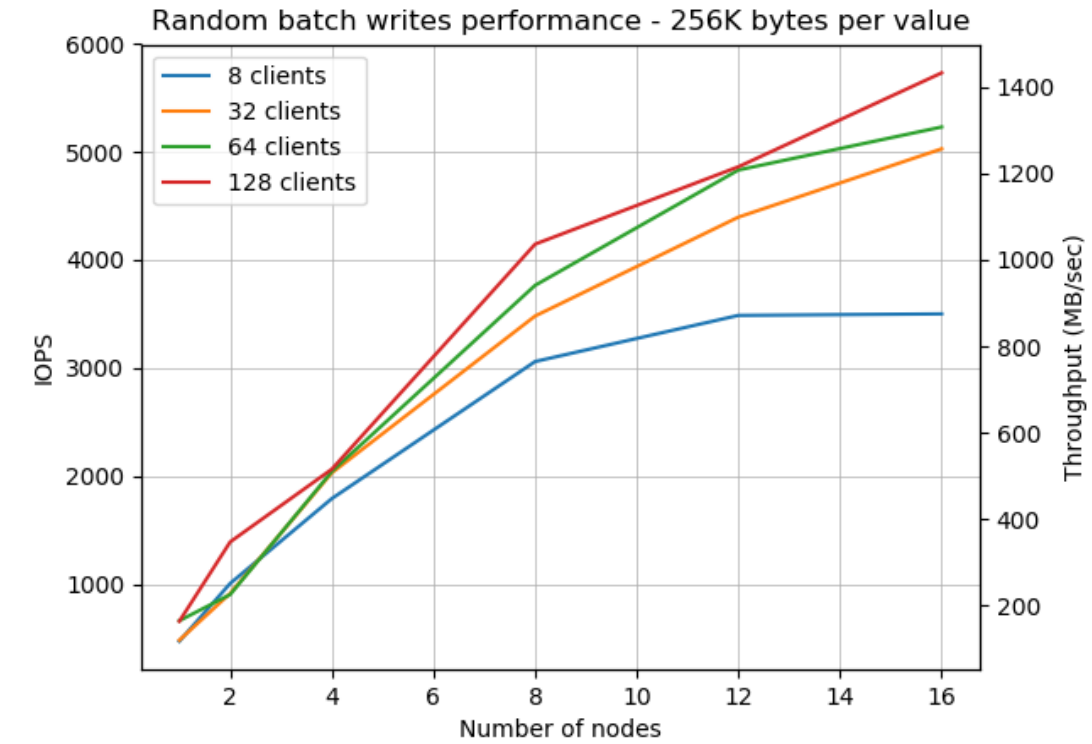
Εγγραφές με τη χρήση write batch

Τέλος βλέπουμε τα benchmarks των μαζικών εγγραφών.



Στην περίπτωση των μαζικών εγγραφών των 128 bytes, φαίνεται ότι σημείο συμφόρησης αποτελεί ο πελάτης, ο οποίος αδυνατεί να παράγει περισσότερες από 1100000 περίπου τιμές το δευτερόλεπτο. Η τιμή αυτή επιτυγχάνεται στους 8 κόμβους, και από εκεί και έπειτα η απόδοση παραμένει σταθερή.





Στις υπόλοιπες περιπτώσεις, σημείο συμφόρησης αποτελεί το δίκτυο και οι εγγραφές

ξεπερνάνε το 1 GB/sec, τόσο στις τιμές των 4 KB όσο και σε αυτές των 256 KB.

6

Συμπεράσματα - μελλοντικές επεκτάσεις

Στα πλαίσια της διπλωματικής αυτής εργασίας, υλοποιήσαμε ένα κατανεμημένο σύστημα αποθήκευσης κλειδιού-τιμής, που παρουσιάζει πολύ καλή απόδοση γραμμικά κλιμακώσιμη με την αύξηση των κόμβων. Η απόδοση εξαρτάται από τα χαρακτηριστικά των κόμβων, του δικτύου καθώς και του χώρου που καταλαμβάνει η βάση, και είναι συγκεκριμένη και εύκολα υπολογίσιμη εκ των προτέρων.

