ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Βελτιστοποίηση Επίδοσης Εγγραφών στο Κατανεμημένο Σύστημα Αποθήκευσης Κλειδιού-Τιμής etcd μέσω Ενσωμάτωσης του Αποθηκευτικού Μηχανισμού RocksDB

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεωργία Μ. Κοκκίνου

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Μάρτιος 2018

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

Βελτιστοποίηση Επίδοσης Εγγραφών στο Κατανεμημένο Σύστημα Αποθήκευσης Κλειδιού-Τιμής etcd μέσω Ενσωμάτωσης του Αποθηκευτικού Μηχανισμού RocksDB

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεωργία Μ. Κοκκίνου

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Μαρτίου 2018.

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

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Μάρτιος 2018

National Technical University of Athens

School of Electrical and Computer Engineering

Division of Computer Science

Optimizing Write Performance in the etcd Distributed Key-Value Store via Integration of the RocksDB Storage Engine

DIPLOMA THESIS

Georgia M. Kokkinou

Computing Systems Laboratory

Athens, March 2018

. . . . . .

Γεωργία Μ. Κοκκίνου
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Ο *etcd* είναι ένα αξιόπιστο κατανεμημένο σύστημα αποθήκευσης κλειδιού-τιμής ανοιχτού κώδικα. Σχεδιάστηκε πρωτίστως για αποθήκευση μεταδεδομένων και χρησιμοποιείται συνήθως για εντοπισμό υπηρεσιών, κοινόχρηστη ρύθμιση παραμέτρων και κατανεμημένα κλειδώματα. Ωστόσο, η υψηλή του διαθεσιμότητα, η απλότητα και οι αξιοσημείωτες επιδόσεις, σε συνδυασμό με το ότι υποστηρίζεται από μια μεγάλη και δραστήρια κοινότητα προγραμματιστών, τον καθιστούν ελκυστική επιλογή και για άλλα είδη εφαρμογών. Ο τρέχων αποθηκευτικός μηχανισμός του etcd είναι η *BoltDB*, μια ενσωματωμένη βάση δεδομένων που στηρίζεται στα B+ δέντρα και είναι βελτιστοποιημένη για αναγνώσεις. Στην εποχή των κατανεμημένων συστημάτων, υπάρχει ένα ευρύ φάσμα εφαρμογών με έμφαση στις εγγραφές που θα επωφελούνταν από έναν αξιόπιστο τρόπο αποθήκευσης δεδομένων σε μια συστοιχία υπολογιστών. Παραδείγματα τέτοιων εφαρμογών αποτελούν οι διαδικτυακές υπηρεσίες αντιγράφων ασφαλείας, η συλλογή δεδομένων από αισθητήρες, οι εξυπηρετητές ηλεκτρονικού ταχυδρομείου και οι ιστότοποι κοινωνικής δικτύωσης. Έτσι, σε αυτήν την εργασία σχεδιάζουμε και υλοποιούμε την αντικατάσταση της BoltDB με την *RocksDB*, μια υψηλών επιδόσεων βάση δεδομένων που εσωτερικά χρησιμοποιεί ένα LSM-δέντρο, με σκοπό να βελτιστοποιήσουμε τον etcd για περιπτώσεις χρήσης συχνών εγγραφών. Αυτό το καταφέρνουμε αναπτύσσοντας ένα πακέτο κώδικα σε γλώσσα Go που *μεταφράζει* συναρτήσεις και έννοιες από την προγραμματιστική διεπαφή (API) της BoltDB στις αντίστοιχες της RocksDB, τροποποιώντας τον κώδικα του ίδιου του etcd μόνο στα σημεία που είναι απαραίτητο. Κατά τη διάρκεια της διαδικασίας αυτής, υλοποιούμε κάποιες δυνατότητες που δεν υπήρχαν ήδη στα εμπλεκόμενα προγράμματα, κάνοντας και τις αντίστοιχες συνεισφορές λογισμικού. Στη συνέχεια, επαληθεύουμε την ορθότητα και την ευρωστία της υλοποίησής μας, με χρήση της πλατφόρμας λειτουργικού ελέγχου του etcd μεταξύ άλλων εργαλείων. Επιπλέον, δημιουργούμε μια συστοιχία εικονικών μηχανών σε περιβάλλον νέφους, ώστε να αξιολογήσουμε μέσω του ενσωματωμένου εργαλείου benchmark την επίδοση του etcd στην περίπτωση χρήσης της RocksDB ως αποθηκευτικό μηχανισμό και να την συγκρίνουμε με την επίδοση στην περίπτωση χρήσης της BoltDB. Έπειτα, εφαρμόζουμε σταδιακά κάποιες βελτιστοποιήσεις, εξετάζουμε την επίδραση ενός συνόλου παραμέτρων στα αποτελέσματα, και σχολιάζουμε τα πλεονεκτήματα και μειονεκτήματα των δύο προσεγγίσεων. Τέλος, σκιαγραφούμε ιδέες για βελτίωση και περαιτέρω έρευνα πάνω στο θέμα.

Λέξεις-Κλειδιά

σύστημα αποθήκευσης κλειδιού-τιμής, κατανεμημένη αποθήκευση, μηχανισμός αποθήκευσης, δομές δεδομένων, B+ δέντρο, LSM-δέντρο, etcd, BoltDB, RocksDB

Abstract

Etcd is an open-source, distributed key-value store with a focus on reliability. It was primarily designed to store metadata and is often used for service discovery, shared configuration and distributed locking. However, its numerous good qualities, such as its high-availability, simplicity and notable performance, in conjunction with the fact that it is an actively maintained project backed by a large developer community, render it an attractive option for other usecases as well. Etcd currently uses *BoltDB*, a read-optimized persistence solution based on B+ trees, as its storage engine. In the era of distributed computing, there is a wide range of write-intensive applications that would benefit from having a reliable way to store data across a cluster of machines. Examples of such applications include cloud backup services, sensor data collection, mail servers and social media websites. To that end, in this thesis we design and implement the replacement of BoltDB with *RocksDB*, a high-performance embedded database that internally uses a log-structured merge-tree (LSM-tree), in order to optimize etcd for write-heavy workloads. We achieve this by developing a Go wrapper that maps BoltDB API functions and concepts to their RocksDB counterparts, modifying core etcd code only where necessary. During this process, we make software contributions to some of the projects involved, implementing features that we needed but were missing. Furthermore, we verify the functionality and robustness of our implementation, using the functional test suite of etcd, among other tools. In addition, we set up a cluster of virtual machines on a cloud platform, in order to evaluate the performance of etcd with RocksDB as its storage engine using the built-in benchmark tool, and compare it to that of BoltDB-backed etcd. Then, we gradually apply a number of optimizations upon our initial implementation, examine the impact of a set of parameters on the results, and comment on the trade-offs of both approaches. Finally, we suggest some improvements and outline directions for further investigation on this topic.

Keywords

key-value store, distributed storage, storage engine, data structures, B+ tree, LSM-tree, etcd, BoltDB, RocksDB

Αντί Προλόγου

Στο σημείο αυτό θα ήθελα να εκφράσω την ευγνωμοσύνη μου προς τους ανθρώπους που συνέδραμαν στην ολοκλήρωση αυτής της διπλωματικής εργασίας, αλλά και στην ευρύτερη ακαδημαϊκή μου πορεία. Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Νεκτάριο Κοζύρη, που μέσα από τις διαλέξεις του καλλιέργησε το ενδιαφέρον μου για τον τομέα των Υπολογιστικών Συστημάτων. Επίσης, ευχαριστώ θερμά τον διδάκτορα Βαγγέλη Κούκη για την εμπιστοσύνη που μου έδειξε, για τον μεταδοτικό ενθουσιασμό του και τις πολύτιμες γνώσεις που αποκόμισα κατά τη διάρκεια της εκπόνησης αυτής της εργασίας χάρη στη βοήθειά του. Ακόμη, θέλω να ευχαριστήσω τον Χρήστο Σταυρακάκη, που είναι και ο εμπνευστής του θέματος αυτής της διπλωματικής, για την ουσιαστική συμβολή του στην αντιμετώπιση τόσο τεχνικών όσο και θεωρητικών ζητημάτων, καθώς και τον συμφοιτητή μου Χρήστο Κατσακιώρη για τις ιδιαίτερα εποικοδομητικές συζητήσεις μας γύρω από το αντικείμενο των Κατανεμημένων Συστημάτων Αποθήκευσης. Επιπλέον, θέλω να εκφράσω την ευγνωμοσύνη μου προς όσες και όσους υποστηρίζουν έμπρακτα την ελεύθερη και δωρεάν διακίνηση της γνώσης, αναφέροντας χαρακτηριστικά τους προγραμματιστές ελεύθερου λογισμικού. Τέλος, ευχαριστώ από καρδιάς την οικογένειά μου και τους αγαπημένους μου φίλους για τη στήριξη, την κατανόηση και την ανεκτίμητη συντροφιά τους.

Γεωργία Κοκκίνου
Μάρτιος 2018

Contents

Περίληψη iii

Abstract v

Αντί Προλόγου vii

List of figures xiii

List of tables xiv

Βελτιστοποίηση Εγγραφών στον etcd μέσω Ενσωμάτωσης της RocksDB 1

1 Εισαγωγή 1

1.1 Σκοπός & Κίνητρο 1

1.2 Υπάρχουσες Προσεγγίσεις 2

2 Υπόβαθρο 3

2.1 Το Σύστημα Αποθήκευσης Κλειδιού-Τιμής etcd 3

2.2 Ο Αποθηκευτικός Μηχανισμός BoltDB 9

2.3 Ο Αποθηκευτικός Μηχανισμός RocksDB 16

3 Σχεδιασμός 24

3.1 Αρχιτεκτονική & Σχεδιαστικές Επιλογές 24

3.2 Αντιστοίχιση Εννοιών & Δομών 29

4 Υλοποίηση 32

4.1 Η Βιβλιοθήκη Περιτύλιξης της RocksDB στην BoltDB 32

4.2 Τροποποιήσεις στον Κώδικα του etcd 35

4.3 Βελτιστοποιήσεις 38
4.4 Εξωτερικές Συνεισφορές 44
4.5 Έλεγχος Ορθότητας 45
5 Πειραματική Αξιολόγηση 47
5.1 Εργαλεία, Μεθοδολογία & Περιβάλλον 47
5.2 Αποτελέσματα 48
6 Επίλογος 56
6.1 Συμπεράσματα 56
6.2 Μελλοντικές Δυνατότητες 56

1 Introduction **59**
1.1 Problem Statement 59
1.2 Motivation 60
1.3 Existing Solutions 61
1.3.1 CockroachDB 62
1.3.2 TiKV 63
1.3.3 Other Related Work 64
1.4 Thesis Structure 66

2 Background **67**
2.1 Distributed Systems & Data Storage Concepts 67
2.1.1 An Overview of Distributed Storage 67
2.1.2 Key-Value Stores 69
2.1.3 CAP Theorem 71
2.1.4 ACID Properties 73
2.1.5 Consistency Models & Isolation Levels 74
2.1.6 Write-Ahead Logging 76
2.1.7 Multi-Version Concurrency Control 77
2.1.8 The Raft Consensus Algorithm 79
2.2 etcd Distributed Key-Value Store 84
2.2.1 Overview 84
2.2.2 Evolution of the Storage Backend 89
2.2.3 Use Cases 90
2.2.4 Similar Systems 93
2.2.5 Performance 94

2.3 BoltDB Storage Engine 96
2.3.1 B+ Trees 96
2.3.2 Basic Concepts & API 102
2.3.3 Caveats & Limitations 104
2.4 RocksDB Storage Engine 106
2.4.1 Log-Structured Merge-Trees 106
2.4.2 Basic Concepts & API 115
2.4.3 Comparison with BoltDB 119
2.5 The Go Programming Language 121
2.5.1 Overview 121
2.5.2 Cgo: A Necessary Evil 124

3 Design **129**
3.1 Proposed Architecture & Design Choices 130
3.1.1 Integration of RocksDB into etcd 130
3.1.2 Why RocksDB? 132
3.1.3 The gorocksdb Wrapper 132
3.1.4 The RocksDB C API 134
3.1.5 Removing the Storage Quota 135
3.2 Mapping of Concepts & Constructs 136
3.2.1 DB 136
3.2.2 Buckets to Prefixes 136
3.2.3 Get, Put & Delete Operations 138
3.2.4 Cursor to Iterator 138
3.2.5 Transactions 139
3.2.6 Snapshot to Checkpoint 139
3.2.7 Defragmentation 141
3.2.8 Read-Only Mode 141

4 Implementation **143**
4.1 Wrapping RocksDB in BoltDB 143
4.2 Modifications in etcd Code 168
4.3 Optimizations 174
4.3.1 Base Implementation 174
4.3.2 Bucket Access 174

4.3.3 Optimistic Transactions 178
4.3.4 WriteBatchWithIndex & Snapshot 178
4.3.5 WriteBatch 180
4.3.6 Tuning RocksDB 181
4.4 External Contributions 187
4.5 Installation & Configuration 191
4.6 Testing 195
4.6.1 Unit Tests 196
4.6.2 Functional Test Suite 196

5 Experimental Evaluation **199**
5.1 Tools, Methodology & Environment 200
5.2 Results 201

6 Conclusion **213**
6.1 Concluding Remarks 213
6.2 Future Work 214

Bibliography **217**

List of figures

1.1 Overview of CockroachDB architecture 62

1.2 Sharding and region replication in TiKV 64

1.3 Basic concept of a heterogeneous MyCassandra cluster 65

2.1 Visualization of the CAP theorem 72

2.2 Replicated state machine architecture 79

2.3 The leader election process in the Raft consensus algorithm 81

2.4 Replicated Raft logs 82

2.5 Log compaction in Raft 83

2.6 v2 and v3 storage engines in etcd 90

2.7 Average throughput as clients scale: etcd vs ZooKeeper vs Consul . . . 96

2.8 A B-tree of order 5 97

2.9 A B+ tree of order 4 99

2.10 A copy-on-write B+ tree with 3 levels 101

2.11 On-disk layout of the pages of a B+ tree 101

2.12 A copy-on-write B+ tree after updating a value 102

2.13 Internal fragmentation in BoltDB 105

2.14 External fragmentation in BoltDB 106

2.15 Schematic diagram of an LSM-tree of two components 107
2.16 LSM-tree storage 108
2.17 A Bloom filter with a 32-bit array and 3 hash functions 110
2.18 Levelled compaction layout in LSM-trees 112
2.19 The architecture of RocksDB 115
2.20 Popularity growth of golang 122

3.1 Architecture of the etcd backend with RocksDB as its storage engine . 131

4.1 The role of the `WriteTo()` function 154
4.2 CPU profile of the base implementation 176
4.3 The functional test suite of etcd 197

5.1 Write throughput of etcd in all versions 202
5.2 Average write latency of etcd in all versions 203
5.3 Contributing factors in the write latency of etcd 204
5.4 Point lookup throughput of etcd in all versions 204
5.5 Average point lookup latency of etcd in all versions 205
5.6 Range query throughput of etcd in all versions 206
5.7 Average range query latency of etcd in all versions 206
5.8 The impact of cgo overhead on put latency 207
5.9 Total duration of put and commit operations 208
5.10 Write throughput of etcd with number of clients 209
5.11 Average write latency of etcd with value size 210
5.12 On-disk size of the backend database of etcd 211
5.13 Memory consumption of etcd 212

List of tables

2.1 Quorum and fault tolerance in relation to etcd cluster size 89

2.2 Write performance of etcd 95

2.3 Read performance of etcd 95

2.4 Complexity of basic operations in B-trees and B+ trees 100

2.5 Read & write amplification in universal and levelled style compaction 112

3.1 Key differences between BoltDB and RocksDB 142

1 Εισαγωγή

1.1 Σκοπός & Κίνητρο

Τα *κατανεμημένα συστήματα αποθήκευσης* γίνονται σήμερα ολοένα και πιο δημοφιλή εξαιτίας της κλιμακωσιμότητάς τους, της υψηλής διαθεσιμότητας και της ανοχής σε σφάλματα που παρουσιάζουν. Ένα *σύστημα αποθήκευσης κλειδιού-τιμής* είναι ένα είδος NoSQL βάσης δεδομένων που εσωτερικά χρησιμοποιεί έναν πίνακα συσχέτισης για την αποθήκευση δεδομένων στη μορφή ζευγών κλειδιού-τιμής.

Ο *etcd* είναι ένα κατανεμημένο σύστημα αποθήκευσης κλειδιού-τιμής που χρησιμοποιείται συνήθως για την αποθήκευση των κρίσιμων δεδομένων άλλων κατανεμημένων συστημάτων. Για την επίτευξη ομοφωνίας σε σχέση με την κατάσταση των δεδομένων μεταξύ των κόμβων που απαρτίζουν μια συστοιχία (cluster) etcd χρησιμοποιείται ο αλγόριθμος ομοφωνίας (consensus) Raft. Ο σχεδιασμός του etcd ευνοεί την επίδοση των αιτημάτων ανάγνωσης, καθώς αυτά απαντώνται συχνότερα στην συνήθη περίπτωση χρήσης του, αφήνοντας την επίδοση των αιτημάτων εγγραφής σε δεύτερη μοίρα. Ο τρέχων μηχανισμός αποθήκευσης του etcd είναι η *BoltDB*, μια υλοποίηση της βελτιστοποιημένης για αναγνώσεις δομής δεδομένων *B+ δέντρο*.

Σκοπός της παρούσας εργασίας είναι η βελτιστοποίηση του etcd για την περίπτωση φόρτου εργασίας με έμφαση στις εγγραφές, μέσω αντικατάστασης του μηχανισμού αποθήκευσής του. Ως εναλλακτικός μηχανισμός αποθήκευσης επιλέχθηκε η *RocksDB*, ένα τοπικό σύστημα αποθήκευσης κλειδιού-τιμής βασισμένο στα *LSM-δέντρα*, τα οποία είναι δομή δεδομένων σχεδιασμένη ειδικά για αποδοτικές εγγραφές.

Στην εποχή των Big Data, όλο και περισσότερες εφαρμογές παρουσιάζουν έμφαση στις εγγραφές και θα μπορούσαν να επωφεληθούν από τη συνεισφορά μας. Ακολουθούν μερικά παραδείγματα: συστήματα συλλογής δεδομένων (π.χ. από δίκτυα αισθητήρων, διαδικτυακές έρευνες, παρακολούθηση της κίνησης στο οδικό δίκτυο), διαδικτυακές υπηρεσίες διατήρησης αντιγράφων ασφαλείας, εξυπηρετητές ηλεκτρονικού ταχυδρομείου, ιστότοποι κοινωνικής δικτύωσης, διαδραστικά παιχνίδια, διατήρηση αρχείων καταγραφής γεγονότων (event logs) και τέλος διαδικτυακά συστήματα διεκπεραίωσης συναλλαγών (π.χ. παραγγελιών, κρατήσεων αεροπορικών εισιτηρίων).

1.2 Υπάρχουσες Προσεγγίσεις

Μια προσέγγιση παρόμοια με τη δική μας συναντάμε στην CockroachDB [8], μια κατανεμημένη SQL βάση δεδομένων που λειτουργεί πάνω από ένα ισχυρής συνέπειας σύστημα αποθήκευσης κλειδιού-τιμής που υποστήριζει συναλλαγές (transactions). Πρόκειται για ένα υψηλής διαθεσιμότητας, συνεπές και ανεκτικό στις διαμερίσεις δικτύου σύστημα (CP), που χρησιμοποιεί το πρωτόκολλο ομοφωνίας Raft, όπως ο etcd.

Η αρχιτεκτονική της CockroachDB περιλαμβάνει ένα σύνολο κόμβων, κάθε ένας από τους οποίους περιέχει ένα ή περισσότερα συστήματα αποθήκευσης. Κάθε ένα από αυτά τα συστήματα αποθήκευσης είναι τοποθετημένο σε ξεχωριστό δίσκο και χρησιμοποιεί την RocksDB ως αποθηκευτικό μηχανισμό. Η CockroachDB εξασφαλίζει οριζόντια κλιμακωσιμότητα μέσω κατάτμησης (sharding) του πεδίου κλειδιών και διαμοιρασμού του στα προαναφερθέντα συστήματα αποθήκευσης. Για κάθε τμήμα του πεδίου κλειδιών διατηρούνται αντίγραφα σε τρεις τουλάχιστον κόμβους. Η βασική διαφορά της CockroachDB με τον etcd εντοπίζεται στο ότι μπορεί να διαχειριστεί όγκο δεδομένων της τάξεως των terabytes, καθώς η προσθήκη νέων κόμβων στο σύστημα αυξάνει τη χωρητικότητά του. Από την άλλη πλευρά, η ύπαρξη του επιπέδου SQL, καθώς και η ανάγκη συντονισμού μεταξύ των διαφόρων ομάδων αντιγράφων εισάγουν επιπρόσθετες καθυστερήσεις.

Το TiKV [11] είναι ένα ακόμα συνεπές, κατανεμημένο σύστημα αποθήκευσης κλειδιού-τιμής με υποστήριξη συναλλαγών, που χρησιμοποιεί τον αλγόριθμο ομοφωνίας Raft και τον αποθηκευτικό μηχανισμό RocksDB. Ακολουθεί μια στρατηγική κατάτμησης παρόμοια με αυτήν της CockroachDB, στην οποία ο Οδηγός Τοποθέτησης (Placement Driver), που υλοποιείται ως μια συστοιχία etcd, έχει το ρόλο του ενορχηστρωτή. Ωστόσο, το TiKV υστερεί σε σχέση με τον etcd στην πληρότητα του API[1] του.

[1] Application programming interface / διεπαφή προγραμματισμού εφαρμογών

2 Υπόβαθρο

Στην ενότητα αυτή θα παρουσιάσουμε τα βασικά θεωρητικά στοιχεία που είναι απαραίτητα για την κατανόηση της εργασίας μας, δίνοντας έμφαση στην αρχιτεκτονική και στη λειτουργία του etcd, της BoltDB και της RocksDB, καθώς και στις δομές δεδομένων στις οποίες αυτές βασίζονται.

2.1 Το Σύστημα Αποθήκευσης Κλειδιού-Τιμής etcd

Στα συστήματα αποθήκευσης κλειδιού-τιμής ο ρόλος του κλειδιού είναι να ταυτοποιεί μοναδικά μια τιμή, η οποία περιέχει τα πραγματικά δεδομένα. Το κλειδί έχει τη μορφή αυθαίρετης συμβολοσειράς, ενώ η τιμή μπορεί να είναι οποιοσδήποτε τύπος δεδομένων, π.χ. εικόνα, έγγραφο κτλ. Στη γενική περίπτωση, τα συστήματα αποθήκευσης κλειδιού-τιμής δεν διαθέτουν κάποια γλώσσα ερωτημάτων (query language). Διαχειρίζονται τα δεδομένα μέσω απλών εντολών ανάγνωσης, εγγραφής και διαγραφής. Η *απλότητα* αυτού του μοντέλου τα καθιστά ταχέα, εύκολα στη χρήση, κλιμακώσιμα και ευέλικτα.

Ο etcd [1] είναι ένα ανοιχτού κώδικα, κατανεμημένο σύστημα αποθήκευσης κλειδιού-τιμής για αξιόπιστη αποθήκευση δεδομένων σε μια συστοιχία υπολογιστών. Είναι ένα συνεπές, ανεκτικό σε σφάλματα, υψηλής διαθεσιμότητας σύστημα. Με βάση την ορολογία του θεωρήματος CAP[2], ο etcd αποτελεί σύστημα CP.

Τρέχει σε κάθε ένα από τα μηχανήματα μιας συστοιχίας και χρησιμοποιεί τον αλγόριθμο Raft για την επίτευξη ομοφωνίας. Μια συστοιχία αποτελείται συνήθως από 3 μέλη, αλλά συχνά χρησιμοποιούνται και συστοιχίες με περισσότερους κόμβους με σκοπό την αύξηση της ανοχής σε σφάλματα. Εφαρμογές που τρέχουν σε μηχανήματα-πελάτες στέλνουν αιτήματα ανάγνωσης (range), εγγραφής (put) ή διαγραφής (delete) στον etcd. Κάθε εισερχόμενο αίτημα τροποποίησης πρέπει να περάσει από το πρωτόκολλο Raft πριν κατοχυρωθεί.

Ο etcd είναι γραμμένος στη γλώσσα προγραμματισμού Go και η ονομασία του προέρχεται από τον κατάλογο "`/etc`", που χρησιμοποιείται για την αποθήκευση ρυθμίσεων

[2]Σύμφωνα με το θεώρημα CAP, είναι αδύνατον ένα κατανεμημένο σύστημα να παρέχει ταυτόχρονα συνέπεια (consistency), διαθεσιμότητα (availability) και ανοχή στις διαμερίσεις (partition tolerance) [24].

συστήματος, και το γράμμα “d” από τη λέξη distributed (κατανεμημένος). Ακολουθούν συνήθεις περιπτώσεις χρήσης του etcd:

- **Κοινόχρηστες ρυθμίσεις:** οι κόμβοι ενός κατανεμημένου συστήματος που αποθηκεύει τις παραμέτρους του σε μια συστοιχία etcd μπορούν να μοιράζονται τις ίδιες ρυθμίσεις και να ενημερώνονται αυτόματα για οποιεσδήποτε αλλαγές σε αυτές.

- **Ανακάλυψη υπηρεσιών:** σε αυτή την περίπτωση η συστοιχία etcd παίζει το ρόλο ενός μητρώου διαθέσιμων υπηρεσιών στο δίκτυο, που αυτοματοποιεί την ανίχνευση τους από τρίτους. Η συστοιχία etcd διατηρεί επίσης και πληροφορίες σύνδεσης στην κάθε υπηρεσία (π.χ. διεύθυνση IP, αριθμός θύρας).

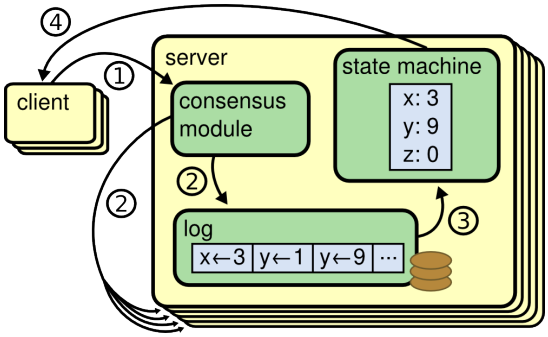
- **Κατανεμημένα κλειδώματα:** σκοπός ενός κατανεμημένου κλειδώματος είναι να εξασφαλίσει ότι μεταξύ ορισμένων κόμβων που θα αποπειραθούν να κάνουν μια συγκεκριμένη εργασία, μόνο ένας κάθε φορά θα την κάνει. Χαρακτηριστικό παράδειγμα εφαρμογής είναι η εκτέλεση ενημερώσεων. Σε αυτή την περίπτωση, εάν όλοι οι κόμβοι ενός κατανεμημένου συστήματος εκτελούσαν τις ενημερώσεις και επανεκκινούσαν ταυτόχρονα, το σύστημα θα έχανε τη διαθεσιμότητά του.

2.1.1 Ο αλγόριθμος ομοφωνίας Raft

Η διατήρηση αντιγράφων μηχανής καταστάσεων (state-machine replication) στους κόμβους μιας συστοιχίας υπολογιστών, δηλαδή η τεχνική που χρησιμοποιείται για την επίτευξη της ανοχής σε σφάλματα στα κατανεμημένα συστήματα αποθήκευσης, εισάγει την ανάγκη επίτευξης ομοφωνίας. Η ομοφωνία αναφέρεται στη διαδικασία επίτευξης συναίνεσης μεταξύ διαφορετικών κόμβων, σχετικά με τις τιμές των αποθηκευμένων δεδομένων.

Κεντρικό ρόλο στην διατήρηση αντιγράφων μηχανής καταστάσεων παίζει το *αρχείο καταγραφής* (replicated log). Αυτό βρίσκεται αποθηκευμένο σε κάθε κόμβο της συστοιχίας και περιέχει μια ακολουθία εντολών, τις οποίες η μηχανή καταστάσεων του κόμβου εκτελεί με τη σειρά. Οι αλγόριθμοι ομοφωνίας εξασφαλίζουν τη συνέπεια του αρχείου καταγραφής, δηλαδή φροντίζουν αυτό να περιέχει τις ίδιες εντολές με την ίδια σειρά σε όλους τους κόμβους [35]. Η συνέπεια του αρχείου καταγραφής συνεπάγεται

ότι όλες οι ενδιάμεσες αλλά και η τελική κατάσταση όλων των μηχανών καταστάσεων είναι ίδιες.



Σχήμα 1: *Αρχιτεκτονική αντιγράφων μηχανής καταστάσεων [35]*

Στον αλγόριθμο ομοφωνίας Raft[3] κάθε κόμβος βρίσκεται σε μία από τις εξής τρεις καταστάσεις: *ηγέτης*, *ακόλουθος* ή *υποψήφιος*. Υπό κανονικές συνθήκες υπάρχει μόνο ένας ηγέτης και οι υπόλοιποι κόμβοι είναι ακόλουθοι. Ο αλγόριθμος Raft συνεχίζει τη λειτουργία του όσο είναι διαθέσιμη οποιαδήποτε πλειοψηφία κόμβων της συστοιχίας. Διασπά το πρόβλημα της επίτευξης ομοφωνίας στα παρακάτω δύο υποπροβλήματα:

- **Εκλογή ηγέτη:** οι κόμβοι εκκινούν στην κατάσταση ακολούθου και παραμένουν σε αυτήν για όσο διάστημα λαμβάνουν ειδικά περιοδικά μηνύματα από τον ηγέτη. Αν ένας ακόλουθος δεν λάβει τέτοιο μήνυμα κατά τη διάρκεια του *χρονικού ορίου εκλογής*, μεταβαίνει στην κατάσταση υποψηφίου. Τότε, είτε θα λάβει ψήφους από την πλειοψηφία των μελών της συστοιχίας και θα εκλεχθεί ηγέτης, είτε θα ειδοποιηθεί ότι κάποιος άλλος έχει εκλεχθεί και θα επιστρέψει στην κατάσταση ακολούθου. Οι ψήφοι δίνονται με σειρά προτεραιότητας υπό την προϋπόθεση ότι το αρχείο καταγραφής του υποψηφίου είναι τουλάχιστον εξίσου ενημερωμένο με το αυτό του κόμβου ακολούθου, ενώ κάθε υποψήφιος ψηφίζει τον εαυτό του. Εάν υπάρξει ισοψηφία η διαδικασία εκλογής επαναλαμβάνεται. Επιπλέον, σε περίπτωση διαμέρισης του δικτύου, όταν αυτό επανέλθει στην φυσιολογική του κατάσταση υπάρχει περίπτωση δύο κόμβοι να βρίσκονται ταυτόχρονα σε κατάσταση ηγέτη. Τότε ο ηγέτης με το λιγότερο πρόσφατο αριθμό περιόδου παραχωρεί τη θέση του [35]. Η περίοδος στην ορολογία του Raft είναι μια μονάδα χρόνου αυθαίρετης διάρκειας που χαρακτηρίζεται από έναν συγκεκριμένο ηγέτη.

[3] `https://github.com/coreos/etcd/tree/master/raft`

- **Αντιγραφή αρχείου καταγραφής:** ο ηγέτης λαμβάνει από τους πελάτες (clients) αιτήματα, κάθε ένα από τα οποία περιέχει μια εντολή προς εκτέλεση από τις μηχανές καταστάσεων. Πρώτα, προσαρτά την εντολή στο αρχείο καταγραφής του. Ύστερα, την διανέμει στους υπόλοιπους κόμβους. Μόλις λάβει επιβεβαίωση λήψης της εντολής από την πλειοψηφία των κόμβων, ο ηγέτης εφαρμόζει την εντολή στη μηχανή καταστάσεών του. Σε αυτό το σημείο η εντολή θεωρείται *κατοχυρωμένη* (committed). Τότε ο ηγέτης επιστρέφει το αποτέλεσμα στον πελάτη και ενημερώνει τους υπόλοιπους κόμβους ότι μπορούν να εφαρμόσουν κι αυτοί την εντολή στις μηχανές καταστάσεών τους. Οι ασυνέπειες που μπορεί να προκύψουν μεταξύ των αρχείων καταγραφής όταν ο ηγέτης υποστεί σφάλμα επιλύονται με την επιβολή του αρχείου καταγραφής του νέου ηγέτη στους υπόλοιπους κόμβους [35].

Για να αποτραπεί η χωρίς όριο αύξηση του μεγέθους του αρχείου καταγραφής, εφαρμόζεται μια τεχνική "*συμπύκνωσης*" (compaction). Σύμφωνα με αυτήν, η τρέχουσα κατάσταση του συστήματος εγγράφεται σε ένα *στιγμιότυπο* (snapshot) στον δίσκο και το αρχείο καταγραφής μέχρι εκείνο το σημείο απορρίπτεται.

2.1.2 Εγγυήσεις του etcd

- **Ατομικότητα:** κάθε αίτημα στον etcd είτε ολοκληρώνεται είτε δεν πραγματοποιείται καθόλου.

- **Συνέπεια:** ο etcd εξασφαλίζει ακολουθιακή συνέπεια, δηλαδή ανεξαρτήτως του εξυπηρετητή etcd στον οποίο ο πελάτης στέλνει ένα αίτημα, θα διαβάσει τα ίδια γεγονότα με την ίδια σειρά.

- **Απομόνωση:** τα αιτήματα ανάγνωσης δεν "βλέπουν" ποτέ δεδομένα που προστέθηκαν κατά τη διάρκεια διεκπεραίωσής τους (σειριοποιήσιμη απομόνωση).

- **Ανθεκτικότητα:** όλα τα ολοκληρωμένα αιτήματα είναι ανθεκτικά, καθώς εγγράφονται σε συσκευή μόνιμης αποθήκευσης.

- **Υψηλή διαθεσιμότητα:** μια συστοιχία etcd μπορεί να συνεχίσει την κανονική λειτουργία της όσο υπάρχει μια λειτουργική πλειοψηφία κόμβων.

- **Ανοχή διαμερίσεων:** αν η συστοιχία αποτελείται από περιττό αριθμό κόμβων είναι βέβαιο ότι ο etcd θα μπορέσει να συνεχίσει τη λειτουργία του κανονικά σε περίπτωση διαμέρισης του δικτύου, καθώς ένα από τα τμήματα που θα έχουν προκύψει θα περιέχει την πλειοψηφία των κόμβων.

2.1.3 Μοντέλο δεδομένων

Ο etcd υιοθετεί *έλεγχο ταυτοχρονισμού πολλαπλών εκδόσεων*[4] (multi-version concurrency control / MVCC). Έτσι, κάθε τροποποίηση δημιουργεί μια νέα αναθεώρηση (revision) του πεδίου κλειδιών. Η λογική όψη του αποθηκευτικού συστήματος είναι ένα επίπεδο (χωρίς καταλόγους), αλφαβητικά ταξινομημένο πεδίο κλειδιών. Ο etcd αποθηκεύει τα φυσικά δεδομένα ως ζεύγη κλειδιού-τιμής στον δίσκο, σε ένα B+ δέντρο, υλοποιημένο από τον μηχανισμό αποθήκευσής του, την BoltDB. Κάθε κλειδί που εισάγεται στο B+ δέντρο έχει τη μορφή μιας πλειάδας τριών στοιχείων: (major, sub, type). Το στοιχείο major είναι η αναθεώρηση στην οποία πραγματοποιήθηκε η σχετική τροποποίηση. Το στοιχείο sub διαφοροποιεί μεταξύ τροποποιήσεων με τον ίδιο αριθμό αναθεώρησης, δηλαδή τροποποιήσεων που έγιναν κατά τη διάρκεια της ίδιας συναλλαγής. Το στοιχείο type καθορίζει τον τύπο της τροποποίησης (π.χ. put, tombstone). Η τιμή που συνοδεύει το κλειδί που εισάγεται στο B+ δένδρο, περιέχει το πραγματικό ζεύγος κλειδιού τιμής στο οποίο αναφέρεται η τροποποίηση, μαζί με άλλα μεταδεδομένα. Ο etcd διατηρεί επίσης ένα δευτερεύον B-δέντρο στη μνήμη, ώστε να επιταχύνει την εξυπηρέτηση των ερωτημάτων ανάγνωσης. Τα κλειδιά σε αυτό το B-δέντρο είναι τα πραγματικά κλειδιά, δηλαδή αυτά που "βλέπει" ο πελάτης, ενώ οι τιμές περιέχουν δείκτες στην πιο πρόσφατη τροποποίησή των κλειδιών στο B+ δέντρο [40].

Για να αποτραπεί η υπερβολική αύξηση του μεγέθους του B+ δέντρου λόγω της συσσώρευσης προηγούμενων αναθεωρήσεων, ο etcd διενεργεί περιοδικά μια διαδικασία *συμπύκνωσης* (compaction), κατά την οποία απορρίπτονται οι παρωχημένες εκδόσεις των δεδομένων.

[4]Σε μια βάση δεδομένων που χρησιμοποιεί MVCC, οι ενημερώσεις δεν αντικαθιστούν τα παλιά δεδομένα. Αντιθέτως, τα επισημαίνουν ως παρωχημένα και προσθέτουν τη νέα έκδοσή τους. Έτσι, διατηρούνται πολλαπλές εκδόσεις, αλλά μόνο μία είναι η πιο πρόσφατη [34].

2.1.4 Αποκατάσταση καταστροφών

Σε περίπτωση που η πλειοψηφία των κόμβων μιας συστοιχίας etcd υποστεί σφάλμα, η συστοιχία σταματά να δέχεται αιτήματα. Για να ανανήψει από αυτή την κατάσταση απαιτείται ένα αρχείο στιγμιοτύπου (snapshot). Αυτού του τύπου τα στιγμιότυπα λειτουργούν ως αντίγραφα ασφαλείας των δεδομένων του etcd και ολόκληρη η συστοιχία μπορεί να αποκατασταθεί με χρήση ενός από αυτά [43]. Ο ίδιος τύπος στιγμιοτύπου στέλνεται μέσω του δικτύου σε νεοεισαχθέντα μέλη της συστοιχίας ή σε ακολούθους που έχουν υποστεί μεγάλη καθυστέρηση για να τους βοηθήσει να φτάσουν στο ίδιο σημείο προόδου με την υπόλοιπη συστοιχία.

Είναι χρήσιμο σε αυτό το σημείο να τονίσουμε ότι πρόκειται για διαφορετικό τύπο στιγμιοτύπου από αυτόν που χρησιμοποιείται στο επίπεδο του αλγορίθμου Raft για να επιτρέψει την συμπύκνωση του αρχείου καταγραφής. Ο δεύτερος αυτός τύπος δεν είναι προσβάσιμος μέσω του API. Ως στιγμιότυπο για την συμπύκνωση του αρχείου καταγραφής χρησιμοποιείται η ίδια η βάση δεδομένων του μηχανισμού αποθήκευσης BoltDB.

2.1.5 Επίδοση

Όταν μιλάμε για την επίδοση του etcd αναφερόμαστε ουσιαστικά σε δύο μετρικές: την διεκπεραιωτική ικανότητα (throughput) και την καθυστέρηση (latency) εξυπηρέτησης αιτημάτων. Η επίδοση του etcd καθορίζεται από πολλαπλούς παράγοντες [50]:

- **Καθυστέρηση Ε/Ε δίσκου:** η ενημέρωση του αρχείου καταγραφής του αλγορίθμου Raft απαιτεί συχνές κλήσεις της συνάρτησης `fsync`. Η συνήθης διάρκεια μιας τέτοιας κλήσης σε σκληρό δίσκο (HDD) ανέρχεται περίπου στα $10ms$, ενώ σε δίσκο στερεάς κατάστασης (SSD) είναι συχνά μικρότερη από $1ms$. Προκειμένου να αυξήσει την διεκπεραιωτική του ικανότητα, ο etcd ομαδοποιεί πολλαπλά αιτήματα (batching) και τα υποβάλει μαζί στον αλγόριθμο Raft. Με αυτόν τον τρόπο το κόστος της κλήσης `fsync` μοιράζεται μεταξύ των αιτημάτων. Ωστόσο, η ενημέρωση του αρχείου καταγραφής παραμένει καθοριστικός παράγοντας για την επίδοση των εγγραφών στον etcd.
- **Καθυστέρηση δικτύου:** ο ελάχιστος χρόνος που απαιτείται για να ολοκληρωθεί ένα αίτημα στον etcd ισούται με τον χρόνο αποστολής μετ' επιστροφής (RTT)

ενός μηνύματος μεταξύ των κόμβων της συστοιχίας συν τον χρόνο της κλήσης `fsync` που εγγράφει τα δεδομένα στον δίσκο. Το RTT εντός ενός κέντρου δεδομένων (datacenter) είναι της τάξης μερικών εκατοντάδων μs, εντός των ΗΠΑ ανέρχεται περίπου στα $50ms$, ενώ μπορεί να φτάσει και τα $400ms$ όταν οι κόμβοι βρίσκονται σε διαφορετικές ηπείρους.

- **Καθυστέρηση αποθηκευτικού μηχανισμού:** κάθε αίτημα στον etcd πρέπει τελικά να περάσει από τον αποθηκευτικό μηχανισμό BoltDB, κάτι το οποίο διαρκεί συνήθως μερικές δεκάδες μs.

- **Συμπύκνωση:** οι συμπυκνώσεις ιστορικού που συμβαίνουν στο παρασκήνιο επηρεάζουν την επίδοση του etcd. Ευτυχώς, η επίδρασή τους είναι συχνά ασήμαντη, καθώς γίνονται σταδιακά, οπότε δεν ανταγωνίζονται με τα αιτήματα για τους πόρους του συστήματος.

- **gRPC API:** το σύστημα κλήσης απομακρυσμένων διαδικασιών που χρησιμοποιεί ο etcd για την επικοινωνία μεταξύ των μελών της συστοιχίας, καθώς και για την επικοινωνία πελάτη-εξυπηρετητή, εισάγει μια μικρή επιπρόσθετη καθυστέρηση.

2.2 Ο Αποθηκευτικός Μηχανισμός BoltDB

Ο *αποθηκευτικός μηχανισμός* είναι ένα τμήμα λογισμικού, σκοπός του οποίου είναι η διαχείριση των δεδομένων που αποθηκεύονται στη μνήμη ή στον δίσκο. Συνήθως, ενσωματώνεται σε άλλα συστήματα λογισμικού που απαιτούν πρόσβαση σε δεδομένα. Κάθε αποθηκευτικός μηχανισμός υλοποιεί έναν αλγόριθμο ευρετηρίου (indexing algorithm) [15].

Ο αποθηκευτικός μηχανισμός BoltDB [58] είναι ένα ενσωματωμένο σύστημα αποθήκευσης κλειδιού-τιμής. Αποθηκεύει τα δεδομένα σε ένα *αρχείο αντιστοίχισης μνήμης* (memory-mapped file), υλοποιώντας ένα copy-on-write (αντιγραφής κατά την εγγραφή) B+ δέντρο που υποστηρίζει έλεγχο ταυτοχρονισμού πολλαπλών εκδόσεων. Το γεγονός αυτό καθιστά τις αναγνώσεις εξαιρετικά ταχείες, καθώς μπορούν να εκτελούνται ταυτόχρονα με τις εγγραφές χωρίς να χρειάζεται κάποιο κλείδωμα. Στην BoltDB επιτρέπεται να υπάρχει μόνο ένας εγγραφέας κάθε στιγμή, αλλά δεν υπάρχει περιορισμός στον αριθμό των αναγνωστών. Επίσης, η BoltDB διαθέτει συναλλαγές τύ-

που ACID[5] με σειριοποιήσιμη απομόνωση. Όλες οι ενέργειες γίνονται υποχρεωτικά μέσω συναλλαγών.

2.2.1 B+ δέντρα

Το B+ δέντρο, το οποίο είναι μια παραλλαγή του B-δέντρου, είναι η δομή ευρετηρίου που χρησιμοποιεί η BoltDB. Ακολούθως, θα δώσουμε τον ορισμό και τα χαρακτηριστικά και των δύο αυτών δομών.

Το **B-δέντρο** είναι μια γενίκευση του δυαδικού δέντρου αναζήτησης, υπό την έννοια ότι οι κόμβοι μπορούν να έχουν περισσότερα από δύο παιδιά [51]. Πρόκειται για μια αυτοεξισορροπούμενη δομή που διατηρεί τα δεδομένα ταξινομημένα. Οι ενέργειες (αναζητήσεις, εισαγωγές, διαγραφές) που γίνονται σε ένα B-δέντρο ολοκληρώνονται σε *λογαριθμικό* χρόνο. Σύμφωνα με τον ορισμό του D. Knuth [52], ένα B-δέντρο τάξης m ικανοποιεί τις παρακάτω συνθήκες:

1. Κάθε κόμβος έχει το πολύ m παιδιά.
2. Κάθε κόμβος που δεν είναι φύλλο (εκτός από τη ρίζα) έχει τουλάχιστον $\lceil m/2 \rceil$ παιδιά.
3. Η ρίζα έχει τουλάχιστον δύο παιδιά αν δεν είναι φύλλο.
4. Κάθε κόμβος που δεν είναι φύλλο και έχει k παιδιά περιέχει $k - 1$ κλειδιά.
5. Όλα τα φύλλα είναι στο ίδιο επίπεδο.

Για την ανάλυση πολυπλοκότητας των ενεργειών σε ένα B-δέντρο ωστόσο, μας βολεύει περισσότερο να χρησιμοποιήσουμε τον ορισμό του D. Comer [51], σύμφωνα με τον οποίο η τάξη d του B-δέντρου είναι ο ελάχιστος αριθμός κλειδιών σε έναν κόμβο που δεν είναι φύλλο. Το μήκος h του μονοπατιού από τη ρίζα σε οποιοδήποτε φύλλο καλείται *ύψος* του δέντρου και είναι στη χειρότερη περίπτωση ίσο με $\log_d n$, όπου n είναι ο αριθμός των κλειδιών που υπάρχουν στο δέντρο. Στο B-δέντρο τα κλειδιά κάθε εσωτερικού κόμβου λειτουργούν ως διαχωριστικές τιμές για τον καθορισμό των υποδέντρων του. Για παράδειγμα, αν ένας εσωτερικός κόμβος έχει 3 παιδιά, τότε θα πρέπει να περιέχει 2 κλειδιά: το a_1 και το a_2. Οι τιμές στο αριστερό υποδέντρο που εκκινεί

[5] Atomicity, consistency, isolation, durability / ατομικότητα, συνέπεια, απομόνωση, μονιμότητα.

από τον κόμβο αυτόν θα είναι μικρότερες του a_1, οι τιμές του μεσαίου υποδέντρου θα βρίσκονται μεταξύ των a_1 και a_2, ενώ οι τιμές στο δεξί υποδέντρο θα είναι μεγαλύτερες του a_2.

Αναζήτηση: η αναζήτηση σε ένα B-δέντρο είναι παρόμοια με αυτήν σε ένα δυαδικό δέντρο αναζήτησης. Ξεκινώντας από τη ρίζα και συγκρίνοντας το ζητούμενο κλειδί με τα κλειδιά που περιέχονται σε έναν κόμβο, επιλέγεται το κατάλληλο μονοπάτι. Η διαδικασία αυτή επαναλαμβάνεται σε κάθε κόμβο έως ότου να εντοπιστεί το κλειδί ή να καταλήξει σε φύλλο. Μια αναζήτηση σε B-δέντρο τάξης d που περιέχει n κλειδιά θα επισκεφθεί στη χειρότερη περίπτωση $1 + \log_d n$ κόμβους [51].