Εκτελώντας διάφορα πειράματα, παρατηρήσαμε ότι η καθυστέρηση στα περισσότερα αιτήματα ήταν εξαιρετικά χαμηλή, και καταφέραμε να πραγματοποιήσουμε τυχαίες εγγραφές με ταχύτητα 1.4 GB ανά δευτερόλεπτο (όσο δηλαδή επέτρεπε το δίκτυο), χρησιμοποιώντας μια συστοιχία μόλις 10 κόμβων.

Παρ' όλα αυτά, το crocks δεν μπορεί να θεωρηθεί ολοκληρωμένο, και έχει αρκετά περιθώρια βελτίωσης. Αναφέρουμε ενδεικτικά κάποιες από τις σημαντικότερες βελτιώσεις που μπορούν να πραγματοποιηθούν στο μέλλον:

Υπερεπάρκεια (redundancy) και αυτόματη μετάπτωση (failover)

Στην τρέχουσα υλοποίηση, η αποτυχία ενός κόμβου ενώ δεν οδηγεί σε απώλεια δεδομένων, καθιστά το σύστημα μη αποκρίσιμο μέχρι ο διαχειριστής να επαναφέρει τον κόμβο. Για να ξεπεραστεί αυτό το πρόβλημα θα πρέπει να υλοποιηθεί ένας μηχανισμός αντιγραφής (replication), έτσι ώστε το κάθε shard να βρίσκεται ανά πάσα στιγμή σε περισσότερους από έναν κόμβους, και σε περίπτωση αποτυχίας, οι πελάτες να παρα-

πέμπονται αυτόματα σε κάποιον άλλο.

Πρόβλεψη για περαιτέρω κλιμακωσιμότητα

Η εφαρμογή μας αυτή τη στιγμή έχει το πρόβλημα ότι το πλήθος των κόμβων μιας συστοιχίας, δεν μπορεί να ξεπεράσει το πλήθος των αρχικά επιλεγμένων shards. Αυτό είναι μια σχεδιαστική απόφαση που έχει παρθεί με στόχο την βέλτιστη απόδοση του συστήματος.

Παρ᾽ όλα αυτά, αυτή η αδυναμία θα μπορούσε να ξεπεραστεί, με ανακατανομή των κλειδιών σε νέα, περισσότερα shards. Αυτό θα απαιτούσε την μεταφορά όλων σχεδόν των κλειδιών σε νέους κόμβους, και ενδεχομένως η απόδοση κατά τη μετανάστευση να ήταν τόσο χαμηλή που να μην είχε νόημα να πραγματοποιείται ζωντανά. Μια τέτοια δυνατότητα όμως, είναι είναι αναγκαία, καθώς συχνά είναι αδύνατο να προβλεφθούν οι μελλοντικές ανάγκες μιας εφαρμογής, και δεν πρέπει να υπάρχει δέσμευση από τις αρχικές επιλογές.

Υλοποίηση άλλων λειτουργιών του RocksDB

Το RocksDB όπως έχουμε αναφέρει, περιλαμβάνει μεγάλο πλήθος λειτουργιών, πολλές από τις οποίες θα ήταν χρήσιμο να υλοποιηθούν και από τη δική μας εφαρμογή. Ίσως η χρησιμότερη, από αυτές να είναι η υλοποίηση των συναλλαγών (transactions), καθώς θα μπορέσουν να εξασφαλίσουν την ορθή λειτουργία των write batches, διατηρώντας παράλληλα την επίδοσή τους.

Οι συναλλαγές στο RocksDB [1] υποστηρίζουν το πρωτόκολλο δέσμευσης δύο φάσεων [15] (Two-phase commit — 2PC) κάτι που μας επιτρέπει να τις υλοποιήσουμε και στο δικό μας κατανεμημένο σύστημα. Η δέσμευση δύο φάσεων λειτουργεί ως εξής:

Κάθε ανάγνωση ή εγγραφή που πραγματοποιείται στα πλαίσια της συναλλαγής, παίρνει ένα κλείδωμα στο αντίστοιχο κλειδί. Από εκείνη τη στιγμή, μέχρι να ολοκληρωθεί η διαδικασία της δέσμευσης, δεν επιτρέπεται να πραγματοποιηθεί άλλη εγγραφή πάνω σε αυτό το κλειδί. Έτσι εξασφαλίζεται το επίπεδο απομόνωσης επαναλαμβανόμενων αναγνώσεων (repeatable reads).