Εισαγωγή: η διαδικασία εισαγωγής περιλαμβάνει δύο βήματα. Αρχικά, διενεργείται μια αναζήτηση με στόχο να εντοπιστεί το κατάλληλο φύλλο για την εισαγωγή. Τότε, πραγματοποιείται η εισαγωγή και αν είναι απαραίτητο αποκαθίσταται η ισορροπία του δέντρου. Αν το φύλλο μπορεί να φιλοξενήσει άλλο ένα κλειδί χωρίς να καταστρατηγεί την πρώτη συνθήκη του ορισμού του D. Knuth, τότε δεν χρειάζεται να γίνει καμία άλλη ενέργεια. Αν ωστόσο το φύλλο είναι ήδη γεμάτο συμβαίνει *διάσπαση* του: τα πρώτα d από τα $2d + 1$ κλειδιά τοποθετούνται σε έναν κόμβο, τα τελευταία d σε έναν άλλον και το εναπομένον κλειδί προάγεται στον κόμβο γονέα. Αν ο κόμβος γονέας είναι κι αυτός γεμάτος εφαρμόζεται ξανά η διαδικασία διάσπασης. Στη χειρότερη περίπτωση, οι διασπάσεις θα φτάσουν μέχρι τη ρίζα και το ύψος του δέντρου θα αυξηθεί κατά ένα επίπεδο. Η διαδικασία εισαγωγής απαιτεί $\mathcal{O}(2 \log_d n)$ προσβάσεις σε κόμβους [51].

Διαγραφή: αυτή η διαδικασία απαιτεί πρώτα μια αναζήτηση που θα εντοπίσει το κλειδί προς διαγραφή. Σε περίπτωση που το κλειδί βρίσκεται σε κόμβο που δεν είναι φύλλο, ένα νέο διαχωριστικό κλειδί θα πρέπει να βρεθεί και να το αντικαταστήσει, ώστε οι αναζητήσεις στο δέντρο να συνεχίσουν να λειτουργούν. Το κλειδί αυτό βρίσκεται στο αριστερότερο φύλλο του δεξιού υποδέντρου που εκκινεί από την θέση που βρισκόταν το διαγεγραμμένο κλειδί. Μόλις πραγματοποιηθεί η αντικατάσταση, γίνεται έλεγχος για να διασφαλιστεί ότι εξακολουθούν να υπάρχουν τουλάχιστον d κλειδιά στο φύλλο. Σε περίπτωση *υποχείλισης*, για να αποκατασταθεί η ισορροπία μπορεί να μεταφερθεί στο φύλλο ένα κλειδί από κάποιο γειτονικό φύλλο που διαθέτει παραπάνω από d κλειδιά. Αν δεν υπάρχει κανένα τέτοιο φύλλο, τότε το ελλιπές φύλλο θα πρέπει να συγχωνευτεί με κάποιο άλλο. Τότε ο γονέας του χάνει ένα διαχωριστικό κλειδί και είναι πι-

θανό να χρειαστεί να εφαρμοστεί επανεξισορρόπηση του δέντρου, που θα εξαπλωθεί μέχρι τη ρίζα στη χειρότερη περίπτωση. Η διαδικασία διαγραφής απαιτεί $\mathcal{O}(2\log_d n)$ προσβάσεις σε κόμβους [51].

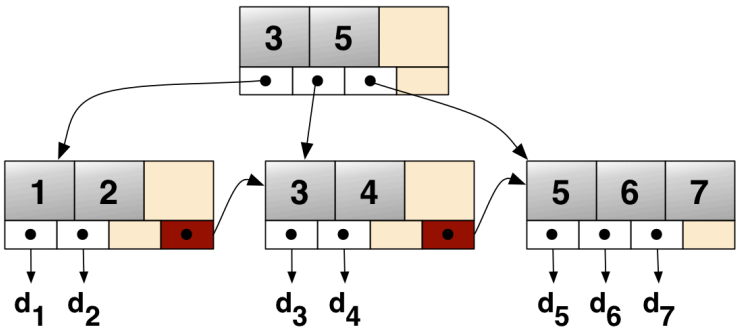
Ακολουθιακές προσβάσεις: μέχρι στιγμής έχουμε εξετάσει μόνο την περίπτωση των τυχαίων αναζητήσεων. Η επίδοση του B-δέντρου όμως δεν είναι το ίδιο καλή στην περίπτωση που ένα αίτημα αφορά ένα εύρος κλειδιών (range query). Η λειτουργία εύρεσης του επόμενου κλειδιού στο B-δέντρο μπορεί να χρειαστεί μέχρι και $\log_d n$ προσβάσεις σε κόμβους. Το πρόβλημα αυτό λύνεται από το B+ δέντρο με τον τρόπο που θα δούμε παρακάτω [51].

Το **B+ δέντρο** είναι μια δομή δεδομένων εξαιρετικά παρόμοια με το B-δέντρο, αλλά με μερικές ουσιαστικές διαφορές [51]:

1. Όλα τα κλειδιά βρίσκονται στα φύλλα. Τα ανώτερα επίπεδα του δέντρου αποτελούν απλώς ένα ευρετήριο, έναν χάρτη που καθιστά δυνατό τον αποτελεσματικό εντοπισμό των πραγματικών κλειδιών.

2. Τα φύλλα είναι συνδεδεμένα μεταξύ τους από αριστερά προς τα δεξιά, εξασφαλίζοντας έτσι αποδοτικές ακολουθιακές προσβάσεις.

Όπως φαίνεται στο Σχήμα 2, τα φύλλα μπορούν επιπλέον να περιέχουν δείκτες στις τιμές που αντιστοιχούν στα κλειδιά τους.



Σχήμα 2: *B+ δέντρο τάξης 4 [54]*

Η αποσύζευξη των κλειδιών από τα διαχωριστικά στοιχεία του ευρετηρίου απλοποιεί τη διαδικασία επανεξισορρόπησης μετά από διαγραφές, καθώς επιτρέπει το να υπάρχουν διαχωριστικά στοιχεία που δεν είναι κλειδιά στο τμήμα του δέντρου που λει-

τουργεί ως ευρετήριο. Μια διαφορά σε σχέση με το B-δέντρο, που αφορά τη διαδικασία εισαγωγής, είναι ότι όταν συμβαίνει υπερχείλιση και το φύλλο χωρίζεται στα δύο, αντί το μεσαίο κλειδί να προωθηθεί στον κόμβο γονέα, προωθείται ένα αντίγραφό του και το πρωτότυπο διατηρείται στο δεξί φύλλο που προέκυψε από τον διαχωρισμό. Επιπλέον, οι αναζητήσεις στο B+ δέντρο δεν σταματούν μόλις συναντήσουν το ζητούμενο στοιχείο αν αυτό βρίσκεται στο τμήμα του δέντρου που λειτουργεί ως ευρετήριο. Αντιθέτως, ακολουθείται ο δεξιός δείκτης και η αναζήτηση συνεχίζεται ώσπου να φτάσει σε κάποιο φύλλο.

Ο αριθμός προσβάσεων που απαιτούνται στη χειρότερη περίπτωση για την εισαγωγή, την αναζήτηση και τη διαγραφή ενός κλειδιού στο B+ δέντρο είναι $\mathcal{O}(\log_d n)$, όπως και στο B-δέντρο. Ωστόσο, όταν πρόκειται για *ακολουθιακές* αναζητήσεις το πλεονέκτημα του B+ δέντρου είναι ξεκάθαρο. Χάρη στη συνδεδεμένη λίστα που βρίσκεται στο κατώτερο επίπεδό του, η λειτουργία εύρεσης του επόμενου κλειδιού απαιτεί το πολύ 1 πρόσβαση σε κόμβο. Αυτό σημαίνει ότι η πρόσβαση σε όλα τα κλειδιά απαιτεί $\mathcal{O}(\log_d n + n)$ προσβάσεις σε κόμβους, ενώ στο B-δέντρο απαιτεί $\mathcal{O}(n \log_d n)$ [51].

Το **copy-on-write** B+ δέντρο, που είναι η δομή ευρετηρίου που χρησιμοποιεί η BoltBD, αποφεύγει τις τυχαίες επιτόπιες (in-place) εγγραφές, αντικαθιστώντας τες με ακολουθιακές εγγραφές στο τέλος του αρχείου. Γενικά, τα copy-on-write B+ δέντρα δεν υποστηρίζουν συνδέσμους μεταξύ των φύλλων, αφού αν το έκαναν θα έπρεπε ολόκληρο το δέντρο να επανεγγράφεται μετά από κάθε ενημέρωση.

Θεωρούμε ότι κάθε κόμβος του δέντρου αντιστοιχεί σε μια σελίδα. Στο αρχείο της βάσης δεδομένων οι σελίδες αποθηκεύονται διαδοχικά. Όταν ενημερώνεται μια τιμή σε μια σελίδα του δέντρου, αντί να γίνει επιτόπια ενημέρωση της σελίδας, μια νέα σελίδα με τα περιεχόμενα της παλιάς, καθώς και την ενημερωμένη τιμή, προσαρτάται στο τέλος του αρχείου. Επειδή η τοποθεσία της σελίδας άλλαξε, θα πρέπει να ενημερωθεί και η σελίδα γονέας ώστε να δείχνει στη σωστή τοποθεσία. Έτσι, αυτή η διαδικασία επαναλαμβάνεται μέχρι τη ρίζα. Οι παλιές σελίδες δεν διαγράφονται αμέσως, επιτρέποντας στις λειτουργίες ανάγνωσης που έχουν πρόσβαση στην παλιά ρίζα να "βλέπουν" ένα συνεπές στιγμιότυπο του δέντρου. Συμπερασματικά, το copy-on-write B+ δέντρο αποφεύγει τις τυχαίες εγγραφές αλλά εισάγει σημαντική ενίσχυση εγγραφών και χώρου (write and space amplification)[6] [55].

[6]Ο ορισμός της ενίσχυσης αναγνώσεων, εγγραφών και χώρου δίνεται στην υποενότητα 2.3.2.

2.2.2 Βασικές έννοιες & API

Οι βασικές έννοιες της BoltDB, καθώς και οι λειτουργίες που προσφέρονται από το API της συνοψίζονται στη λίστα που ακολουθεί [58]:

- **DB:** στην BoltDB το ανώτερου επιπέδου αντικείμενο είναι μια `DB` (βάση δεδομένων), που αντιπροσωπεύει ένα αρχείο αντιστοίχισης μνήμης αποθηκευμένο στο δίσκο. Παραδείγματα συναρτήσεων που μπορούν να εφαρμοστούν πάνω στην `DB` αποτελούν οι `DB.Open()` και `DB.Close()`.

- **Συναλλαγές:** μόνο μία συναλλαγή *ανάγνωσης και εγγραφής* (read-write) επιτρέπεται να είναι ενεργή κάθε στιγμή, ενώ δεν υπάρχει περιορισμός στον αριθμό των ενεργών συναλλαγών *μόνο ανάγνωσης* (read-only). Κάθε συναλλαγή έχει πρόσβαση σε μια *συνεπή όψη* της βάσης δεδομένων όπως αυτή ήταν όταν η συναλλαγή ξεκίνησε. Μια συναλλαγή μπορεί να δημιουργηθεί (`DB.Begin()`), να κατοχυρωθεί (`Tx.Commit()`) ή να αναιρεθεί (`Tx.Rollback()`). Η BoltDB διαθέτει επίσης τις συναρτήσεις `DB.Update()` και `DB.View()` που αποκρύπτουν τις λεπτομέρειες διαχείρισης των συναλλαγών.

- **Κάδοι:** ο χώρος αποθήκευσης στην BoltDB είναι χωρισμένος σε κάδους (buckets). Οι κάδοι είναι συλλογές ζευγών κλειδιού-τιμής εντός των οποίων κάθε κλειδί πρέπει να είναι μοναδικό. Κατά μια έννοια, οι κάδοι αντιπροσωπεύουν διακριτούς χώρους ονομάτων (namespaces). Ένας κάδος μπορεί να δημιουργηθεί (`Tx.CreateBucket()`), να διαγραφεί (`Tx.DeleteBucket()`) ή να επιστραφεί στον χρήστη στο πλαίσιο μιας συναλλαγής (`Tx.Bucket()`), με σκοπό στη συνέχεια να τοποθετηθούν ή να επιστραφούν ζεύγη κλειδιού-τιμής σε/από αυτόν.

- **Ζεύγη κλειδιού-τιμής:** η τοποθέτηση ενός ζεύγους κλειδιού-τιμής σε έναν κάδο πραγματοποιείται με τη συνάρτηση `Bucket.Put()`. Παρομοίως, η επιστροφή ενός ζεύγους κλειδιού-τιμής γίνεται με κλήση της συνάρτησης `Bucket.Get()`, ενώ η διαγραφή με κλήση της `Bucket.Delete()`.

- **Δρομείς:** ένας δρομέας (cursor) χρησιμοποιείται για να διατρέξει τα κλειδιά που είναι αποθηκευμένα σε έναν κάδο. Δημιουργείται με την συνάρτηση `Bucket.Cursor()` και κινείται στο χώρο των κλειδιών με τις συναρτήσεις `Cursor.`-

`First()`, `Cursor.Last()`, `Cursor.Seek()`,`Cursor.Next()` και `Cursor.-Prev()`. Επιπλέον, η συνάρτηση `Bucket.ForEach()` επιτρέπει την εκτέλεση μιας καθορισμένης από τον χρήστη συνάρτησης για κάθε ζεύγος κλειδιού-τιμής σε έναν κάδο.

- **Αντίγραφα ασφαλείας:** το γεγονός ότι η βάση δεδομένων στην BoltDB είναι ένα μεμονωμένο αρχείο καθιστά εύκολη τη δημιουργία αντιγράφων ασφαλείας. Η συνάρτηση `Tx.WriteTo()` μπορεί να κληθεί στο πλαίσιο μιας συναλλαγής μόνο ανάγνωσης και να γράψει μια συνεπή όψη της βάσης δεδομένων σε έναν εγγραφέα (π.χ. σε ένα αρχείο ή σε έναν σωλήνα της Go).

2.2.3 Μειονεκτήματα & περιορισμοί

- Η χρήση της BoltDB ενδείκνυται για φόρτο εργασίας με έμφαση στις αναγνώσεις. Η επίδοση των ακολουθιακών εγγραφών είναι ικανοποιητική, αλλά οι *τυχαίες εγγραφές* τείνουν να είναι αργές, ειδικά καθώς το μέγεθος της βάσης δεδομένων αυξάνεται [58].

- Το λειτουργικό σύστημα διατηρεί στη μνήμη όσο περισσότερο από το αρχείο αντιστοίχισης μνήμης της BoltDB είναι δυνατό. Συνεπώς, η BoltDB εμφανίζει *υψηλή κατανάλωση μνήμης* όταν η βάση δεδομένων είναι μεγάλη [58].

- Στην BoltDB εμφανίζεται τόσο το φαινόμενο του εξωτερικού κατακερματισμού (fragmentation), όσο κι ένα είδος εσωτερικού κατακερματισμού. Σχετικά με το δεύτερο, η BoltDB κάθε φορά που ο διαθέσιμος χώρος του αρχείου της εξαντλείται, διπλασιάζει το μέγεθος του. Από τη στιγμή που το αρχείο φτάνει το 1GB κι ύστερα, το μέγεθός του αυξάνεται σε βήματα του 1GB. Έτσι, η βάση δεδομένων δεσμεύει χώρο στον δίσκο τον οποίο στην πραγματικότητα δεν χρησιμοποιεί [60].

  Ο εξωτερικός κατακερματισμός συμβαίνει κατά τη διαγραφή δεδομένων από την BoltDB. Επειδή οι σελίδες που φιλοξενούν αυτά τα δεδομένα μπορεί να βρίσκονται οπουδήποτε μέσα στο αρχείο, όταν αυτές απομακρύνονται δεν είναι δυνατή η περικοπή (truncation) του αρχείου. Παρόλο που ο χώρος που καταλαμβάνουν οι σελίδες αυτές δεν αποδεσμεύεται, η BoltDB διατηρεί μια λίστα με

τις ελεύθερες σελίδες, ώστε να μπορεί να τις επαναχρησιμοποιήσει. Η *ανασυγκρότηση* (defragmentation) μπορεί να επιτευχθεί μόνο με την αντιγραφή ολόκληρης της βάσης δεδομένων σε ένα νέο αρχείο [58].

2.3 Ο Αποθηκευτικός Μηχανισμός RocksDB

Η RocksDB [69] είναι μια βιβλιοθήκη C++, που παρέχει ένα ενσωματωμένο σύστημα αποθήκευσης κλειδιού-τιμής. Η δομή ευρετηρίου που χρησιμοποιεί είναι το LSM-δέντρο (log-structured merge-tree / δέντρο συγχώνευσης με δομή αρχείου καταγραφής).

2.3.1 LSM-δέντρα

Τα βασικά δομικά στοιχεία ενός LSM-δέντρου είναι ο *πίνακας μνήμης* (memtable) και τα *αρχεία SST* (sorted string table files / αρχεία ταξινομημένου πίνακα συμβολοσειρών). Ένα αρχείο SST περιέχει ένα αυθαίρετο σύνολο ταξινομημένων ζευγών κλειδιού-τιμής. Τα αρχεία SST αποθηκεύονται *στον δίσκο* και μπορεί να περιέχουν διπλότυπα κλειδιά. Μια από τις σημαντικότερες ιδιότητες των αρχείων SST είναι ότι παραμένουν *αμετάβλητα*. Συχνά υλοποιούνται ως B-δέντρα.

Ο πίνακας μνήμης είναι μια δενδροειδής δομή που διατηρείται *στη μνήμη*. Η απλούστερη μορφή του LSM-δέντρου έχει μόνο δύο επίπεδα: τον πίνακα μνήμης κι ένα επίπεδο με SST αρχεία στον δίσκο [61]. Ωστόσο, τα περισσότερα LSM-δέντρα που χρησιμοποιούνται στην πράξη διαθέτουν πολλαπλά επίπεδα αυξανόμενου μεγέθους στον δίσκο. Με αυτόν τον τρόπο, μειώνεται ο αριθμός αρχείων ανά επίπεδο, καθιστώντας πιο αποδοτικές τις αναγνώσεις αλλά και τις συγχωνεύσεις από το ένα επίπεδο στο επόμενο. Στη συνέχεια, θα εξετάσουμε τις βασικές λειτουργίες του LSM-δέντρου.

Εισαγωγή: όλες οι εγγραφές κατευθύνονται στον πίνακα μνήμης, γι᾽ αυτό και είναι εξαιρετικά ταχείες.

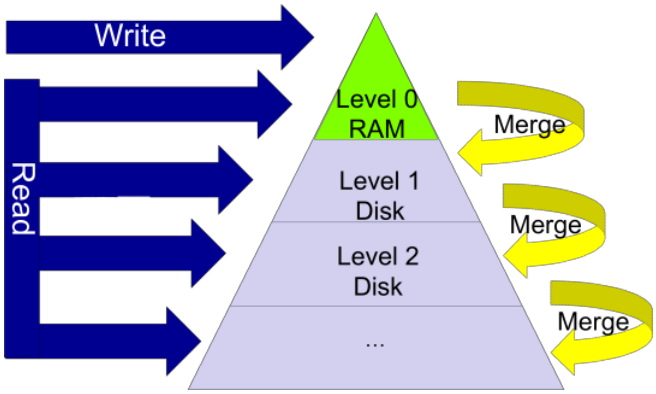
Αναζήτηση: όταν πραγματοποιείται ένα αίτημα ανάγνωσης, ελέγχεται πρώτα ο πίνακας μνήμης. Αν το ζητούμενο κλειδί δεν βρεθεί εκεί, τα αρχεία SST στον δίσκο θα ελεγχθούν ένα προς ένα με αντίστροφη χρονολογική σειρά δημιουργίας, ώσπου να βρεθεί το κλειδί [7]. Στη χειρότερη περίπτωση μπορεί να χρειαστεί να ελεγχθούν όλα τα επίπεδα του δέντρου πριν να βρεθεί το κλειδί ή να αποφασιστεί ότι δεν υπάρχει. Η

εξυπηρέτηση αιτημάτων που αφορούν ένα εύρος κλειδιών είναι αρκετά χρονοβόρα, καθώς τα κλειδιά που ανήκουν στο εύρος αυτό μπορεί να είναι διασκορπισμένα σε πολλαπλά επίπεδα του δέντρου, με αποτέλεσμα να απαιτούνται πολλές προσβάσεις σε αρχεία.

Ενημερώσεις και διαγραφές: οι ενημερώσεις δεν είναι επιτόπιες, αφού όπως έχει αναφερθεί τα αρχεία SST είναι αμετάβλητα. Αντιθέτως, εισάγονται στον πίνακα μνήμης. Επίσης, όταν συμβαίνει μια διαγραφή, ένα σημάδι διαγραφής (tombstone) αποθηκεύεται στον πίνακα μνήμης. Επειδή ελέγχεται πρώτα ο πίνακας μνήμης, εξασφαλίζεται ότι τα αιτήματα ανάγνωσης λαμβάνουν την πιο πρόσφατη έκδοση ενός κλειδιού [62].

Μόλις το μέγεθος του πίνακα μνήμης φτάσει μια προκαθορισμένη τιμή, ο πίνακας μεταφέρεται στον δίσκο με τη μορφή ενός νέου αμετάβλητου αρχείου SST, ενώ ένας νέος πίνακας μνήμης παίρνει τη θέση του. Η ομαδοποίηση των εγγραφών που συμβαίνει με αυτόν τον τρόπο στο LSM-δέντρο εξασφαλίζει τον διαμοιρασμό του κόστους Ε/Ε.

Όταν ο αριθμός ή το συνολικό μέγεθος των αρχείων σε ένα επίπεδο του LSM-δέντρου ξεπεράσει ένα προκαθορισμένο κατώφλι, πραγματοποιείται μια διαδικασία *συμπύκνωσης* (compaction). Κατά τη συμπύκνωση, τα αρχεία SST *συγχωνεύονται* δημιουργώντας νέα αρχεία τα οποία αποθηκεύονται στο επόμενο επίπεδο του δέντρου. Επιπλέον, κατά τη διάρκεια αυτής της διαδικασίας γίνεται απαλοιφή των διπλοτύπων, δηλαδή διατηρούνται μόνο οι πιο πρόσφατες ενημερώσεις και τα σημάδια διαγραφής ενώ τα παλαιότερα δεδομένα απορρίπτονται.



Σχήμα 3: *Η δομή του LSM-δέντρου [63]*

Μια σημαντική ιδιότητα του LSM-δέντρου που προκύπτει από την παραπάνω περιγραφή, είναι ότι *μετατρέπει* τις τυχαίες εγγραφές σε ακολουθιακές. Συχνά, το LSM-δέντρο συνοδεύεται από ένα αρχείο προεγγραφής (write-ahead log / WAL), προκει-

μένου να διασφαλιστεί η μονιμότητα (durability). Έτσι, σε περίπτωση απώλειας ρεύματος, τα δεδομένα που βρίσκονται στον πίνακα μνήμης δεν χάνονται.

2.3.2 Ενίσχυση αναγνώσεων, εγγραφών και χώρου

Η ενίσχυση αναγνώσεων (read amplification), η ενίσχυση εγγραφών (write amplification) και η ενίσχυση χώρου (space amplification) είναι έννοιες ύψιστης σημασίας για τις σχεδιαστικές αποφάσεις που πρέπει να ληφθούν κατά την υλοποίηση ενός LSM-δέντρου. Η *ενίσχυση εγγραφών* είναι ο λόγος των bytes που γράφονται στη συσκευή αποθήκευσης προς τα bytes που γράφονται στη βάση δεδομένων. Η υψηλή ενίσχυση εγγραφών είναι ανεπιθύμητη, όχι μόνο επειδή βλάπτει την επίδοση των εγγραφών, αλλά και γιατί είναι επιζήμια για τους δίσκους στερεάς κατάστασης. Η *ενίσχυση αναγνώσεων* αναφέρεται στον αριθμό αναγνώσεων από τον δίσκο ανά αίτημα ανάγνωσης. Τέλος, η *ενίσχυση χώρου* περιγράφει πόσο επιπλέον χώρο στον δίσκο καταλαμβάνει μια βάση δεδομένων σε σύγκριση με τον όγκο των δεδομένων που φυλάσσονται σε αυτήν. Ένα μέσο για τη μείωση της ενίσχυσης χώρου είναι η συμπίεση (compression) [64].

2.3.3 Τύποι συμπύκνωσης

Όταν χρησιμοποιείται ο *καθολικός τύπος συμπύκνωσης* (universal compaction), τα αρχεία SST είναι διατεταγμένα με βάση τη *χρονολογική σειρά* τους. Κάθε ένα από αυτά καλύπτει ολόκληρο τον χώρο κλειδιών και περιέχει τις ενημερώσεις ενός συγκεκριμένου χρονικού διαστήματος. Ανάμεσα στα αρχεία SST δεν υπάρχει χρονική επικάλυψη, υπάρχει όμως επικάλυψη στο πεδίο κλειδιών. Η συμπύκνωση πραγματοποιείται μεταξύ δύο ή περισσότερων γειτονικών αρχείων SST και η έξοδός της είναι ένα αρχείο SST το οποίο αποθηκεύεται στο επόμενο επίπεδο του LSM-δέντρου και του οποίου το χρονικό διάστημα προκύπτει από τον συνδυασμό των διαστημάτων των αρχείων εισόδου [67].

Κατά τη χρήση του *τύπου συμπύκνωσης με επίπεδα* (levelled compaction), τα πιο πρόσφατα δεδομένα διατηρούνται στο πρώτο επίπεδο ενώ τα παλαιότερα στο τελευταίο. Κάθε επίπεδο εκτός από το πρώτο, το οποίο είναι δομημένο όπως στην καθολική συμπύκνωση, έχει ένα κατώφλι μεγέθους και στο εσωτερικό του δεν υπάρχουν επικαλύψεις στο πεδίο των κλειδιών. Με άλλα λόγια, το διάστημα των κλειδιών διαμοιρά-

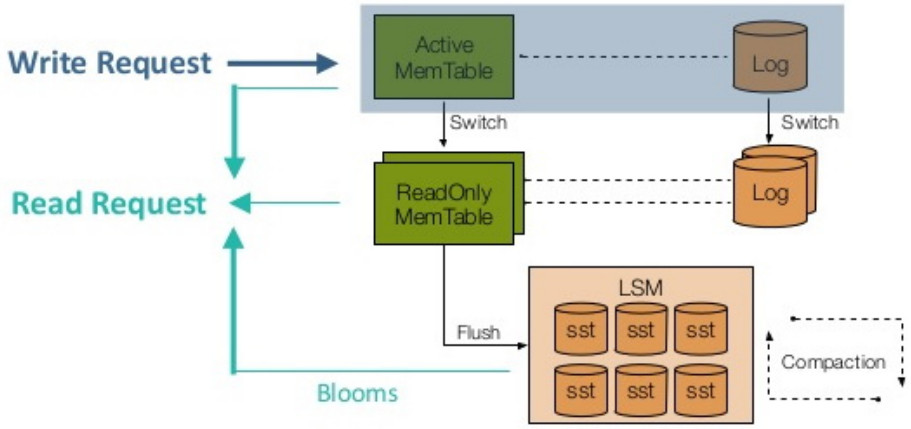
ζεται στα αρχεία SST ενός επιπέδου κι έτσι ολόκληρο το επίπεδο είναι ταξινομημένο με βάση τα κλειδιά. Στην περίπτωση του τύπου συμπύκνωσης με επίπεδα, όλα τα αρχεία SST έχουν το ίδιο μέγεθος. Όταν το μέγεθος ενός επιπέδου ξεπεράσει το κατώφλι, επιλέγεται ένα αρχείο SST και συγχωνεύεται στο επόμενο επίπεδο. Συνήθως, κάθε επίπεδο είναι 10 φορές μεγαλύτερο από το προηγούμενο [68]. Ο συγκριτικός Πίνακας 1 παρουσιάζει την ενίσχυση αναγνώσεων και εγγραφών στους δύο τύπους συμπύκνωσης.

	Καθολική Συμπύκνωση	**Συμπύκνωση με Επίπεδα**
Ενίσχυση αναγνώσεων (χειρότερη περίπτωση)	Μεγαλύτερη. Πρόσβαση σε κάθε αρχείο SST σε όλα τα επίπεδα	Ίση με $number\_of\_L0\_files + (n-1)$, όπου n ο αριθμός των επιπέδων στον δίσκο
Ενίσχυση εγγραφών	Κάθε ενημέρωση θα γραφτεί το πολύ n φορές	Μεγαλύτερη. Σε κάθε συμπύκνωση αντί να γράφεται μόνο το αρχείο που συγχωνεύεται από το L_n στο L_{n+1}, επανεγγράφονται και τα αρχεία του L_{n+1} με τα οποία υπάρχει επικάλυψη

Πίνακας 1: *Ενίσχυση αναγνώσεων & εγγραφών στους δύο τύπους συμπύκνωσης*

2.3.4 Βασικές έννοιες & API

Ακολουθώντας τον σχεδιασμό του LSM-δέντρου, η RocksDB έχει ως βασικές της δομές τον πίνακα μνήμης, τα αρχεία SST και το αρχείο προεγγραφών. Μόλις ο πίνακας μνήμης γεμίσει και μεταφερθεί ως αρχείο SST στη συσκευή μόνιμης αποθήκευσης, το αρχείο προεγγραφών που του αντιστοιχεί μπορεί να διαγραφεί με ασφάλεια. Η RocksDB μπορεί επίσης να ρυθμιστεί ώστε να υποστηρίζει την ύπαρξη πολλαπλών πινάκων μνήμης. Όταν ένας από αυτούς γεμίσει, γίνεται αμετάβλητος και ένα νήμα στο παρασκήνιο αναλαμβάνει να μεταφέρει τα περιεχόμενά του στον δίσκο. Παράλληλα, οι νέες εγγραφές κατευθύνονται σε έναν νέο πίνακα μνήμης, αντί να τίθενται σε αναμονή μέχρι να ολοκληρωθεί η μεταφορά [69].



Σχήμα 4: *Η αρχιτεκτονική της RocksDB [73]*

Η ακόλουθη λίστα συνοψίζει μερικές βασικές έννοιες και λειτουργίες της RocksDB [69]:

- **DB:** το όνομα που δίνεται στη βάση δεδομένων αντιστοιχεί σε έναν κατάλογο στο σύστημα αρχείων. Οι συναρτήσεις που μπορούν να κληθούν πάνω στην `DB` συμπεριλαμβάνουν την `Open()` και την `DestroyDB()`.

- **Key-value pairs:** τα κλειδιά και οι τιμές στην RocksDB έχουν τη μορφή πινάκων από bytes. Η συνάρτηση `Get()` επιτρέπει την επιστροφή ενός ζεύγους κλειδιού-τιμής από τη βάση δεδομένων. Αντίστοιχα, η συνάρτηση `Put()` εισάγει και η συνάρτηση `Delete()` διαγράφει ένα ζεύγος κλειδιού-τιμής. Η συνάρτηση `Write()` επιτρέπει την *ατομική* εισαγωγή, ενημέρωση ή διαγραφή πολλαπλών ζευγών κλειδιού-τιμής ταυτόχρονα. Το σύνολο αυτών των ζευγών κλειδιού-τιμής καλείται `WriteBatch`. Με άλλα λόγια, το `WriteBatch` περιέχει μια ακολουθία ενημερώσεων που πρόκειται να γίνουν στη βάση δεδομένων και τις εφαρμόζει με τη σειρά όταν καλείται η `Write()`. Συνήθως, η χρήση του `WriteBatch` για την ενημέρωση πολλαπλών ζευγών κλειδιού-τιμής είναι πιο αποδοτική από τη χρήση της συνάρτησης `Put()` για κάθε ένα από αυτά, καθώς το κόστος της *σύγχρονης* εγγραφής στο αρχείο προεγγραφών διαμοιράζεται μεταξύ των ενημερώσεων.

  Το `WriteBatchWithIndex` είναι μια παραλλαγή του `WriteBatch`, σχεδιασμένη για την εξυπηρέτηση της περίπτωσης στην οποία ένας αναγνώστης χρειάζε-

ται πρόσβαση στις μη κατοχυρωμένες εγγραφές ενός `WriteBatch`. Το `WriteBatchWithIndex` το καταφέρνει αυτό διατηρώντας έναν εσωτερικό απομονωτή (buffer) με μορφή ευρετηρίου, που περιέχει όλα τα κλειδιά που έχουν εγγραφεί. Τα αιτήματα ανάγνωσης από ένα `WriteBatchWithIndex` *συνδυάζουν* τα περιεχόμενα του απομονωτή με τα περιεχόμενα της βάσης δεδομένων και επιστρέφουν τα πιο πρόσφατα αποτελέσματα [74].

- **Επαναλήπτης:** ο επαναλήπτης (`Iterator`) χρησιμοποιείται για την εξυπηρέτηση ερωτημάτων που αφορούν ένα εύρος κλειδιών (range queries). Έχει τη δυνατότητα να εντοπίζει ένα συγκεκριμένο κλειδί και στη συνέχεια, ξεκινώντας από εκείνο το σημείο, να σαρώνει ένα κλειδί τη φορά.

- **Δομές επιλογών:** η RocksDB χρησιμοποιεί δομές επιλογών για τη ρύθμιση των παραμέτρων της σε μια πληθώρα περιπτώσεων, μεταξύ των οποίων βρίσκεται το άνοιγμα της βάσης δεδομένων, η κάθε ανάγνωση, η κάθε εγγραφή και η δημιουργία μιας συναλλαγής.

- **Στιγμιότυπο:** το στιγμιότυπο (`Snapshot`) επιτρέπει τη δημιουργία μιας *συνεπούς όψης* της βάσης δεδομένων. Η συνάρτηση `Get()` και ο `Iterator` μπορούν να ρυθμιστούν έτσι ώστε να διαβάζουν δεδομένα από ένα `Snapshot`, ορισμένο στην δομή επιλογών τους.

- **Σημείο ελέγχου:** το σημείο ελέγχου (`Checkpoint`) είναι ένα αντίγραφο της βάσης δεδομένων, που βρίσκεται σε διαφορετικό κατάλογο στο ίδιο σύστημα αρχείων και έχει τη δυνατότητα να ανοιχθεί ως ξεχωριστή βάση δεδομένων. Η ιδιαιτερότητά του έγκειται στο ότι περιέχει *hard-links* στα αρχεία SST της πρωτότυπης βάσης δεδομένων, αντί για κανονικά αντίγραφά τους, γεγονός που καθιστά τη δημιουργία του εξαιρετικά αποδοτική [76].

- **Συναλλαγές:** οι συναλλαγές στη RocksDB εξασφαλίζουν ότι μια ομάδα εγγραφών θα πραγματοποιηθεί μόνο αν δεν υπάρχουν διενέξεις. Η διαχείριση των συναλλαγών γίνεται με τις συναρτήσεις `Begin()`, `Commit()` και `Rollback()`. Υποστηρίζουν επίσης την ανάγνωση των ζευγών κλειδιού τιμής που έχουν εισαχθεί σε αυτές αλλά δεν έχουν ακόμη κατοχυρωθεί [77].

  Η RocksDB διαθέτει τόσο *αισιόδοξες* (optimistic) όσο και *απαισιόδοξες* συναλλαγές (pessimistic transactions). Όταν χρησιμοποιούνται απαισιόδοξες συναλ-

λαγές, κάθε φορά που γράφεται ένα κλειδί στη συναλλαγή, αυτό κλειδώνεται εσωτερικά, ώστε να γίνει ανίχνευση διενέξεων. Αν ένα κλειδί δεν μπορεί να κλειδωθεί επιστρέφεται μήνυμα λάθους. Οι αισιόδοξες συναλλαγές πραγματοποιούν την ανίχνευση διενέξεων κατά την κατοχύρωσή τους για να επικυρώσουν ότι κανένας άλλος εγγραφέας δεν έχει τροποποιήσει τα κλειδιά που γράφονται από αυτές. Αν εντοπιστεί διένεξη, η συνάρτηση κατοχύρωσης `Commit()` επιστρέφει μήνυμα λάθους και η συναλλαγή αναιρείται. Οι απαισιόδοξες συναλλαγές είναι προτιμότερες σε περιβάλλοντα όπου η *συχνότητα* των διενέξεων είναι μεγάλη, οπότε το κόστος των κλειδωμάτων για την προστασία των δεδομένων είναι μικρότερο από το κόστος της διαρκούς αναίρεσης κι επανεκκίνησης των συναλλαγών. Αντιθέτως, οι αισιόδοξες συναλλαγές προτιμούνται σε περιβάλλοντα όπου οι διενέξεις είναι σπάνιες.

- **Κρυφή μνήμη για blocks:** η RocksDB χρησιμοποιεί μια κρυφή μνήμη τύπου LRU (least recently used / λιγότερο πρόσφατα χρησιμοποιημένα) που φιλοξενεί τα blocks των αρχείων SST με τη μεγαλύτερη ζήτηση για την αποδοτικότερη εξυπηρέτηση των αιτημάτων ανάγνωσης.

2.3.5 Σύγκριση με την BoltDB

1. Όπως είδαμε στις αντίστοιχες ενότητες, το LSM-δέντρο είναι δομή βελτιστοποιημένη για εγγραφές ενώ το B+ δέντρο για αναγνώσεις. Συνεπώς, η χρήση της BoltDB ενδείκνυται όταν ο φόρτος εργασίας χαρακτηρίζεται από συχνές αναγνώσεις ενώ η RocksDB προτιμάται σε περιπτώσεις χρήσης με έμφαση στις εγγραφές.

2. Η διεκπεραιωτική ικανότητα εγγραφών του copy-on-write B+ δέντρου είναι μεγαλύτερη από αυτή του απλού B+ δέντρου, λόγω του ότι αποφεύγει τις τυχαίες εγγραφές. Ωστόσο, η ενίσχυση εγγραφών σε αυτό είναι σημαντικά μεγαλύτερη από ό,τι στο LSM-δέντρο, επειδή κάθε ενημέρωση προκαλεί την επανεγγραφή ενός μέρους της δομής.

3. Η επίδοση των B+ δέντρων, και κατ' επέκταση της BoltDB, μειώνεται δραματικά όταν το σύνολο των δεδομένων ξεπερνά σε μέγεθος τη διαθέσιμη μνήμη του συστήματος, καθώς αυξάνεται η πιθανότητα οι σελίδες που απαιτούνται για την

εξυπηρέτηση ενός αιτήματος να μην βρίσκονται στη μνήμη και να χρειαστεί να διαβαστούν από τον δίσκο.

4. Τα B+ δέντρα χειρίζονται τις τιμές μεγάλου μεγέθους πιο αποδοτικά. Στα LSM-δέντρα η εισαγωγή μεγάλων τιμών μπορεί να πυροδοτήσει διαδοχικές συμπυκνώσεις οι οποίες προκαλούν επιπρόσθετη καθυστέρηση.

5. Η RocksDB αξιοποιεί πιο αποδοτικά τον χώρο στον δίσκο, καθώς δεν εμφανίζει το φαινόμενο του κατακερματισμού και εφαρμόζει συμπίεση των δεδομένων.

6. Τόσο η BoltDB όσο και η RocksDB έχουν υψηλές απαιτήσεις σε μνήμη λόγω των δομών δεδομένων στις οποίες βασίζονται.

7. Ο χρόνος επανεκκίνησης της RocksDB κατά την ανάνηψη από σφάλμα είναι μεγαλύτερος αφού χρειάζεται να διαβάσει το αρχείο προεγγραφών για να επανακατασκευάσει τον πίνακα μνήμης και να εντοπίσει την τελευταία επιτυχή ενημέρωση.

8. Τέλος, η RocksDB είναι σύστημα πολύ πιο περίπλοκο από την BoltDB, με πληθώρα παραμέτρων που απαιτούν διεξοδική μελέτη προκειμένου να ρυθμιστούν επιτυχώς.

3 Σχεδιασμός

Όπως έχει ήδη αναφερθεί, ο etcd είναι σχεδιασμένος για την αξιόπιστη αποθήκευση μεταδεδομένων, στα οποία οι ενημερώσεις δεν είναι συχνές. Ο πρωτεύων στόχος μας είναι να βελτιώσουμε τη διεκπεραιωτική ικανότητα εγγραφών του, καθώς θεωρούμε ότι έχει τη δυνατότητα να χρησιμοποιηθεί ως σύστημα αποθήκευσης κλειδιού-τιμής γενικού σκοπού. Σκοπεύουμε να το επιτύχουμε αυτό αντικαθιστώντας τον τρέχοντα, βελτιστοποιημένο για αναγνώσεις αποθηκευτικό μηχανισμό του, την BoltDB, με την βελτιστοποιημένη για εγγραφές RocksDB. Οι προσδοκίες μας από το τελικό σύστημα, σύμφωνα με την ανάλυση που έγινε στο κεφάλαιο 2 είναι οι εξής:

- Βελτίωση της επίδοσης των εγγραφών
- Διατήρηση των εγγυήσεων αξιοπιστίας, συνέπειας και υψηλής διαθεσιμότητας του etcd
- Ελαφρά επιδείνωση της επίδοσης των αναγνώσεων

3.1 Αρχιτεκτονική & Σχεδιαστικές Επιλογές

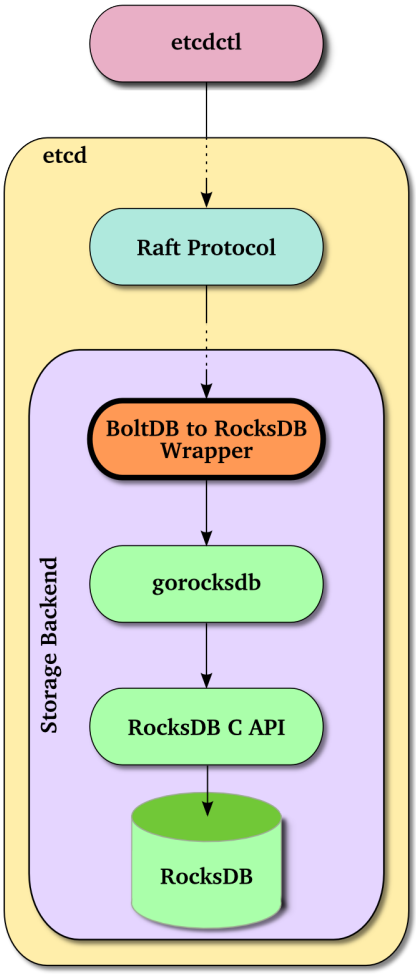
3.1.1 Ενσωμάτωση της RocksDB στον etcd

Κάθε αίτημα εγγραφής στον etcd περνά πρώτα από το πρωτόκολλο Raft και στη συνέχεια προωθείται στο πακέτο `backend` του etcd, το οποίο το κατευθύνει στον μηχανισμό αποθήκευσης. Στο πλαίσιο αυτής της εργασίας, υλοποιούμε μια *βιβλιοθήκη περιτύλιξης* (wrapper library) σε γλώσσα προγραμματισμού Go, σκοπός της οποίας είναι η *αντιστοίχιση* των κλήσεων που πραγματοποιεί το backend του etcd στην BoltDB στις ανάλογες κλήσεις συναρτήσεων της βιβλιοθήκης RocksDB. Η προσέγγιση αυτή μας επιτρέπει να ελαχιστοποιήσουμε τις παρεμβάσεις στον κώδικα του ίδιου του etcd.

Τοποθετούμε τη βιβλιοθήκη περιτύλιξής μας στο πακέτο `bolt`, όπου βρισκόταν προηγουμένως η βιβλιοθήκη BoltDB. Με αυτόν τον τρόπο η αλλαγή του μηχανισμού αποθήκευσης δεν είναι ορατή στα υπόλοιπα πακέτα του etcd. Διατηρούμε στη βιβλιοθήκη όλες τις συναρτήσεις του API της BoltDB που χρησιμοποιούνται από τον etcd, όμως αλλάζουμε τον κώδικά τους ώστε να καλούν τις αντίστοιχες συναρτήσεις της

RocksDB. Τέλος, προσθέτουμε τις απαραίτητες βοηθητικές συναρτήσεις στα σημεία όπου η λειτουργικότητα των δύο αποθηκευτικών μηχανισμών αποκλίνει. Αργότερα, θα είναι πολύ απλό να τροποποιήσουμε τον etcd ώστε να υποστηρίζει και τους δύο μηχανισμούς αποθήκευσης. Αρκεί να προσθέσουμε τη σχετική επιλογή στη φάση της μεταγλώττισης ή της εκκίνησης.

Το Σχήμα 5 αναπαριστά την αρχιτεκτονική του backend του etcd μετά την ενσωμάτωση της RocksDB. Τα δομικά στοιχεία του backend περιγράφονται στις παραγράφους που ακολουθούν.



Σχήμα 5: *Αρχιτεκτονική του etcd backend με τη RocksDB ως αποθηκευτικό μηχανισμό*

3.1.2 Γιατί RocksDB;

Ο κυρίαρχος λόγος για την επιλογή της RocksDB είναι η βελτιστοποιημένη για εγγραφές δομή ευρετηρίου που υλοποιεί, το LSM-δέντρο. Σύμφωνα με τον προγραμματιστή της BoltDB, Ben Johnson, η RocksDB είναι η καλύτερη επιλογή στις περιπτώσεις που απαιτείται υψηλή διεκπεραιωτική ικανότητα εγγραφών (>10.000 εγγραφές/sec) [58]. Επιπλέον, η RocksDB είναι μια υψηλής επίδοσης βάση δεδομένων κλειδιού-τιμής και το γεγονός ότι υποστηρίζεται από μια έμπειρη κοινότητα προγραμματιστών, καθώς και το ότι χρησιμοποιείται ήδη στην παραγωγή από αρκετά συστήματα λογισμικού την καθιστούν εύρωστη και σταθερή.

Ένας ακόμη λόγος που μας οδήγησε στην επιλογή της RocksDB ανάμεσα σε άλλες βάσεις δεδομένων που βασίζονται στο LSM-δέντρο, ήταν ότι είναι η μόνη της οποίας τα χαρακτηριστικά ικανοποιούν όλες τις απαιτήσεις μας για ένα ανοιχτού κώδικα, τοπικό σύστημα αποθήκευσης κλειδιού-τιμής βασισμένο στο LSM-δέντρο, στη μορφή ενσωματωμένης βιβλιοθήκης. Τέλος, η ενσωμάτωση της RocksDB είναι μια ιδέα που και οι ίδιοι οι προγραμματιστές του etcd έχουν συζητήσει, καταλήγοντας όμως στο ότι η BoltDB καλύπτει ικανοποιητικά την τρέχουσα περίπτωση χρήσης και αφήνοντας ανοιχτό το ενδεχόμενο μελλοντικής διερεύνησης του ζητήματος [102].

3.1.3 Η βιβλιοθήκη περιτύλιξης gorocksdb

Καθώς ο κώδικας του etcd είναι γραμμένος σε Go, ενώ ο κώδικας της RocksDB σε C++, απαιτείται για την επικοινωνία τους ένα ενδιάμεσο σύστημα. Η RocksDB διαθέτει ήδη ένα API σε C (μια βιβλιοθήκη περιτύλιξης του C++ API της), οπότε αυτό που μας λείπει είναι μια βιβλιοθήκη περιτύλιξης του API αυτού σε Go. Η βιβλιοθήκη gorocksdb[7] εξυπηρετεί ακριβώς αυτόν τον σκοπό. Κάθε συνάρτηση αυτής της βιβλιοθήκης περιτύλιξης περιέχει μια κλήση στην αντίστοιχη συνάρτηση του C API της RocksDB. Επίσης, κάθε δομή (struct) στην gorocksdb περιέχει ένα πεδίο που φιλοξενεί έναν δείκτη της C στην αντίστοιχη δομή του C API της RocksDB.