[1] `https://github.com/facebook/rocksdb/wiki/Transactions`

Επίσης υπάρχει η δυνατότητα να εξασφαλιστεί απομόνωση στιγμιότυπου, όπου δημιουργείται ένα στιγμιότυπο της βάσης κατά την έναρξη της συναλλαγής και σε κάθε ανάγνωση ή εγγραφή ελέγχεται ότι δεν έχει τροποποιηθεί το αντίστοιχο κλειδί από τη στιγμή της δημιουργίας.

Πριν τη δέσμευση της συναλλαγής, πραγματοποιείται η διαδικασία της προετοιμασίας (prepare), όπου τοποθετούνται στο WAL όλες οι εγγραφές της συναλλαγής, αλλά όχι και στο MemTable, με αποτέλεσμα να μην είναι ακόμα ορατές από έναν τρίτο που έχει πρόσβαση στη βάση.

Όταν η προετοιμασία ολοκληρωθεί επιτυχώς, υπάρχει η επιλογή είτε να πραγματοποιηθεί η δέσμευση, όπου οι εγγραφές περνάνε στο MemTable, είτε να γίνει επαναφορά (rollback) όπου η συναλλαγή ακυρώνεται. Και στις δύο περιπτώσεις τα κλειδώματα αποδεσμεύονται και προστίθεται η αντίστοιχη εγγραφή στο WAL.

Εάν συμβεί κάποια αποτυχία μετά το στάδιο της προετοιμασίας, η δέσμευση μπορεί να πραγματοποιηθεί κανονικά μετά την επαναφορά του συστήματος.

Επομένως θα υλοποιούσαμε τις συναλλαγές στο δικό μας σύστημα ως εξής: Ο πελάτης ξεκινάει μια συναλλαγή και αρχίζει να στέλνει εγγραφές στους κόμβους. Εάν αποτύχει κάποια εγγραφή σε έναν οποιονδήποτε κόμβο (π.χ. επειδή κάποια άλλη συναλλαγή έχει πάρει κλείδωμα στο αντίστοιχο κλειδί), ο κόμβος ακυρώνει τη συναλλαγή, επιστρέφει το αντίστοιχο status, και ο πελάτης ζητάει την ακύρωση και από τους υπόλοιπους κόμβους. Εάν πετύχουν όλες οι εγγραφές, ο πελάτης ζητάει να γίνει η διαδικασία της προετοιμασίας. Αν όλοι οι κόμβοι ανακοινώσουν ότι η προετοιμασία ολοκληρώθηκε επιτυχώς, ο πελάτης ζητάει να πραγματοποιηθεί η δέσμευση, ενώ αν αποτύχει έστω και μία (π.χ. επειδή η τιμή κάποιου κλειδιού έχει τροποποιηθεί από τη στιγμή της δημιουργίας του στιγμιότυπου), ζητάει την επαναφορά.

Βιβλιογραφία

[1] Facebook. RocksDB. `http://rocksdb.org/`, 2015.

[2] Patrick O'Neil and Edward Cheng and Dieter Gawlick and Elizabeth O'Neil. *The Log-Structured Merge-Tree (LSM-Tree).* Acta Inf., 33(4):351–385, 1996.

[3] Google. LevelDB. `https://github.com/google/leveldb`, 2014.

[4] William Pugh. *Skip Lists: A Probabilistic Alternative to Balanced Trees.* 1990.

[5] Keir Fraser and Tim Harris. *Concurrent Programming Without Locks.* 2004.

[6] Hyeontaek Lim, David G. Andersen, Michael Kaminsky. *Towards Accurate and Fast Evaluation of Multi-Stage Log-Structured Designs.* Carnegie Mellon University, Intel Labs, 2016.

[7] R Bayer, E McCreight. *Organization and maintenance of large ordered indexes.* 1972.

[8] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition.* 2007.

[9] Burton H. Bloom. *Space/Time Trade-offs in Hash Coding with Allowable Errors*, Communications of the ACM, 1970.

[10] Li Fan and Pei Cao and Jussara Almeida and Andrei Z. Broder. *Summary cache: A scalable wide-area web cache sharing protocol,* 1998.

[11] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, Michael Stumm. *Optimizing Space Amplification in RocksDB.* 2017.

[12] Diego Ongaro and John Ousterhout. *In Search of an Understandable Consensus Algorithm.* 2014.

[13] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms.* 2002.

[14] Abraham Silberschatz, Henry F. Korth, S. Sudarshan *Database System Concepts, 6th Edition.* 2010.

[15] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* 1987.