[7] `https://github.com/tecbot/gorocksdb`

3.1.4 Το εργαλείο cgo

Ο μηχανισμός που επιτρέπει την κλήση συναρτήσεων της C από κώδικα γραμμένο σε Go είναι το εργαλείο *cgo*. Με τη χρήση του cgo αρκεί η φόρτωση ενός ψευδοπακέτου που καλείται `"C"` για να μπορεί στο εξής ο κώδικας Go να αναφέρεται σε τύπους της C όπως ο `C.size_t`, μεταβλητές όπως η `C.stdout` και συναρτήσεις όπως η `C.putchar`. Όταν η εντολή `go build` "βλέπει" ότι ένα ή περισσότερα αρχεία Go χρησιμοποιούν το ειδικό `import "C"`, ψάχνει για C/C++ αρχεία στον κατάλογο και καλεί τον C/C++ μεταγλωττιστή για να τα μεταγλωττίσει ως τμήμα του πακέτου Go. Αν αμέσως πριν το `import "C"` υπάρχει ένα σχόλιο, τότε αυτό το σχόλιο, που καλείται προοίμιο (preamble) και περιέχει κώδικα σε C, λειτουργεί ως επικεφαλίδα κατά τη μεταγλώττιση των αρχείων C του καταλόγου [93].

Παρόλο που το εργαλείο cgo είναι εξαιρετικά χρήσιμο, παρουσιάζει μια σειρά σημαντικών μειονεκτημάτων από τα οποία το πιο καθοριστικό για την υλοποίησή μας είναι ότι προκαλεί *επιδείνωση της επίδοσης*. Μία από τις αιτίες της καθυστέρησης που εισάγει η χρήση του cgo είναι ότι η C δεν γνωρίζει τίποτα για τον τρόπο κλήσης συναρτήσεων ή για τη *δυνατότητα επέκτασης της στοίβας*[8] στην Go. Επομένως, μια κλήση σε κώδικα C πρέπει να καταγράψει όλες τις λεπτομέρειες της στοίβας της goroutine[9] πριν να την *αντικαταστήσει*[10] με μια στοίβα της C [95], [96].

Ωστόσο, το μεγαλύτερο μέρος της καθυστέρησης που εισάγει το cgo οφείλεται στο γεγονός ότι κάθε κλήση συνάρτησης μέσω αυτού θεωρείται ότι μπλοκάρει (blocking) και αντιμετωπίζεται από το περιβάλλον χρόνου εκτέλεσης (runtime) της Go σαν κλήση συστήματος. Όταν μια goroutine πραγματοποιεί κλήση cgo, κλειδώνεται στο νήμα στο οποίο έτρεχε και αυτό το νήμα μπλοκάρει περιμένοντας την ολοκλήρωση της κλήσης. Αυτή η διαδικασία απαιτεί συντονισμό με τον runtime scheduler της Go και ενδέχεται να προκαλέσει τη δημιουργία ενός νέου νήματος ώστε να συνεχίσουν να τρέχουν οι υπόλοιπες goroutines [98], [99], [100]. Ακόμη μια πηγή καθυστέρησης είναι

[8] Η Go προκειμένου να εξοικονομήσει μνήμη αυξάνει το μέγεθος της στοίβας των goroutines σταδιακά και κατ' απαίτηση αντί να τους εκχωρεί από την αρχή μια στοίβα ικανού μεγέθους, όπως κάνει η C με τα νήματα. Ο κώδικας της C δεν θα γνωρίζει πώς να επεκτείνει τη στοίβα αν χρειαστεί περισσότερο χώρο από τα λιγοστά kilobytes που προσφέρει η στοίβα της Go.

[9] Η έννοια της goroutine στην Go είναι αντίστοιχη με αυτή του νήματος, με τη διαφορά ότι ο χρονικός προγραμματισμός των goroutines γίνεται από τον Go scheduler και όχι από το λειτουργικό σύστημα. Συνήθως πολλαπλές goroutines τρέχουν στο ίδιο νήμα.

[10] Η αντικατάσταση στοίβας (stack switch) συνίσταται στην αποθήκευση των καταχωρητών κατά την κλήση της συνάρτησης της C και στην επαναφορά τους κατά την επιστροφή της.

η δημιουργία αντιγράφων που συχνά είναι απαραίτητη κατά το πέρασμα δεδομένων από την Go στη C και αντίστροφα.

Η συνολική καθυστέρηση που εισάγει το cgo εκτιμάται ότι είναι από δέκα έως εκατό φορές μεγαλύτερη από την καθυστέρηση μιας απλής κλήσης στην Go. Συμπερασματικά, η χρήση του cgo συνίσταται μόνο όταν η διάρκεια εκτέλεσης της συνάρτησης της C που καλείται καθιστά την εισαγόμενη από το cgo καθυστέρηση αμελητέα ή όταν η επίδοση της καλούμενης συνάρτησης της C είναι σημαντικά καλύτερη από ό,τι θα ήταν στην Go.

3.1.5 Το C API της RocksDB

Το C API της RocksDB[11] είναι υλοποιημένο ως ένα ζεύγος αρχείου C++ και αρχείου επικεφαλίδας C. Οι δομές περιτύλιξης του C API περιέχουν ένα μόνο πεδίο, το οποίο φιλοξενεί έναν δείκτη στην αντίστοιχή τους C++ δομή. Παρομοίως, οι συναρτήσεις περιτύλιξης περιέχουν μια κλήση στην αντίστοιχή τους συνάρτηση C++.

3.1.6 Άρση του περιορισμού αποθηκευτικού χώρου

Ο etcd επιβάλλει ένα άνω φράγμα στο μέγεθος του backend του, το οποίο μπορεί να ρυθμιστεί από τα 2GB έως τα 8GB. Παρόλο που η χωρητικότητα αυτής της τάξης ίσως να είναι αρκετή για κάποιες από τις περιπτώσεις χρήσης του etcd ως σύστημα αποθήκευσης μεταδεδομένων, σίγουρα δεν επαρκεί στη γενική περίπτωση.

Σύμφωνα με τα έγγραφα τεκμηρίωσης του etcd, ο περιορισμός αυτός τίθεται εν μέρει για να αποτρέψει την *εξάντληση του χώρου στο δίσκο* και την *υποβάθμιση της επίδοσης* [41]. Η εξάντληση του χώρου στο δίσκο δε θα έπρεπε να μας ανησυχεί, αφού τα περισσότερα σύγχρονα υπολογιστικά συστήματα είναι εξοπλισμένα με συσκευές αποθήκευσης χωρητικότητας που ξεπερνά κατά πολύ τα 8GB. Επιπλέον, η υποβάθμιση της επίδοσης πιθανότατα αναφέρεται στην αδυναμία αποδοτικής διαχείρισης συνόλων δεδομένων μεγαλύτερων από τη διαθέσιμη μνήμη από πλευράς της BoltDB. Το πρόβλημα αυτό δεν υπάρχει στη δική μας υλοποίηση, όπου έχουμε αντικαταστήσει την BoltDB με τη RocksDB.

[11] `https://github.com/facebook/rocksdb/blob/master/db/c.cc`

Κατόπιν σχετικής συζήτησής μας με τον Xiang Li, έναν από τους βασικούς προγραμματιστές του etcd, στη σχετική με την ανάπτυξη του etcd λίστα ηλεκτρονικού ταχυδρομείου, μάθαμε πως ο κυριότερος λόγος ύπαρξης του περιορισμού είναι η διατήρηση του *μέσου χρόνου αποκατάστασης* (mean time to recovery / MTTR) εντός αποδεκτών ορίων. Στο σενάριο στο οποίο ένα μέλος της συστοιχίας υφίσταται βλάβη κι ένα νέο το αντικαθιστά, η αναμονή για λήψη ενός μεγάλου μεγέθους στιγμιοτύπου θα έβλαπτε τη διαθεσιμότητα του etcd [104]. Αυτό είναι ένα πρόβλημα που δεν μπορούμε να ξεπεράσουμε εύκολα, ωστόσο, αν για κάποια περίπτωση χρήσης η δυνατότητα διαχείρισης ενός μεγάλου dataset έχει μεγαλύτερη σημασία από το MTTR, μπορούμε να το παραβλέψουμε.

3.2 Αντιστοίχιση Εννοιών & Δομών

Σε αυτή την ενότητα εξετάζουμε τις αρχικές επιλογές που κάναμε κατά την αντιστοίχιση εννοιών και δομών της BoltDB στις ανάλογες δομές και έννοιες της RocksDB, στο πλαίσιο της βιβλιοθήκης περιτύλιξης που αναπτύξαμε.

3.2.1 DB

Προφανώς, μπορούμε να εφαρμόσουμε απευθείας αντιστοίχιση του αντικειμένου `DB` της BoltDB στο αντικείμενο `DB` της RocksDB. Παρόλο που τα δύο αυτά αντικείμενα υλοποιούνται με ουσιαστικά διαφορετικό τρόπο (το πρώτο ως αρχείο αντιστοίχισης μνήμης ενώ το δεύτερο ως κατάλογος), έχουν τις ίδιες λειτουργίες (π.χ. `Open()`, `Close()`) και σε μεγάλο βαθμό το API τους είναι κοινό.

3.2.2 Κάδοι με προθέματα

Ο ρόλος των κάδων στην BoltDB είναι να διαιρούν το σύνολο δεδομένων σε διαφορετικούς χώρους ονομάτων. Για να επιτύχουμε το ίδιο αποτέλεσμα στην υλοποίησή μας προσαρτούμε το όνομα του κάδου ως πρόθεμα στο ζητούμενο κλειδί πριν προωθήσουμε ένα αίτημα στη RocksDB. Επειδή οι συμβολοσειρές που αναπαριστούν τους κάδους στον etcd ποικίλλουν σε μήκος και κάποιες από αυτές ξεκινούν με την ίδια ακολουθία χαρακτήρων χρειαζόμαστε έναν οριοθέτη (delimiter) για να μπορούμε να διακρίνουμε το σημείο στο οποίο τελειώνει το πρόθεμα κι αρχίζει το όνομα του κλειδιού.

Για αυτόν τον σκοπό επιλέχθηκε ο χαρακτήρας "/". Για παράδειγμα, αν στο πλαίσιο ενός αιτήματος πρέπει το κλειδί `foo` να αποθηκευτεί στον κάδο `keys`, αποθηκεύουμε το κλειδί `keys/foo` στη RocksDB.

3.2.3 Λειτουργίες ανάγνωσης, εγγραφής και διαγραφής

Σε ό,τι αφορά τον πυρήνα του API, μπορεί να εφαρμοστεί ευθεία αντιστοίχιση μεταξύ των δύο μηχανισμών αποθήκευσης. Καθώς είναι και οι δύο συστήματα αποθήκευσης κλειδιού-τιμής, διαθέτουν συναρτήσεις `Get()`, `Put()` και `Delete()` με την ίδια σημασιολογία. Και στις δύο περιπτώσεις αυτές οι λειτουργίες θα εφαρμοστούν στο πλαίσιο συναλλαγών.

3.2.4 Δρομέας με επαναλήπτη

Η αντιστοίχιση του δρομέα (`Cursor`) της BoltDB σε κατάλληλη έννοια της RocksDB δεν παρουσιάζει δυσκολία. Ο επαναλήπτης (`Iterator`) της RocksDB υλοποιεί τις ίδιες βασικές λειτουργίες που χρειάζονται για να διατρέξει κανείς ένα σύνολο ζευγών κλειδιού-τιμής (π.χ. `First()`, `Seek()`, `Next()` κτλ.).

3.2.5 Συναλλαγές

Σε πρώτη φάση, αντιστοιχίζουμε τις συναλλαγές της BoltDB με τις απαισιόδοξες συναλλαγές της RocksDB. Στο κεφάλαιο 5 θα δούμε πώς αυτή η προσέγγιση οδήγησε σε υποβέλτιστα αποτελέσματα και στην ενότητα 4.3 θα εξετάσουμε περισσότερο αποδοτικές εναλλακτικές, εξηγώντας παράλληλα την ορθότητά τους.

Η έννοια της συναλλαγής στην BoltDB και στη RocksDB παρουσιάζει ορισμένες σημασιολογικές διαφορές που αφορούν κυρίως την πιθανότητα αποτυχίας, όπως προκύπτει από την περιγραφή της, στην υποενότητα 2.2.2 και στην υποενότητα 2.3.4 αντίστοιχα. Ωστόσο, στο πλαίσιο του etcd, όπου υπάρχει *μόνο μία* ενεργή συναλλαγή ανάγνωσης και εγγραφής κάθε στιγμή, δεν υπάρχουν διενέξεις, οπότε αυτές οι σημασιολογικές διαφορές δεν παίζουν κανένα ρόλο.

3.2.6 Στιγμιότυπο με σημείο ελέγχου

Όπως έχει αναφερθεί στην υποενότητα 2.1.4, ο etcd χρησιμοποιεί τα στιγμιότυπα ως αντίγραφα ασφαλείας για την αποκατάσταση καταστροφών, αλλά και για να βοηθήσει νέα μέλη της συστοιχίας να φτάσουν γρήγορα στο επίπεδο προόδου των ήδη υπαρχόντων μελών. Στην BoltDB η λήψη στιγμιοτύπου γίνεται με τη συνάρτηση `Tx.WriteTo()` η οποία γράφει μια συνεπή όψη της βάσης δεδομένων σε ένα αρχείο ή μια σωλήνωση, όπως έχουμε περιγράψει στην υποενότητα 2.2.2.

Υιοθετώντας την RocksDB ως αποθηκευτικό μηχανισμό του etcd, επιλέγουμε να αντιστοιχίσουμε την έννοια του στιγμιοτύπου (`Snapshot`) της BoltDB στην έννοια του σημείου ελέγχου (`Checkpoint`), καθώς αυτή κρίθηκε καταλληλότερη από άλλες παρεμφερείς έννοιες της RocksDB. Αξίζει να σημειωθεί ότι ο κώδικας του etcd αντιμετωπίζει το στιγμιότυπο ως αρχείο, ενώ μια βάση δεδομένων RocksDB και κατ᾽ επέκταση το `Checkpoint` της έχει τη μορφή καταλόγου. Για να επιλύσουμε αυτό το πρόβλημα αποφεύγοντας εκτεταμένη τροποποίηση του κώδικα του etcd μετατρέπουμε το `Checkpoint` σε αρχείο *tar*.

3.2.7 Ανασυγκρότηση

Η αιτία της ανάγκης τακτικής ανασυγκρότησης (defragmentation) στην BoltDB έχει εξηγηθεί στην υποενότητα 2.2.3. Η RocksDB, ως υλοποίηση LSM-δέντρου, δεν παρουσιάζει κατακερματισμό. Συνεπώς, δεν χρειάζεται να αντιστοιχίσουμε τη συνάρτηση ανασυγκρότησης `Defrag()` της BoltDB με κάποια λειτουργία στη RocksDB.

3.2.8 Λειτουργία μόνο ανάγνωσης

Η συνάρτηση `Open()` της BoltDB δέχεται ένα όρισμα το οποίο καθορίζει το αν η βάση δεδομένων θα ανοιχθεί σε κατάσταση λειτουργίας ανάγνωσης και εγγραφής ή σε κατάσταση λειτουργίας μόνο ανάγνωσης. Στην περίπτωση της RocksDB, χρησιμοποιούμε τις συναρτήσεις `Open()` και `OpenForReadOnly()` αντίστοιχα για το άνοιγμα στις προαναφερθείσες καταστάσεις λειτουργίας. Αμφότερες οι βάσεις δεδομένων μπορούν να ανοιχθούν από πολλές διεργασίες ταυτόχρονα σε κατάσταση λειτουργίας μόνο ανάγνωσης, αλλά μόνο από μία διεργασία σε κατάσταση λειτουργίας ανάγνωσης και εγγραφής.

4 Υλοποίηση

4.1 Η Βιβλιοθήκη Περιτύλιξης της RocksDB στην BoltDB

Όπως αναφέρθηκε και στην υποενότητα 3.1.1, για να επιτρέψουμε στον κώδικα του etcd να συνεχίσει να χρησιμοποιεί για τον αποθηκευτικό του μηχανισμό το ίδιο API με πριν, διατηρήσαμε τις δηλώσεις των συναρτήσεων της BoltDB ανέπαφες και αλλάξαμε μόνο την υλοποίησή τους, ώστε να καλούν τις αντίστοιχες συναρτήσεις της gorocksdb. Ασχοληθήκαμε μόνο με το υποσύνολο του API της BoltDB που χρησιμοποιεί ο etcd. Με ανάλογο τρόπο, διατηρήθηκαν οι δομές του πακέτου `bolt`, μόνο που τώρα λειτουργούν ως δομές περιτύλιξης των αντίστοιχων δομών της gorocksdb. Επίσης, στα σημεία που η αντιστοίχιση εννοιών της BoltDB σε έννοιες της RocksDB ήταν μη τετριμμένη προσθέσαμε επιπλέον κώδικα και βοηθητικές συναρτήσεις.

Σε αυτή την ενότητα, σκιαγραφούμε την τελική, *βελτιστοποιημένη* έκδοση της υλοποίησής μας. Λόγω της μεγάλης έκτασης που θα καταλάμβανε η αναλυτική περιγραφή όλων των συναρτήσεων και δομών της βιβλιοθήκης, παραθέτουμε απλώς δύο χαρακτηριστικά παραδείγματα. Ωστόσο, κάνουμε ειδική αναφορά στις περιπτώσεις στις οποίες συναντήσαμε εμπόδια στην εφαρμογή των σχεδιαστικών αποφάσεών μας, καθώς και στις ενέργειες που έγιναν ώστε να τα ξεπεράσουμε. Ο πλήρης πηγαίος κώδικας της υλοποίησής μας είναι διαθέσιμος στη διεύθυνση `https://github.com/boolean5/etcd-rocks`.

Χαρακτηριστικό παράδειγμα δομής περιτύλιξης στη βιβλιοθήκη μας αποτελεί η δομή **`DB`**, η οποία αντιπροσωπεύει τη βάση δεδομένων. Λειτουργεί ως δομή περιτύλιξης για τη δομή `DB` της gorocksdb, που παρέχει πρόσβαση στη βάση δεδομένων στον δίσκο. Τα πεδία της περιέχουν πληροφορίες για την τοποθεσία (path) του καταλόγου της βάσης δεδομένων και για το αν αυτή έχει ανοιχτεί σε κατάσταση λειτουργίας μόνο ανάγνωσης. Επιπλέον, συμπεριλαμβάνουν έναν μηχανισμό αμοιβαίου αποκλεισμού (mutex) που επιτρέπει την ύπαρξη μόνο ενός εγγραφέα κάθε στιγμή, έναν δείκτη σε μια δομή επιλογών (`Options struct`) κι έναν χάρτη (map) που περιέχει τους κάδους που υπάρχουν στη βάση δεδομένων. Τέλος, ανάμεσα στα πεδία της δομής `DB` βρίσκεται κι ένας δείκτης σε δομή `WriteBatch` της gorocksdb, που χρησιμοποιείται για την προσομοίωση μιας συναλλαγής BoltDB, καθώς κι ένας δείκτης σε δομή

`WriteBatchWithIndex`, που χρησιμοποιείται μόνο στην ειδική περίπτωση που αναφέρεται στην υποενότητα 4.2.5.

```
type DB struct {
	readOnly		bool
	db			*gorocksdb.DB
	path			string
	rwlock			sync.Mutex
	wb			*gorocksdb.WriteBatch
	wbwi			*gorocksdb.WriteBatchWithIndex
	options			*Options
	buckets			map[string]bool
}
```

Απόσπασμα κώδικα 1: *Η δομή DB*

Η συνάρτηση **`(b *Bucket) Put(key []byte, value []byte) error`**, που τοποθετεί ένα ζεύγος κλειδιού-τιμής σε έναν κάδο, αποτελεί χαρακτηριστικό παράδειγμα συνάρτησης περιτύλιξης. Η συνάρτηση αυτή αρχικά διενεργεί έλεγχο σφαλμάτων και στη συνέχεια πραγματοποιεί κλήση στη συνάρτηση `Put()` της gorocksdb, η οποία εφαρμόζεται στο συσχετισμένο με την τρέχουσα συναλλαγή `WriteBatch` ή `WriteBatchWithIndex`. Το κλειδί που τοποθετείται στη βάση δεδομένων προκύπτει από την συνένωση του προθέματος του κάδου με το αρχικό όνομα του κλειδιού.

```
func (b *Bucket) Put(key []byte, value []byte) error {
	if b.tx.db == nil {
		return ErrTxClosed
	} else if !b.tx.writable {
		return ErrTxNotWritable
	} else if len(key) == 0 {
		return ErrKeyRequired
	} else if len(key) > maxKeySize {
		return ErrKeyTooLarge
	} else if len(value) > maxValueSize {
		return ErrValueTooLarge
	}
```

```
13          s := make([][]byte, 3)
14          s[0] = b.prefix; s[1] = sep; s[2] = key
15          b.tx.wb.Put(concatenate(s), value)
16          return nil
17  }
```

Απόσπασμα κώδικα 2: *Η συνάρτηση* `Put()`

Κατά το άνοιγμα της βάσης δεδομένων από τη συνάρτηση `Open()`, πρέπει να λάβουμε υπ' όψιν μας την περίπτωση η τοποθεσία που δίνεται ως όρισμα αντί να δείχνει σε έναν κατάλογο όπως συμβαίνει συνήθως να δείχνει σε ένα `Checkpoint` σε μορφή αρχείου tar. Για τον σκοπό αυτόν υλοποιούμε έναν σχετικό έλεγχο με χρήση των συναρτήσεων `Stat()` και `IsDir()` της Go και της βοηθητικής συνάρτησης `IsTar()` της βιβλιοθήκης μας. Η `IsTar()` επιβεβαιώνει ότι πρόκειται για αρχείο tar επιθεωρώντας την υπογραφή *μαγικών αριθμών*[12] του. Έτσι, ανάλογα με το αποτέλεσμα του ελέγχου, καλούμε αν χρειάζεται τη συνάρτηση `untar()` πριν επιχειρήσουμε να ανοίξουμε τη βάση δεδομένων.

Η συνάρτηση `Begin()` συνήθως καλείται για να δημιουργήσει μια νέα συναλλαγή, ωστόσο κάποιες φορές καλείται με σκοπό την δημιουργία ενός στιγμιοτύπου της βάσης δεδομένων. Στο πλαίσιο της BoltDB, η έννοια του στιγμιοτύπου ταυτίζεται με την έννοια της συναλλαγής, είναι δηλαδή μια συνεπής, μόνο προς ανάγνωση όψη της βάσης δεδομένων. Στο πλαίσιο της δικής μας υλοποίησης όμως, χρειάζεται να διακρίνουμε μεταξύ των δύο περιπτώσεων. Έτσι, αρχικά ελέγχουμε ποια συνάρτηση κάλεσε την `Begin()`, με τη βοήθεια των συναρτήσεων `Caller()` και `CallersFrames()` της Go. Αν η συνάρτηση αυτή είναι η συνάρτηση `Snapshot` του πακέτου `backend` του etcd, τότε ακολουθεί κλήση της `createCheckpoint()`.

Στην τελική έκδοση της υλοποίησής μας, για την προσομοίωση μιας συναλλαγής BoltDB χρησιμοποιείται η δομή `WriteBatch`. Για λόγους αποδοτικότητας η δημιουργία δομής `WriteBatch` συμβαίνει μόνο μία φορά κατά διάρκεια λειτουργίας του etcd: όταν ξεκινά η πρώτη συναλλαγή. Στις επόμενες συναλλαγές επαναχρησιμοποιούμε την ίδια δομή, έχοντας φροντίσει για την εκκαθάρισή της κατά την κατοχύρωση ή αναίρεση της αντίστοιχης προηγούμενης συναλλαγής.

[12] Οι μαγικοί αριθμοί είναι συγκεκριμένες σταθερές που βρίσκονται σε συγκεκριμένες θέσεις μεταξύ των αρχικών bytes ενός αρχείου και παρέχουν έναν τρόπο διάκρισης μεταξύ των διάφορων μορφών αρχείων.

Η συνάρτηση `Size()` της βιβλιοθήκης μας καλείται από τον etcd σε δύο περιπτώσεις: κατά την κατοχύρωση μιας συναλλαγής με σκοπό τη σύγκριση του μεγέθους της βάσης δεδομένων στον δίσκο με το επιτρεπόμενο όριο και όταν ο πελάτης ζητά να μάθει το μέγεθος ενός `Snapshot` με την εντολή `etcdctl snapshot status`. Στην πρώτη περίπτωση, καλούμε τη βοηθητική συνάρτηση `rocksdbSize()`, η οποία επιστρέφει μια *εκτίμηση* του μεγέθους της βάσης δεδομένων. Καθώς η RocksDB δεν παρέχει κάποιον απλό και ακριβή τρόπο επιστροφής του συνολικού μεγέθους της στον χρήστη, το υπολογίζουμε προσεγγιστικά, προσθέτοντας το συνολικό μέγεθος των αρχείων SST στο μέγεθος των πινάκων μνήμης. Θεωρούμε ότι το μέγεθος των πινάκων μνήμης είναι περίπου όσο και το μέγεθος του WAL, αφού αυτό διαγράφεται αυτόματα κάθε φορά που οι πίνακες μνήμης μεταφέρονται στον δίσκο και ουσιαστικά περιέχει τις ίδιες εγγραφές.

Η συνάρτηση `WriteTo()` χρησιμοποιείται από την BoltDB για να γράψει ένα στιγμιότυπο στο άκρο μιας σωλήνωσης, ώστε αυτό να μεταφερθεί στον πελάτη που έκανε το σχετικό αίτημα λήψης. Στην υλοποίησή μας αντί να αντιγράφουμε μία μία τις σελίδες ενός αρχείου αντιστοίχισης μνήμης, ανοίγουμε το αρχείο tar του `Checkpoint` και με χρήση ενός επαναληπτικού βρόχου διαβάζουμε σταδιακά τα περιεχόμενά του τοποθετώντας τα σε έναν απομονωτή (buffer) και τα αντιγράφουμε στο άκρο της σωλήνωσης μέχρι να φτάσουμε στο τέλος του αρχείου.

4.2 Τροποποιήσεις στον Κώδικα του etcd

Παρόλο που προσπαθήσαμε να περιορίσουμε τις αλλαγές μας στο εσωτερικό του πακέτου `bolt` και να αποφύγουμε τις τροποποιήσεις στον κώδικα του etcd, υπήρξαν φορές που αυτές ήταν αναπόδραστες.

4.2.1 Ανασυγκρότηση

Όπως εξηγήσαμε στην υποενότητα 3.2.7, μετά την αντικατάσταση της BoltDB από την RocksDB δεν υπάρχει πλέον ανάγκη για ανασυγκρότηση. Στο πακέτο `backend` του etcd απομακρύνουμε τα περιεχόμενα της συνάρτησης `Defrag()` έτσι ώστε αυτή να επιστρέφει αμέσως χωρίς να πραγματοποιεί καμία ενέργεια. Επίσης, απομακρύνουμε τη σχετική συνάρτηση ελέγχου, `TestBackendDefrag()`.

4.2.2 Διαγραφή του καταλόγου της βάσης δεδομένων

Ο κώδικας του etcd αντιμετωπίζει τη βάση δεδομένων του backend ως αρχείο, με αποτέλεσμα όταν επιδιώκει τη διαγραφή της από το σύστημα αρχείων να καλεί τη συνάρτηση `Remove()` του πακέτου `os` της Go. Η συνάρτηση αυτή όμως προορίζεται αποκλειστικά για τη διαγραφή μεμονωμένων αρχείων και δεν έχει τη δυνατότητα αναδρομικής διαγραφής ολόκληρων καταλόγων [97]. Στη δική μας περίπτωση ή βάση δεδομένων του backend έχει τη μορφή καταλόγου, επομένως κρίθηκε απαραίτητη η αντικατάσταση όλων των κλήσεων της `Remove()` με κλήσεις της `RemoveAll()`.

4.2.3 Εφαρμογή του στιγμιοτύπου

Η συνάρτηση `applySnapshot()` του πακέτου `etcdserver` είναι υπεύθυνη για την αντικατάσταση του backend ενός κόμβου etcd με ένα δοθέν στιγμιότυπο. Το εμπόδιο που συναντήσαμε σε αυτό το σημείο προήλθε για μια ακόμη φορά από από τη διαφορά στη μορφή της βάσης δεδομένων μεταξύ BoltDB και RocksDB. Η συνάρτηση `Rename()` του πακέτου `os` της Go αποτυγχάνει όταν το όνομα προορισμού αντιστοιχεί σε κάποιον υπάρχοντα κατάλογο [97]. Η `Rename()` χρησιμοποιείται από την `applySnapshot()` για να δώσει στον κατάλογο του στιγμιοτύπου το όνομα του καταλόγου του προηγούμενου backend, προκαλώντας την αντικατάσταση του τελευταίου. Ξεπεράσαμε αυτήν τη δυσκολία απομακρύνοντας τον κατάλογο του προηγούμενου backend πριν επιχειρήσουμε τη μετονομασία του καταλόγου του στιγμιοτύπου.

Επιπλέον, υποχρεώσαμε το προηγούμενο backend να ολοκληρώσει τη διαδικασία κλεισίματός του πριν ανοίξουμε το νέο backend, καθώς το αρχείο `LOCK` της RocksDB δεν επέτρεπε τη δημιουργία δεύτερης βάσης δεδομένων στην ίδια τοποθεσία. Στην αρχική υλοποίηση του etcd, το κλείσιμο του παλιού backend διεκπεραιώνεται από διαφορετική goroutine, ώστε να αποφευχθεί η αναμονή ολοκλήρωσης της τελευταίας συναλλαγής.

4.2.4 Πρόσβαση στην κατάσταση στιγμιοτύπου

Η βοηθητική συνάρτηση `dbStatus()`, που καλείται από τη συνάρτηση `snapshotStatusCommandFunc()` όταν ο πελάτης εισάγει την εντολή `etcdctl snapshot status`, τροποποιήθηκε ώστε πριν την επιστροφή της να επαναφέρει το στιγμιότυπο στη μορφή

αρχείου tar, που είναι η αναμενόμενη από τη συνάρτηση `snapshotRestoreCommandFunc()`, που καλείται όταν ο πελάτης εισάγει την εντολή `etcdctl snapshot restore`.

4.2.5 Μετάβαση από WriteBatch σε WriteBatchWithIndex

Για λόγους διατήρησης της σημασιολογίας, σε ορισμένες περιπτώσεις είναι απαραίτητη η μετάβαση από τη δομή `WriteBatch`, που αντιστοιχεί σε μια συναλλαγή, σε μια δομή `WriteBatchWithIndex`. Συγκεκριμένα, αυτό συμβαίνει στις περιπτώσεις που ένα αίτημα ανάγνωσης κατευθύνεται σε κάδο εκτός του κάδου `keys`. Σε αντίθεση με τον κάδο `keys`, στον οποίο εφαρμόζεται MVCC, στους υπόλοιπους κάδους είναι πιθανή η παρουσία διπλοτύπων μεταξύ του απομονωτή (buffer) των δομών `batchTx` και `readTx` του πακέτου `backend` και του μηχανισμού αποθήκευσης. Τότε, με σκοπό την αποφυγή επιστροφής διπλοτύπων ή ζευγών κλειδιού-τιμής των οποίων η διαγραφή δεν έχει ακόμη κατοχυρωθεί, ο etcd παρακάμπτει τον απομονωτή και εφαρμόζει το αίτημα ανάγνωσης στο τρέχον `batchTx` αντί για το `readTx` όπως γίνεται συνήθως.

Στην τελική έκδοση της υλοποίησής μας όμως, αντιστοιχίζουμε στο `batchTx` τη δομή `WriteBatch`, η οποία σε αντίθεση με τη δομή `WriteBatchWithIndex` δεν παρέχει πρόσβαση στις μη κατοχυρωμένες ενημερώσεις της. Ξεπερνάμε αυτό το πρόβλημα πραγματοποιώντας αντιγραφή των εγγραφών του τρέχοντος `WriteBatch` σε ένα `WriteBatchWithIndex` πριν την διεκπεραίωση κάθε αιτήματος ανάγνωσης που κατευθύνεται σε κάδο εκτός του κάδου `keys`, διαδικασία που έχουμε υλοποιήσει στη συνάρτηση `Switch()` της βιβλιοθήκης μας. Προσθέτουμε κλήσεις της συνάρτησης `Switch()` εντός των συναρτήσεων `UnsafeRange()` και `UnsafeForEach()` που εφαρμόζονται πάνω σε `batchTx` και βρίσκονται στο πακέτο `backend`.

Θα μπορούσαμε να αποφύγουμε αυτήν την τροποποίηση αντιστοιχίζοντας εξαρχής το `batchTx` στο `WriteBatchWithIndex`, ωστόσο γνωρίζοντας ότι η συχνότητα αυτής της μετάβασης είναι σχετικά μικρή, επιλέγουμε να επωφεληθούμε από την καλύτερη επίδοση του `WriteBatch` σε σχέση με το `WriteBatchWithIndex`, όπως θα περιγράψουμε και στην υποενότητα 4.3.5.

4.2.6 Άρση του περιορισμού αποθηκευτικού χώρου

Η εφαρμογή της άρσης του περιορισμού αποθηκευτικού χώρου που περιγράψαμε στην υποενότητα 3.1.6 επιτυγχάνεται θέτοντας τις σταθερές `DefaultQuotaBytes` και `MaxQuotaBytes` του πακέτου `etcdserver` ίσες με `math.MaxInt64`. Ο χρήστης μπορεί ακόμη να επιβάλει ένα άνω φράγμα στο μέγεθος του backend εάν το επιθυμεί, μέσω της παραμέτρου `--quota-backend-bytes` κατά την εκκίνηση του etcd.

4.2.7 Script εγκατάστασης

Στο script εγκατάστασης του etcd (`build`), καθώς και στο script εγκατάστασης της πλατφόρμας λειτουργικού ελέγχου, θέτουμε τη μεταβλητή περιβάλλοντος `CGO_ENABLED` ίση με την τιμή 1, ώστε να επιτρέψουμε τη χρήση του cgo. Επιπλέον, τοποθετούμε στις μεταβλητές περιβάλλοντος `CGO_CFLAGS` και `CGO_LDFLAGS` τις απαραίτητες σημαίες (flags) μεταγλωττιστή και συνδέτη για τη χρήση της RocksDB ως *κοινόχρηστη* βιβλιοθήκη. Τέλος, αντικαθιστούμε την τιμή της `ORG_PATH` έτσι ώστε να δείχνει στο δικό μας αποθετήριο και προσθέτουμε στη συνάρτηση `etcd_build()` την κατάλληλη εντολή ώστε το script να εγκαταστήσει και το εργαλείο `benchmark` εκτός από τον `etcd` και την εφαρμογή πελάτη `etcdctl`.

4.3 Βελτιστοποιήσεις

Η ενότητα αυτή παρέχει μια επισκόπηση των σημαντικότερων από τις βελτιστοποιήσεις που εφαρμόσαμε στην αρχική μας υλοποίηση.

4.3.1 Αρχική υλοποίηση

Η πρώτη μας προσέγγιση στην ενσωμάτωση της RocksDB στον etcd ακολουθεί πιστά τις σχεδιαστικές αποφάσεις που παρουσιάστηκαν στο κεφάλαιο 3. Μεταξύ άλλων, έγινε αντιστοίχιση των συναλλαγών της BoltDB με τις απαισιόδοξες συναλλαγές της RocksDB. Η απόφαση αυτή ήταν σημασιολογικά ορθή αλλά οδήγησε στην ανάπτυξη περίπλοκου κώδικα και επέβαλε περιττή καθυστέρηση. Επίσης, όπως συνέβαινε και με την BoltDB πριν την αλλαγή, κάθε αίτημα ανάγνωσης / εγγραφής / διαγραφής είχε ως αποτέλεσμα δύο κλήσεις στο πακέτο `bolt`, μία για την εύρεση του κατάλληλου κάδου

και μία δεύτερη για την ανάγνωση / εγγραφή / διαγραφή του ζεύγους κλειδιού-τιμής από αυτόν.

4.3.2 Πρόσβαση στους κάδους

Όπως θα δείξουμε στο κεφάλαιο 5, η επίδοση της αρχικής υλοποίησης δεν ήταν ιδιαίτερα ικανοποιητική. Ειδικότερα, παρόλο που θεωρητικά αναμενόταν βελτίωση της επίδοσης των εγγραφών, προκλήθηκε επιδείνωσή της. Σε αυτό το σημείο, χρησιμοποιήσαμε το *εργαλείο δημιουργίας προφίλ* (profiler) της Go με σκοπό να εντοπίσουμε τα αίτια περιορισμού της επίδοσης. Συγκεκριμένα, λάβαμε ένα προφίλ 30 δευτερολέπτων της CPU ενώ φορτώναμε στον etcd 1000000 ζεύγη κλειδιού-τιμής με χρήση του εργαλείου `benchmark`. Μια ματιά στα αποτελέσματα αποκάλυψε ότι η πιο χρονοβόρα λειτουργία στον etcd ήταν οι *κλήσεις cgo*.

Με μια πιο προσεκτική μελέτη των αποτελεσμάτων του εργαλείου δημιουργίας προφίλ, αντιληφθήκαμε ότι οι περισσότερες κλήσεις cgo προέρχονταν από τη συνάρτηση `Bucket()` του πακέτου `bolt`. Όπως αναφέρθηκε προηγουμένως, η πρώτη ενέργεια για την εξυπηρέτηση ενός αιτήματος εγγραφής από το backend είναι η εύρεση του κατάλληλου κάδου. Επιπλέον, η αρχική μας υλοποίηση της συνάρτησης `Bucket()` ακολουθούσε αφελώς την αντίστοιχη υλοποίηση της BoltDB, με αποτέλεσμα να περιλαμβάνει δύο κλήσεις cgo.

Για να αποφύγουμε αυτές τις κλήσεις cgo διατηρούμε *στη μνήμη* το σύνολο των υπαρχόντων κάδων, υπό τη μορφή χάρτη (map) της Go. Με αυτόν τον τρόπο η συνάρτηση `Bucket()` δεν χρειάζεται πια να απευθύνει κάποιο ερώτημα στον αποθηκευτικό μηχανισμό. Ο χάρτης των κάδων δημιουργείται από τη συνάρτηση `Open()` και ενημερώνεται από τη συνάρτηση `CreateBucket()`. Με αυτή την αλλαγή, περιορίσαμε τον αριθμό κλήσεων που απαιτούνται ανά αίτημα εγγραφής στο backend από τρεις σε μόνο μία[13].

[13]Στο επίπεδο μιας εγγραφής στον etcd περιορίσαμε τις κλήσεις cgo από 6 σε 2, καθώς αυτή μεταφράζεται σε δύο εγγραφές στο backend: μία για την αποθήκευση του ζεύγους κλειδιού-τιμής και μία δεύτερη για την ενημέρωση του `consistentIndex`.

4.3.3 Αισιόδοξες συναλλαγές

Στο πλαίσιο του etcd, μόνο μία συναλλαγή ανάγνωσης και εγγραφής είναι ενεργή κάθε στιγμή. Συνεπώς, δεν υπάρχουν διενέξεις και είναι ασφαλής η αντικατάσταση των απαισιόδοξων συναλλαγών με αισιόδοξες συναλλαγές. Αυτή η αλλαγή προσέφερε μια σημαντική βελτίωση στην επίδοση των εγγραφών, καθώς απέφυγε την καθυστέρηση που εισάγουν τα κλειδώματα των απαισιόδοξων συναλλαγών. Για την εφαρμογή αυτής της βελτιστοποίησης χρειάστηκε η αντικατάσταση της συνδεδεμένης με τη δομή `DB` δομής `TxnDB` της gorocksdb με τη δομή `OptimisticTxnDB`, καθώς και η αντικατάσταση κάποιων συναρτήσεων της gorocksdb με την ισοδύναμη μορφή τους για αισιόδοξες συναλλαγές.

4.3.4 WriteBatchWithIndex & Snapshot

Με βάση το γεγονός ότι στο πλαίσιο του etcd μόνο μία συναλλαγή ανάγνωσης και εγγραφής είναι ενεργή κάθε στιγμή, συμπεραίνουμε ότι δεν υπάρχει ανάγκη για έλεγχο ταυτοχρονισμού στην πλευρά της RocksDB. Με άλλα λόγια, είναι αρκετή η εξασφάλιση ατομικότητας και μονιμότητας. Αυτό πρακτικά σημαίνει ότι μπορούμε να προσομοιώσουμε μια συναλλαγή ανάγνωσης και εγγραφής της BoltDB με μια δομή `WriteBatchWithIndex` της RocksDB, η οποία σε αντίθεση με τη δομή `WriteBatch` παρέχει τη δυνατότητα ανάγνωσης των μη κατοχυρωμένων ενημερώσεών της. Επιπλέον, μπορούμε να προσομοιώσουμε μια συναλλαγή μόνο ανάγνωσης της BoltDB με ένα `Snapshot` της RocksDB, το οποίο από σημασιολογική άποψη είναι το ίδιο πράγμα: μια συνεπής, μόνο προς ανάγνωση όψη της βάσης δεδομένων.

Η δομή `WriteBatchWithIndex` εισάγει πολύ μικρότερη καθυστέρηση από μια συναλλαγή της RocksDB καθώς δεν πραγματοποιεί έλεγχο διενέξεων κατά την κατοχύρωση. Για να υλοποιήσουμε αυτήν τη βελτιστοποίηση χρειάστηκαν οι ακόλουθες αλλαγές:

- Αντικατάσταση της συνδεδεμένης με τη δομή `DB` δομής `OptimisticTxnDB` της gorocksdb με τη δομή `DB` της gorocksdb. Επίσης, αντικατάσταση της συνδεδεμένης με τη δομή `Tx` δομής `Txn` της gorocksdb με τη δομή `WriteBatchWithIndex`.

- Τροποποίηση της συνάρτησης `beginRWTx()` ώστε να δημιουργεί μια δομή `WriteBatchWithIndex` αντί να ξεκινά μια νέα αισιόδοξη συναλλαγή. Ακόμη, τροποποίηση της συνάρτησης `beginROTx()` ώστε να δημιουργεί ένα νέο `Snapshot` αντί για μια αισιόδοξη συναλλαγή.

- Στη συνάρτηση `Rollback()`, εκκαθάριση της δομής `WriteBatchWithIndex` με κλήση της `Clear()` ή απελευθέρωση του `Snapshot` αντί για αναίρεση της αισιόδοξης συναλλαγής. Παρομοίως, στη συνάρτηση `Commit()`, χρήση της `WriteWithIndex()` της gorocksdb για την εγγραφή του `WriteBatchWithIndex` στη βάση δεδομένων, αντί για κατοχύρωση της αισιόδοξης συναλλαγής.

- Στις συναρτήσεις `Put()` και `Delete`, εφαρμογή της εγγραφής / διαγραφής στη δομή `WriteBatchWithIndex`. Στη συνάρτηση `Get()`, κλήση της `GetBytesFromBatchAndDB()`.

- Στη συνάρτηση `Cursor()`, χρήση της `NewIteratorWithBase()` για τη δημιουργία επαναλήπτη με δυνατότητα συνδυασμού των μη κατοχυρωμένων ενημερώσεων του `WriteBatchWithIndex` με τα περιεχόμενα της βάσης δεδομένων.

- Απομάκρυνση του κώδικα διαχείρισης των `OptimisticTxnOptions`.

4.3.5 WriteBatch

Η δομή `WriteBatchWithIndex` εισάγει λίγο περισσότερη καθυστέρηση από τη δομή `WriteBatch`, αφού όπως είδαμε στην υποενότητα 2.3.4, διατηρεί έναν εσωτερικό απομονωτή υπό τη μορφή ευρετηρίου. Μια πρόσφατη αλλαγή στο πακέτο `backend` του etcd οδήγησε στην αποσύμπλεξη των αναγνώσεων από τις εγγραφές, διατηρώντας έναν ενδιάμεσο απομονωτή που καθιστά δυνατή την πρόσβαση των αναγνώσεων στις μη κατοχυρωμένες εγγραφές. Το γεγονός αυτό, μας επιτρέπει να χρησιμοποιήσουμε τη δομή `WriteBatch` στη θέση της `WriteBatchWithIndex`. Ωστόσο, όπως εξηγήθηκε στην υποενότητα 4.2.5, υπάρχουν ορισμένες περιπτώσεις στις οποίες η δομή `WriteBatchWithIndex` εξακολουθεί να είναι απαραίτητη. Για την υλοποίηση αυτής της βελτιστοποίησης χρειάστηκε να αντικαταστήσουμε κάποιες συναρτήσεις της gorocksdb που σχετίζονται με τη δομή `WriteBatchWithIndex` με τις ισοδύναμές

τους για τη δομή `WriteBatch`, καθώς και να υλοποιήσουμε τη συνάρτηση `Switch()`, στην οποία αναφερθήκαμε στην υποενότητα 4.2.5.

4.3.6 Ρύθμιση παραμέτρων της RocksDB

Ο συντονισμός (tuning) της RocksDB είναι μια περίπλοκη διαδικασία που περιλαμβάνει τη ρύθμιση περισσότερων από 120 παραμέτρων με διαφορετικούς βαθμούς αλληλεξάρτησης. Οι προεπιλεγμένες τιμές οδηγούν σε χαμηλή επίδοση, καθώς δεν αξιοποιούν πλήρως τις δυνατότητες του συστήματος. Οι ίδιοι οι σχεδιαστές της RocksDB παραδέχονται ότι ο βέλτιστος συντονισμός της είναι μη τετριμμένη διαδικασία και προτείνουν μια πειραματική προσέγγιση [64]. Εκτός από το διαθέσιμο υλικό, ο συντονισμός εξαρτάται και από το είδος του αναμενόμενου φόρτου εργασίας. Μπορεί να ιδωθεί ως ένας συμβιβασμός μεταξύ ενίσχυσης εγγραφών, αναγνώσεων και χώρου. Στην περίπτωσή μας, επιδιώκουμε να ευνοήσουμε την ενίσχυση εγγραφών εις βάρος των δύο άλλων τύπων ενίσχυσης, καθώς ο τελικός μας στόχος είναι η βελτίωση της επίδοσης των εγγραφών στον etcd.

Η προσέγγιση μας στον συντονισμό της RocksDB συνίσταται αρχικά στην επιλογή ενός υποσυνόλου των παραμέτρων που θεωρούμε ότι θα έχει τη μεγαλύτερη επίδραση στην επίδοση των εγγραφών, με βάση τις συμβουλές του οδηγού συντονισμού [64] και του φόρουμ των προγραμματιστών της RocksDB, καθώς και τις αναφορές προσεγγίσεων από τρίτους. Κατόπιν, χρησιμοποιούμε το εργαλείο `benchmark` του etcd για τη διενέργεια πειραμάτων. Ξεκινάμε από μία παράμετρο και δοκιμάζουμε γι᾽ αυτήν πολλαπλές τιμές, επιλέγοντας αυτήν που μεγιστοποιεί τη διεκπεραιωτική ικανότητα εγγραφών. Στη συνέχεια, σταθεροποιούμε αυτήν την παράμετρο στην επιλεγμένη τιμή και εφαρμόζουμε την ίδια διαδικασία στην επόμενη, ώσπου να σταθεροποιήσουμε όλες τις παραμέτρους.

Ακολούθως, αναφέρουμε τις πιο επιδραστικές από τις παραμέτρους με τις οποίες πειραματιστήκαμε και τελικά διατηρήσαμε στην τελική μας υλοποίηση της συνάρτησης `createOptions()`.

- **Φίλτρα Bloom:** Τα φίλτρα Bloom παρέχουν έναν τρόπο να γνωρίζουμε αν ένα αρχείο δεν περιέχει ένα συγκεκριμένο κλειδί, χωρίς την πραγματοποίηση πρόσβασης στο αρχείο. Υλοποιούνται ως ένας πίνακας bit σε συνδυασμό με k διαφορετικές συναρτήσεις κατακερματισμού [66]. Με τη διατήρηση φίλτρων Bloom

στη μνήμη καταφέρνουμε να μειώσουμε αισθητά τις προσβάσεις σε αρχεία SST στον δίσκο ανά αίτημα ανάγνωσης, καθώς αποφεύγονται οι προσβάσεις στα αρχεία στα οποία γνωρίζουμε ότι δεν περιέχεται το εκάστοτε ζητούμενο κλειδί.

- **Ρυθμίσεις παραλληλισμού:** Στην αρχιτεκτονική του LSM-δέντρου υπάρχουν δύο διεργασίες που λειτουργούν στο παρασκήνιο: η μεταφορά πινάκων μνήμης στον δίσκο (flushing) και οι συμπυκνώσεις. Ρυθμίζουμε τη RocksDB ώστε να αξιοποιήσει τον ταυτοχρονισμό στο επίπεδο τεχνολογίας αποθήκευσης, διαθέτοντας ένα νήμα για τη μεταφορά πινάκων μνήμης και αριθμό νημάτων ίσο με τον αριθμό των πυρήνων της CPU μείον ένα για τις συμπυκνώσεις. Ακόμη, δίνουμε τιμή μεγαλύτερη του 1 στον μέγιστο αριθμό νημάτων που μπορούν να πραγματοποιούν ταυτόχρονα μια συμπύκνωση, χωρίζοντάς την σε πολλαπλές μικρότερες συμπυκνώσεις που τρέχουν παράλληλα. Οι αποδοτικότερες συμπυκνώσεις επηρεάζουν θετικά τη διεκπεραιωτική ικανότητα εγγραφών [113].

- **Ρυθμίσεις μεταφοράς πινάκων μνήμης στον δίσκο:** Προσδιορίζουμε κατάλληλα το μέγεθος και τον μέγιστο αριθμό των πινάκων μνήμης. Η ύπαρξη περισσότερων του ενός πινάκων μνήμης επιτρέπει την συνέχιση των εγγραφών κατά τη διάρκεια μεταφοράς ενός πίνακα μνήμης στον δίσκο. Επίσης, καθορίζουμε τον ελάχιστο αριθμό πινάκων μνήμης που πρέπει να συγχωνευθούν πριν μεταφερθούν στον δίσκο. Όταν πολλαπλοί πίνακες μνήμης συγχωνεύονται είναι πιθανό να εγγραφούν στη συσκευή αποθήκευσης λιγότερα δεδομένα αφού οι ενημερώσεις που αφορούν το ίδιο κλειδί συγχωνεύονται σε μία [64].

- **Ρυθμίσεις συμπύκνωσης:** Σύμφωνα με όσα αναφέραμε στην υποενότητα 2.3.3 και τα αποτελέσματα των πειραμάτων συντονισμού που κάναμε, ο καθολικός (universal) τύπος συμπύκνωσης είναι καταλληλότερος από τον τύπο συμπύκνωσης με επίπεδα (levelled) για φόρτο εργασίας με έμφαση στις εγγραφές. Ωστόσο, αυξάνει την ενίσχυση αναγνώσεων και χώρου.

- **Ρυθμίσεις συμπίεσης:** Κάθε block συμπιέζεται πριν την εγγραφή του στη συσκευή μόνιμης αποθήκευσης. Διατηρούμε την προεπιλεγμένη μέθοδο συμπίεσης, Snappy, ή οποία είναι αρκετά ταχεία.

4.4 Εξωτερικές Συνεισφορές

Σε αρκετές περιπτώσεις κατά τη διάρκεια της υλοποίησής μας, χρειαστήκαμε λειτουργίες που δεν ήταν διαθέσιμες στο C API της RocksDB ή στην gorocksdb και προχωρήσαμε στην ανάπτυξη του αντίστοιχου κώδικα για αυτές τις δύο βιβλιοθήκες. Ο πηγαίος κώδικας των συνεισφορών μας είναι διαθέσιμος στα παρακάτω αποθετήρια:

- `https://github.com/boolean5/rocksdb`
- `https://github.com/boolean5/gorocksdb`

Ακολουθεί σύντομη περιγραφή τους.

- Κανένα από τα δύο είδη συναλλαγών δεν υποστηριζόταν από το C API της RocksDB και την gorocksdb. Συνεπώς, προσθέσαμε και στις δύο βιβλιοθήκες περισσότερες από 40 δομές και συναρτήσεις για τη διαχείριση συναλλαγών.
- Στην gorocksdb υπήρχε επίσης έλλειψη υποστήριξης του `WriteBatchWithIndex`, γεγονός που μας οδήγησε στην ανάπτυξη των απαιτούμενων συναρτήσεων για τη δημιουργία και καταστροφή του, την εγγραφή, ανάγνωση και διαγραφή τιμών από αυτό, την εκκαθάρισή του, την εγγραφή του στη βάση δεδομένων κτλ.
- Από το C API της RocksDB και την gorocksdb έλειπε επίσης η υποστήριξη του `Checkpoint`. Επεκτείναμε και τις δύο βιβλιοθήκες με τις σχετικές δομές και συναρτήσεις.
- Κάποιες από τις παραμέτρους ρύθμισης της RocksDB δεν είχαν εξαχθεί στο C API της και στην gorocksdb. Για να τις καταστήσουμε προσβάσιμες από κώδικα γραμμένο σε Go, προσθέσαμε τις αντίστοιχες συναρτήσεις και στις δύο βιβλιοθήκες.
- Η συνάρτηση `GetProperty()` της RocksDB παρέχει πρόσβαση σε χρήσιμες πληροφορίες, όπως το συνολικό μέγεθος των πινάκων μνήμης ή των αρχείων SST. Προσθέσαμε στο C API και στην gorocksdb μια επιπλέον εκδοχή αυτής της συνάρτησης, συμβατή με τον τύπο βάσης δεδομένων που υποστηρίζει συναλλαγές.

4.5 Έλεγχος Ορθότητας

Σε αυτήν την ενότητα, παρουσιάζουμε τις μεθόδους ελέγχου μέσω των οποίων επαληθεύσαμε ότι ο etcd συνεχίζει να λειτουργεί με τον αναμενόμενο τρόπο μετά την αντικατάσταση του μηχανισμού αποθήκευσής του με τη RocksDB. Έπειτα από αρκετούς γύρους δοκιμών και διορθώσεων σφαλμάτων καταφέραμε να καταστήσουμε την υλοποίησή μας εύρωστη και αξιόπιστη.

4.5.1 Δοκιμές ενοτήτων

Σκοπός των δοκιμών ενοτήτων (unit tests) είναι ο έλεγχος ορθότητας μεμονωμένων δομικών στοιχείων ενός πακέτου, όπως μια συνάρτηση. Ο etcd διαθέτει ένα ειδικό script με την ονομασία `test`, το οποίο κάνοντας χρήση του εργαλείου `go test` τρέχει όλες τις δοκιμές ενοτήτων που βρίσκονται στον κώδικά του. Παραδείγματα λειτουργιών που ελέγχονται από τις δοκιμές ενοτήτων αποτελούν η ανάγνωση και εγγραφή ζευγών κλειδιού-τιμής σε μια συστοιχία, η προσθήκη και αφαίρεση μελών από αυτήν, η λήψη στιγμιοτύπου και η εκκίνηση ενός νέου μέλους από αυτό κτλ.

Επιπλέον, το script `test` ελέγχει την αλληλεπίδραση πελάτη και εξυπηρετητή: εκκινεί έναν εξυπηρετητή etcd στον οποίο στέλνει αιτήματα και ελέγχει τις απαντήσεις του σε αυτά. Επίσης, δημιουργεί μια τοπική συστοιχία αποτελούμενη από 3 μέλη προκειμένου να επαληθεύσει τη σωστή λειτουργία της διεπαφής γραμμής εντολών του etcd [114].

4.5.2 Πλατφόρμα λειτουργικού ελέγχου

Η πλατφόρμα λειτουργικού ελέγχου (functional test suite) του etcd είναι σχεδιασμένη για να εξασφαλίζει ότι ο etcd τηρεί τις εγγυήσεις αξιοπιστίας και ευρωστίας του. Η ροή εργασιών της είναι η εξής [115]:

1. Δημιουργεί μια νέα συστοιχία etcd και στέλνει διαρκώς σε αυτήν αιτήματα εγγραφής.

2. Εισάγει ένα σφάλμα στη συστοιχία. Διάφοροι τύποι σφαλμάτων συστήματος και δικτύου έχουν μοντελοποιηθεί: τερματισμός τυχαίου κόμβου, τερματισμός ηγέτη, τερματισμός πλειοψηφίας, τερματισμός όλων των κόμβων, τερματισμός

κόμβου και επαναφορά του μετά από μεγάλο χρονικό διάστημα ώστε να προκληθεί αποστολή στιγμιοτύπου, διαμέριση δικτύου, δίκτυο με καθυστερήσεις.

3. Επιδιορθώνει το σφάλμα και αναμένει την αποκατάσταση ομαλής λειτουργίας της συστοιχίας etcd εντός σύντομου χρονικού διαστήματος.

4. Περιμένει μέχρι η συστοιχία να είναι πλήρως συνεπής και ξεκινά τον επόμενο γύρο εισαγωγής σφαλμάτων. Κάθε γύρος περιλαμβάνει όλους τους τύπους σφαλμάτων που αναφέρθηκαν στο βήμα 2.

Σε περίπτωση που η συστοιχία δεν καταφέρει να αποκαταστήσει τη λειτουργία της μετά από κάποιο σφάλμα, δημιουργείται ένα αρχείο με την κατάσταση της συστοιχίας το οποίο μπορεί να μελετηθεί αργότερα για τον εντοπισμό του προβλήματος [115]. Αφήσαμε την πλατφόρμα λειτουργικού ελέγχου να τρέξει για 10 συνεχόμενους γύρους χωρίς να αναφερθούν προβλήματα στη συστοιχία μας.

Η πλατφόρμα λειτουργικού ελέγχου αποτελείται από δύο δομικά στοιχεία: τον δαίμονα `etcd-agent` που τρέχει σε κάθε κόμβο της συστοιχίας και διαχειρίζεται την κατάσταση του etcd και τον `etcd-tester` που τρέχει σε ξεχωριστό μηχάνημα, εισάγει σφάλματα μέσω επικοινωνίας με τους `etcd-agent` και επαληθεύει την ορθή λειτουργία του etcd.

5 Πειραματική Αξιολόγηση

5.1 Εργαλεία, Μεθοδολογία & Περιβάλλον

Για την πειραματική αξιολόγηση κάνουμε αποκλειστική χρήση του ενσωματωμένου εργαλείου γραμμής εντολών `benchmark` του etcd, ενώ χρησιμοποιήθηκαν οι παρακάτω εκδόσεις λογισμικού:

- Go 1.8.3
- etcd 3.2.0, τροποποιημένος όπως περιγράφηκε στην ενότητα 4.2
- RocksDB 5.5.1, επεκτεταμένη με τις συνεισφορές που αναφέρθηκαν στην ενότητα 4.4
- gorocksdb, επεκτεταμένη με τις συνεισφορές που αναφέρθηκαν στην ενότητα 4.4

Κάθε πείραμα πραγματοποιήθηκε τρεις φορές και για την εξαγωγή των γραφημάτων που ακολουθούν χρησιμοποιήθηκε ο μέσος όρος των αποτελεσμάτων. Όπου δεν αναφέρεται κάτι διαφορετικό, το μέγεθος των κλειδιών προς ανάγνωση / εγγραφή ήταν 8 bytes και το μέγεθος των τιμών ήταν 256 bytes. Το περιβάλλον στο οποίο διενεργήθηκαν τα πειράματα αποτελείτο από 4 εικονικές μηχανές τύπου `m3.xlarge` στο Elastic Compute Cloud (EC2) της Amazon. Τρεις από αυτές σχημάτιζαν μια συστοιχία etcd, ενώ η τέταρτη έπαιζε τον ρόλο του πελάτη. Κάθε μία ήταν εξοπλισμένη με 4 πυρήνες, 15GB μνήμης, 2 δίσκους SSD των 40GB και εγκατάσταση του λειτουργικού συστήματος Ubuntu.

Στην επόμενη ενότητα θα χρησιμοποιήσουμε την ακόλουθη σύμβαση για την ονομασία των διάφορων εκδόσεων της υλοποίησής μας:

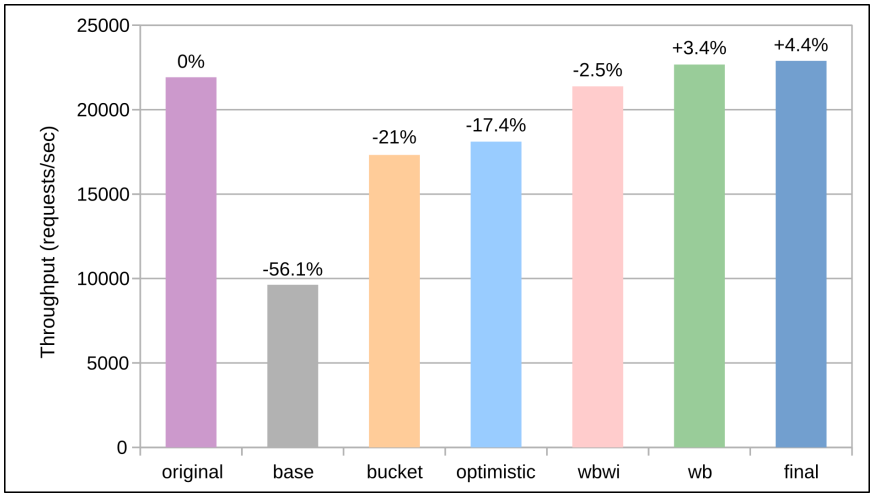
- `original`: η πρωτότυπη εκδοχή του etcd, με την BoltDB ως μηχανισμό αποθήκευσης, στην οποία βασίστηκαν οι επόμενες εκδόσεις.
- `base`: η αρχική μας υλοποίηση, χωρίς καμία βελτιστοποίηση.
- `bucket`: αυτή η έκδοση περιέχει την σχετική με την πρόσβαση στους κάδους βελτιστοποίηση.

- `optimistic`: σε αυτήν την έκδοση, οι απαισιόδοξες συναλλαγές αντικαταστάθηκαν με αισιόδοξες.
- `wbwi`: σε αυτήν την έκδοση, οι συναλλαγές αντικαταστάθηκαν με τον συνδυασμό `WriteBatchWithIndex` και `Snapshot`.
- `wb`: σε αυτήν την έκδοση, η δομή `WriteBatchWithIndex` αντικαταστάθηκε με την περισσότερο αποδοτική δομή `WriteBatch`.
- `final`: η τελική, βελτιστοποιημένη έκδοση, όπως προέκυψε μετά τη ρύθμιση παραμέτρων[14] της RocksDB.

5.2 Αποτελέσματα

5.2.1 Επίδοση εγγραφών

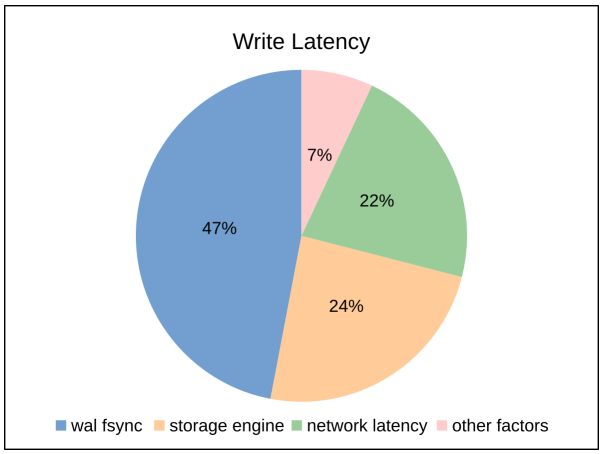
Η διεκπεραιωτική ικανότητα εγγραφών που επιτεύχθηκε από κάθε έκδοση της υλοποίησής μας, καθώς και η βελτίωσή της ως ποσοστό της διεκπεραιωτικής ικανότητας της έκδοσης `original`, μπορούν να μελετηθούν στο Σχήμα 6. Σε αυτό το πείραμα 1000 πελάτες εισήγαγαν στον etcd 1000000 ζεύγη κλειδιού-τιμής. Οι πελάτες προσομοιώνονται από το εργαλείο `benchmark` με διαφορετικές goroutines. Όπως αναμέναμε, κάθε βελτιστοποίηση που εφαρμόσαμε επέφερε μια αισθητή βελτίωση επίδοσης σε σχέση με την προηγούμενη έκδοση της υλοποίησης.



Σχήμα 6: *Διεκπεραιωτική ικανότητα εγγραφών του etcd σε όλες τις εκδόσεις*

[14]Σημειώνεται ότι έχουμε εφαρμόσει σε όλες τις εκδόσεις τις ρυθμίσεις παραλληλισμού που περιγράφηκαν στην υποενότητα 4.3.6.

Προκειμένου να ερμηνεύσει κανείς σωστά αυτά τα αποτελέσματα, θα πρέπει να γνωρίζει ότι η καθυστέρηση που επιβάλλεται από τον μηχανισμό αποθήκευσης του etcd είναι μόνο ένας από τους παράγοντες που συνεισφέρουν στη συνολική καθυστέρηση εξυπηρέτησης ενός αιτήματος εγγραφής. Όπως αναφέρθηκε στην υποενότητα 2.1.5, κυρίαρχοι παράγοντες που επηρεάζουν την επίδοση είναι η καθυστέρηση E/E στον δίσκο λόγω των κλήσεων `fsync` για την ενημέρωση του WAL, και η καθυστέρηση δικτύου λόγω της ανταλλαγής μηνυμάτων για την επίτευξη ομοφωνίας. Στο Σχήμα 7 βλέπουμε μια προσέγγιση του ποσοστού συνεισφοράς κάθε παράγοντα στην συνολική καθυστέρηση μιας εγγραφής στον etcd στο πειραματικό μας περιβάλλον, όπως αυτή προέκυψε από τις μετρικές χρόνου εκτέλεσης του etcd, τη μέση καθυστέρηση μιας εγγραφής στον αποθηκευτικό μηχανισμό και τον χρόνο αποστολής μετ᾽ επιστροφής (RTT) μεταξύ δύο εικονικών μηχανών στο EC2.

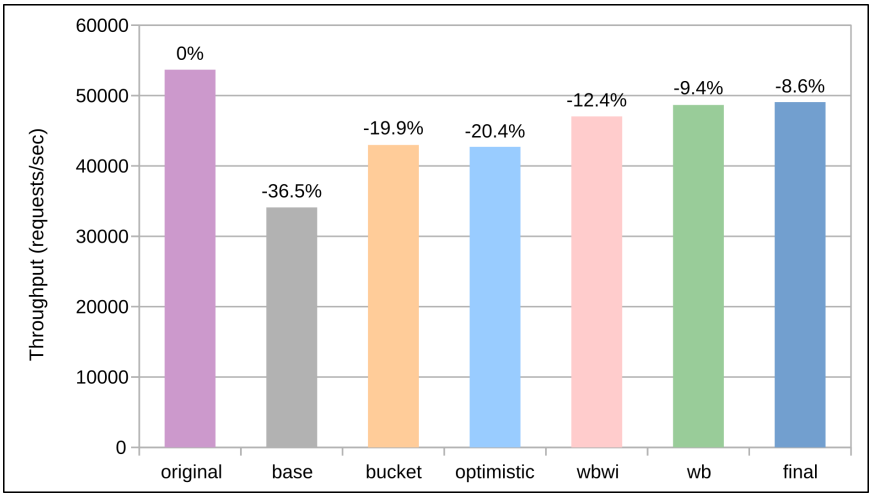


Σχήμα 7: *Παράγοντες που συνεισφέρουν στην καθυστέρηση εγγραφών στον etcd*

5.2.2 Επίδοση αναγνώσεων

Το Σχήμα 8 απεικονίζει τη διεκπεραιωτική ικανότητα αναγνώσεων *μεμονωμένων κλειδιών* (point lookups) σε κάθε έκδοση της υλοποίησής μας. Τα αποτελέσματα αυτά προέκυψαν από την εκτέλεση πειράματος κατά το οποίο 1000 πελάτες πραγματοποίησαν 1000000 αιτήματα ανάγνωσης ενός συγκεκριμένου κλειδιού που είχαμε από πριν φροντίσει να αποθηκεύσουμε στον etcd, ανάμεσα σε 1000000 άλλα.

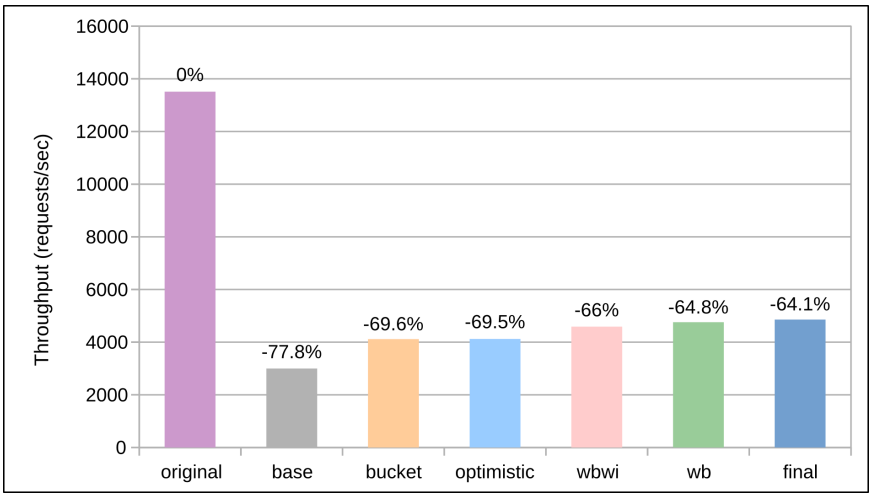
Όπως αναμέναμε, παρατηρείται επιδείνωση της επίδοσης των αναγνώσεων όταν αντικαθιστούμε την BoltDB με τη RocksDB. Ωστόσο, παρόλο που οι βελτιστοποιήσεις μας σχεδιάστηκαν στο πνεύμα της βελτίωσης της επίδοσης των εγγραφών, βοήθησαν



Σχήμα 8: *Διεκπεραιωτική ικανότητα αναγνώσεων μεμονωμένων κλειδιών του etcd*

και στην επαναφορά της επίδοσης των αναγνώσεων σε αποδεκτά επίπεδα. Στην έκδοση `final` παρατηρούμε μια μικρή άνοδο της επίδοσης σε σχέση με την έκδοση `wb`, εξαιτίας της ρύθμισης παραμέτρων της RocksDB, που συμπεριέλαβε την αύξηση του μεγέθους και του αριθμού των πινάκων μνήμης και την προσθήκη φίλτρων Bloom.

Η διεκπεραιωτική ικανότητα αναγνώσεων *εύρους κλειδιών* (range queries) μπορεί να παρατηρηθεί στο Σχήμα 9. Τα αποτελέσματα αυτά προέκυψαν από την εκτέλεση πειράματος κατά το οποίο 100 πελάτες πραγματοποίησαν 100000 αιτήματα ανάγνωσης ενός συγκεκριμένου εύρους κλειδιών.



Σχήμα 9: *Διεκπεραιωτική ικανότητα αναγνώσεων εύρους κλειδιών του etcd*

Σε αυτήν την περίπτωση, η επιδείνωση είναι πολύ μεγαλύτερη από ό,τι στην περίπτωση των αναγνώσεων μεμονωμένων κλειδιών. Γενικά, οι αναγνώσεις εύρους κλει-

διών στα LSM-δέντρα είναι αργές, επειδή τα κλειδιά που ανήκουν στο εύρος μπορεί να είναι διασκορπισμένα σε πολλαπλά επίπεδα του δέντρου. Έτσι, κάθε επίπεδο *πρέπει* να ελεγχθεί, κάτι που μεταφράζεται σε πολυάριθμες προσβάσεις σε αρχεία. Επιπλέον, τα οφέλη των φίλτρων Bloom για τις αναγνώσεις μεμονωμένων κλειδιών δεν επεκτείνονται και στις αναγνώσεις εύρους κλειδιών, καθώς τα ζητούμενα κλειδιά δεν είναι εκ των προτέρων γνωστά. Από την άλλη πλευρά, η δομή της BoltDB στον δίσκο της δίνει ξεκάθαρο πλεονέκτημα για αυτόν τον τύπο φόρτου εργασίας, όπως είδαμε στην υποενότητα 2.2.1.

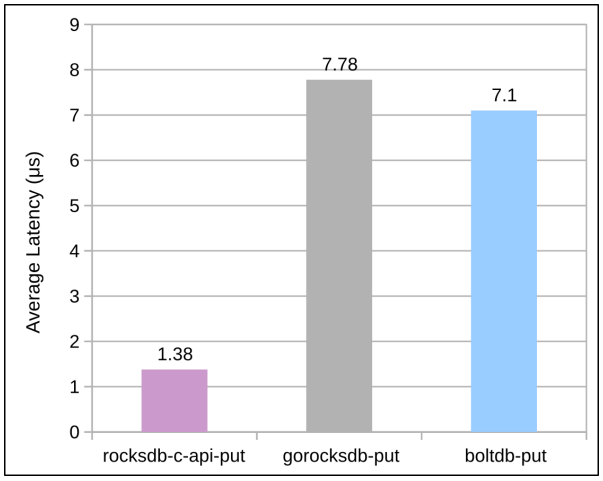
5.2.3 Η καθυστέρηση του cgo

Με βάση τα όσα εξηγήθηκαν στην υποενότητα 3.1.3, καθώς και τα ευρήματα του εργαλείου δημιουργίας προφίλ που παρουσιάστηκαν στην υποενότητα 4.3.2, έχουμε λόγους να υποψιαζόμαστε ότι η χρήση του εργαλείου cgo έχει εισαγάγει καθυστέρηση που δεν επιτρέπει στην υλοποίησή μας να επωφεληθεί πλήρως από τις δυνατότητες της RocksDB σε επίπεδο επίδοσης. Για την εξακρίβωση αυτής της υπόθεσης, πραγματοποιήσαμε πείραμα[15] κατά το οποίο 1000 πελάτες εισήγαγαν στον etcd 1000000 ζεύγη κλειδιού-τιμής και χρονομετρήσαμε τις ενέργειες εισαγωγής (put) σε τρία διαφορετικά σημεία στον κώδικα. Τα δύο από αυτά βρίσκονταν στην έκδοση `final` και ήταν η συνάρτηση `rocksdb_writebatch_put()` του C API της RocksDB και η συνάρτηση `WriteBatchPut()` της gorocksdb, η οποία χρησιμοποιεί το cgo για να καλέσει την `rocksdb_writebatch_put()`. Το τρίτο σημείο βρισκόταν στην έκδοση `original`, στη συνάρτηση `Put()` του πακέτου `bolt`, που επενεργεί στη δομή `Bucket`. Το Σχήμα 10 απεικονίζει τη μέση διάρκεια των τριών αυτών ενεργειών.

Είναι προφανές ότι η RocksDB διαθέτει ξεκάθαρο πλεονέκτημα σε σχέση με την BoltDB σε ό,τι αφορά τις ενέργειες εισαγωγής. Ωστόσο, το πλεονέκτημα αυτό επισκιάζεται από την εισαγόμενη από το cgo καθυστέρηση. Σαν αποτέλεσμα, μια εισαγωγή στην έκδοση `final` καταλήγει να είναι λίγο πιο χρονοβόρα από την αντίστοιχη ενέργεια στην έκδοση `original`.

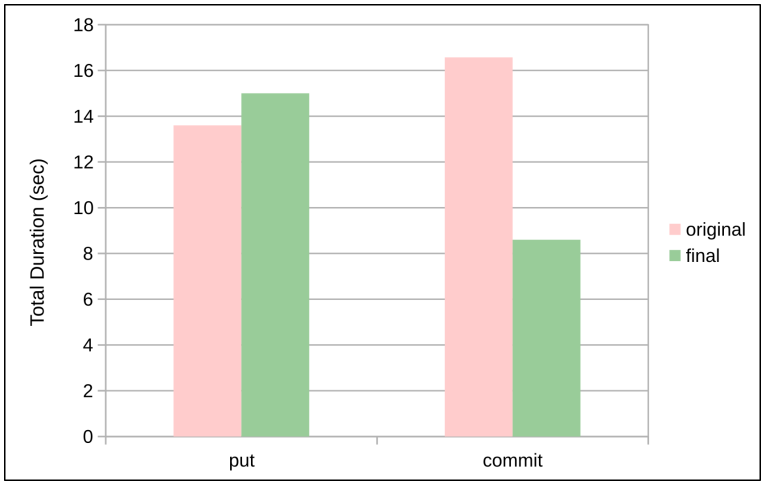
Το Σχήμα 11 δείχνει τον συνολικό χρόνο που δαπανήθηκε σε ενέργειες εισαγωγής (put) και κατοχύρωσης (commit) στις εκδόσεις `final` και `original` του etcd, κατά

[15] Αυτό το πείραμα, καθώς και το επόμενο, δεν διεξήχθησαν στη συστοιχία EC2, αλλά τοπικά, στο μηχάνημά μας με 4 πυρήνες, 4GB μνήμης, δίσκο SSD και λειτουργικό σύστημα Ubuntu.



Σχήμα 10: *Η επίδραση του κόστους του cgo στην καθυστέρηση εγγραφών*

την πραγματοποίηση πειράματος στο οποίο 1000 πελάτες εισήγαγαν 1000000 ζεύγη κλειδιού-τιμής. Τα δεδομένα αυτά αποκτήθηκαν μέσω των μετρικών χρόνου εκτέλεσης του etcd και της άθροισης των τιμών διάρκειας των ενεργειών εισαγωγής του προηγούμενου πειράματος.

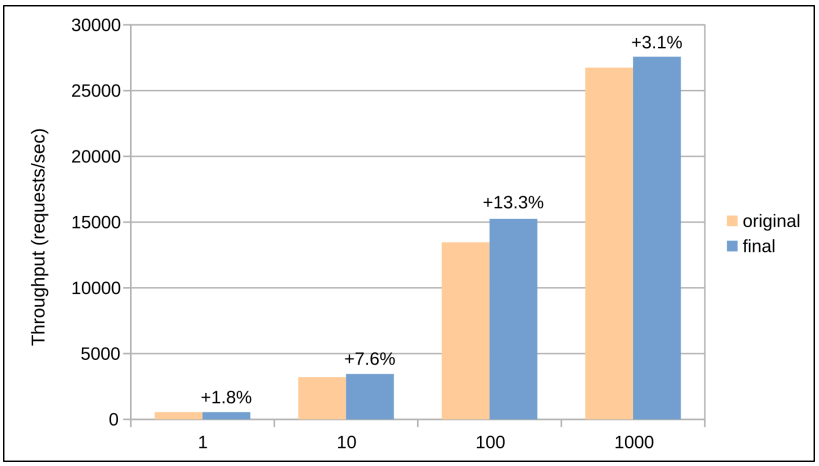


Σχήμα 11: *Συνολική διάρκεια λειτουργιών εγγραφής (put) και κατοχύρωσης (commit)*

Ο αντίκτυπος του κόστους του cgo στην επίδοση των ενεργειών κατοχύρωσης είναι πολύ μικρότερος από ό,τι στις ενέργειες εισαγωγής, καθώς η εισαγόμενη από το cgo καθυστέρηση ανά κλήση είναι αμελητέα σε σχέση με τη διάρκεια κλήσης της συνάρτησης κατοχύρωσης. Επίσης, το προτέρημα της δομής της RocksDB στον δίσκο αξιοποιείται κατά την κατοχύρωση, όταν τα αποθηκευμένα στον απομονωτή του `WriteBatch` ζεύγη κλειδιού-τιμής εγγράφονται στο LSM-δέντρο.

5.2.4 Επίδραση του αριθμού πελατών & του μεγέθους τιμής στην επίδοση

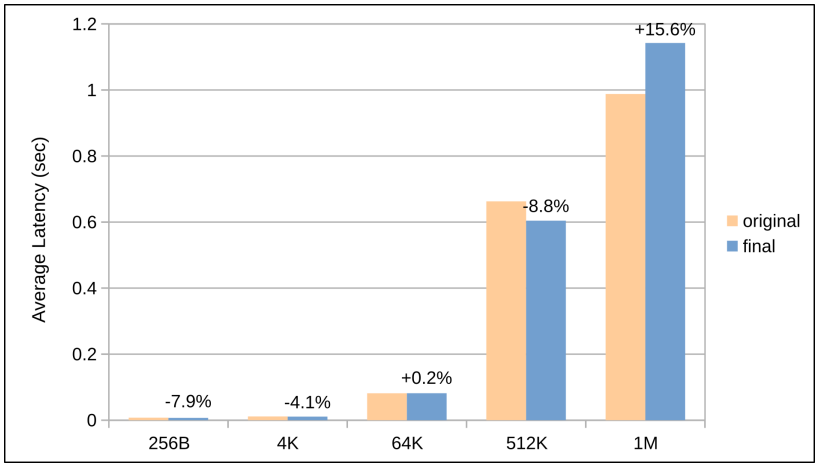
Στο επόμενο πείραμα εξετάζουμε την κλιμακωσιμότητα της υλοποίησής μας με τον αριθμό πελατών που στέλνουν ταυτόχρονα αιτήματα εγγραφής στον etcd και τη συγκρίνουμε με την κλιμακωσιμότητα της έκδοσης `original`. Κατά τη διάρκειά του, ο αριθμός των πελατών πήρε τις τιμές 1, 10, 100 και 1000, ενώ κάθε φορά εισάγονταν 100000 ζεύγη κλειδιού-τιμής. Τα αποτελέσματα παρουσιάζονται στο Σχήμα 12.



Σχήμα 12: *Διεκπεραιωτική ικανότητα εγγραφών συναρτήσει του αριθμού πελατών*

Ανεξαρτήτως αριθμού πελατών, η έκδοση `final` ξεπερνά σε επίδοση την έκδοση `original`. Ωστόσο, τα μεγαλύτερα ποσοστά βελτίωσης απαντώνται στις περιπτώσεις των 10 και των 100 πελατών. Για να ερμηνεύσουμε αυτήν την τάση, αρκεί να σκεφτούμε την πολιτική ομαδοποίησης (batching) του etcd. Όσο περισσότεροι πελάτες συμμετέχουν στο πείραμα, τόσο πιο αποδοτική γίνεται η ομαδοποίηση. Σε όλες τις περιπτώσεις ο αριθμός των ενεργειών εισαγωγής (put) παραμένει ο ίδιος, αλλά ο αριθμός των ενεργειών κατοχύρωσης (commit) είναι αντιστρόφως ανάλογος του αριθμού πελατών. Όσο περισσότερες είναι οι ενέργειες κατοχύρωσης, τόσο περισσότερο επωφελούμαστε από την υπεροχή της επίδοσης κατοχύρωσης της υλοποίησής μας, που παρουσιάστηκε στο Σχήμα 11. Εντούτοις, αυτή η γενική τάση δεν ισχύει στην περίπτωση που ο αριθμός πελατών είναι ίσος με 1, καθώς η διαφορά μεταξύ της διάρκειας των ενεργειών κατοχύρωσης ανάμεσα στις εκδόσεις `final` και `original` αποτελεί μικρότερο ποσοστό του συνολικού χρόνου ολοκλήρωσης του πειράματος σε σχέση με τις άλλες περιπτώσεις. Αυτό συμβαίνει λόγω της επίδρασης του μικρότερου βαθμού ταυτοχρονισμού στην ομαδοποίηση αιτημάτων στο επίπεδο του αρχείου καταγραφής του αλγορίθμου Raft, που γίνεται για τον διαμοιρασμό του κόστους της `fsync`.

Στο πείραμα που ακολουθεί εξετάζουμε την επίδραση του μεγέθους της τιμής στην επίδοση της έκδοσης `final` σε σύγκριση με την έκδοση `original`. 100 πελάτες εισάγουν ζεύγη κλειδιού-τιμής με μέγεθος κλειδιού 8 bytes στον etcd, ενώ ο αριθμός των κλειδιών και το μέγεθος των τιμών τίθενται κάθε φορά ίσα με ένα διαφορετικό ζευγάρι του συνόλου 256 και 1000000, 4000 και 64000, 64000 και 4000, 512000 και 500, 1000000 και 256. Τα αποτελέσματα μπορούν να παρατηρηθούν στο Σχήμα 13.



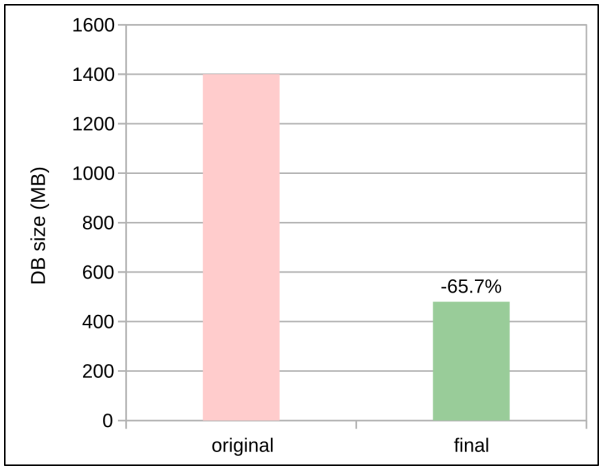
Σχήμα 13: *Μέση καθυστέρηση εγγραφών στον etcd συναρτήσει του μεγέθους τιμής*

Στις περισσότερες περιπτώσεις η έκδοση `final` ξεπερνά σε επίδοση την έκδοση `original`. Όταν όμως οι τιμές αποκτήσουν μέγεθος ίσο με 1MB, η BoltDB τις διαχειρίζεται πιο αποδοτικά. Στα LSM-δέντρα η εισαγωγή μεγάλων τιμών είναι πιθανό να πυροδοτήσει αλλεπάλληλες συμπυκνώσεις, προκαλώντας έτσι επιπρόσθετη καθυστέρηση. Στη συγκεκριμένη περίπτωση, το 1MB είναι το πλησιέστερο στο συνολικό μέγεθος των πινάκων μνήμης μέγεθος τιμής. Συνεπώς, προκαλεί εξαιρετικά συχνά τη μεταφορά τους στον δίσκο.

5.2.5 Κατανάλωση χώρου στον δίσκο

Στη συνέχεια, συγκρίνουμε το μέγεθος του backend στον δίσκο μεταξύ της έκδοσης `final` και της έκδοσης `original`, μετά την ολοκλήρωση της φόρτωσης του etcd με 4000000 ζεύγη κλειδιού-τιμής. Το αποτέλεσμα φαίνεται στο Σχήμα 14.

Όπως ήταν αναμενόμενο, αφού η RocksDB εφαρμόζει συμπίεση δεδομένων ενώ η BoltDB όχι, η κατανάλωση χώρου στον δίσκο είναι κατά $65,7\%$ μικρότερη στην έκδοση `final`. Επιπλέον, όπως εξηγήσαμε στην υποενότητα 2.2.3, η BoltDB εμφανί-

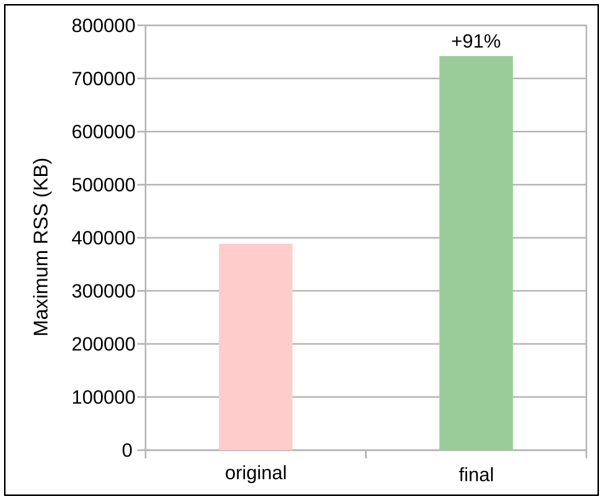


Σχήμα 14: *Μέγεθος της βάσης δεδομένων του backend του etcd στον δίσκο*

ζει ένα είδος εσωτερικού κατακερματισμού, δεσμεύοντας περισσότερο χώρο από ό,τι χρησιμοποιεί στην πραγματικότητα.

5.2.6 Κατανάλωση μνήμης

Εκκινώντας τον etcd με την εντολή `/usr/bin/time -v etcd` και εγγράφοντας σε αυτόν 1000000 ζεύγη κλειδιού-τιμής, λάβαμε το μέγιστο χώρο φυσικής μνήμης που κατέλαβε η διεργασία etcd κατά τη διάρκεια του πειράματος, στις εκδόσεις `final` και `original`. Το Σχήμα 15 απεικονίζει τα αποτελέσματα.



Σχήμα 15: *Κατανάλωση μνήμης στον etcd*

Παρατηρούμε ότι η έκδοση `original` καταναλώνει πολύ λιγότερη μνήμη από την έκδοση `final`. Το αποτέλεσμα αυτό ήταν αναμενόμενο, καθώς έχουμε ρυθμίσει τη RocksDB ώστε να χρησιμοποιεί πολλαπλούς πίνακες μνήμης μεγάλου μεγέθους.

6 Επίλογος

6.1 Συμπεράσματα

Συνολικά, η υλοποίησή μας κατάφερε να ανταποκριθεί στις προσδοκίες που διατυπώθηκαν στο κεφάλαιο 3. Σύμφωνα με τα αποτελέσματα της πειραματικής αξιολόγησης, επιτυγχάνει καλύτερη επίδοση εγγραφών και μικρότερη κατανάλωση χώρου στον δίσκο, χάνοντας όμως σε επίδοση αναγνώσεων και κατανάλωση μνήμης.

Συγκεκριμένα, η επίδοση αναγνώσεων, σε ό,τι αφορά τις αναγνώσεις μεμονωμένων κλειδιών, διατηρήθηκε σε αποδεκτά επίπεδα. Η επίδοση των αναγνώσεων εύρους κλειδιών ωστόσο, παρουσιάζει σημαντική επιδείνωση. Το ποσοστό βελτίωσης της επίδοσης εγγραφών κυμαίνεται μεταξύ του $1,8\%$ και του $13,3\%$, ανάλογα με τον αριθμό πελατών που στέλνουν αιτήματα ταυτόχρονα. Η κατανάλωση χώρου στον δίσκο υποχώρησε κατά περισσότερο από 50%, ενώ η κατανάλωση μνήμης σχεδόν διπλασιάστηκε. Επίσης, η πλατφόρμα ελέγχου του etcd επαλήθευσε την τήρηση των εγγυήσεών αξιοπιστίας, συνέπειας και υψηλής διαθεσιμότητάς του μετά την αλλαγή αποθηκευτικού μηχανισμού που εφαρμόσαμε. Επιπρόσθετη θετική επίπτωση της υλοποίησής μας αποτέλεσε η εξάλειψη της ανάγκης τακτικής ανασυγκρότησης του backend και η άρση του περιορισμού αποθηκευτικού χώρου.

Η εργασία αυτή κατέδειξε ότι μια προσέγγιση βασισμένη στα LSM-δέντρα μπορεί να επηρεάσει καθοριστικά την καταλληλότητα του etcd για φόρτο εργασίας που χαρακτηρίζεται από συχνές εγγραφές. Ωστόσο, το κόστος χρήσης του cgo αποδείχθηκε κάθε άλλο παρά αμελητέο και περιόρισε την επίδοση του συστήματός μας.

6.2 Μελλοντικές Δυνατότητες

Μια πολλά υποσχόμενη εναλλακτική υλοποίηση θα αντικαθιστούσε τη RocksDB με έναν γραμμένο σε γλώσσα Go αποθηκευτικό μηχανισμό βασισμένο στα LSM-δέντρα. Το Badger [116], ένα νέο και αρκετά δημοφιλές ενσωματωμένο σύστημα αποθήκευσης κλειδιού-τιμής βασισμένο στα LSM-δέντρα, προέκυψε ακριβώς από την αναγκαιότητα μιας αποδοτικής λύσης αντικατάστασης της RocksDB για συστήματα λογισμικού γραμμένα σε Go. Το κυρίαρχο κίνητρο των σχεδιαστών του ήταν η ανάγκη αποφυγής

του κόστους και της πολυπλοκότητας του εργαλείου cgo. Ο σχεδιασμός του βασίζεται σε μια δημοσίευση του L. Lu κ.α. που έγινε το 2016 [117], η οποία προτείνει τον διαχωρισμό των κλειδιών από τις τιμές. Ειδικότερα, οι τιμές αποθηκεύονται σε ένα αρχείο προεγγραφών, που ονομάζεται αρχείο καταγραφής τιμών (value log), ενώ τα κλειδιά αποθηκεύονται στο LSM-δέντρο μαζί με δείκτες στις τιμές που τους αντιστοιχούν. Με αυτόν τον τρόπο, ελαχιστοποιείται η ενίσχυση αναγνώσεων και εγγραφών. Καθώς τα κλειδιά τείνουν να έχουν μικρότερο μέγεθος από τις τιμές, το παραγόμενο LSM-δέντρο είναι επίσης πολύ μικρότερο. Κατά συνέπεια, το κόστος των συμπυκνώσεων ελαττώνεται. Επίσης, τα επίπεδα του LSM-δέντρου είναι λιγότερα, γεγονός που συμβάλλει στη μείωση των απαιτούμενων προσβάσεων σε αρχεία για την εύρεση ενός κλειδιού στη χειρότερη περίπτωση [118].

Επίσης, θα παρουσίαζε ενδιαφέρον η διεξαγωγή ενός πειράματος στο οποίο το σύνολο των δεδομένων θα ξεπερνούσε σε μέγεθος τη διαθέσιμη μνήμη, με σκοπό την ανάδειξη της υπεροχής του LSM-δέντρου στη διαχείριση αυτής της περίπτωσης. Όπως έχουμε περιγράψει στην υποενότητα 2.3.5, η BoltDB θα αναγκαζόταν να διαβάζει διαρκώς σελίδες από τον δίσκο προκειμένου να εξυπηρετήσει τα εισερχόμενα αιτήματα. Μια ακόμη πρόταση, στο πνεύμα της πλήρους αξιοποίησης της τρέχουσας υλοποίησής μας, αφορά την αυτοματοποίηση της διαδικασίας βέλτιστης ρύθμισης παραμέτρων της RocksDB μέσω κατάλληλου script. Επιπλέον, ένας τρόπος να μετριάσουμε το κόστος του cgo θα ήταν η προσθήκη ενός επιπέδου κώδικα σε Go που θα λειτουργούσε ως απομονωτής για τη διατήρηση των ενημερώσεων που κατευθύνονται στο `WriteBatch` και θα τις εφάρμοζε όλες μαζί κατά την κατοχύρωση, αντικαθιστώντας έτσι χιλιάδες κλήσεις cgo με μόνο μία.

Μια επιπρόσθετη ιδέα είναι η εφαρμογή κατάτμησης (sharding) του χώρου κλειδιών στο επίπεδο της RocksDB, όπως προτείνεται στον οδηγό συντονισμού της [64], με σκοπό την πλήρη εκμετάλλευση του ταυτοχρονισμού της τεχνολογίας αποθήκευσης από τις συμπυκνώσεις. Προχωρώντας ένα βήμα παραπέρα, αποθηκεύοντας τα τμήματα (shards) σε διαφορετικούς δίσκους θα μπορούσαμε να διαμοιράσουμε σε αυτούς τις εγγραφές, αυξάνοντας σημαντικά τη διεκπεραιωτική ικανότητα του etcd. Τέλος, θα μπορούσαμε να τροποποιήσουμε τον etcd ώστε να υποστηρίζει πολλαπλούς ηγέτες, κάθε ένας εκ των οποίων θα είναι υπεύθυνος για ένα διαφορετικό τμήμα.

1 Introduction

In this chapter, we outline the scope of our work. We first provide a quick overview of the problem at hand. Next, we illustrate our proposed solution and mention some types of applications that can benefit from it. We move on to briefly describe some existing solutions and their shortcomings. Finally, we present the structure of the document.

1.1 Problem Statement

Data storage has always played a fundamental role in computing. Unlike centralized storage, distributed storage is scalable, highly-available and fault-tolerant. Nowadays, *distributed storage systems* not only provide a mere means to store data remotely, but also offer innovative services like peer-to-peer file sharing, online backup solutions, distributed file systems, etc.

A *key-value store* is a type of NoSQL database that internally uses an associative array to store data in the form of key-value pairs. Key-value stores tend to be more scalable and expose a simpler API[1] than relational databases.

Etcd is a distributed key-value store for the most critical data of a distributed system [1]. Since it was designed for the purpose of storing shared configuration for large-scale clusters, and for similar use cases where write requests are expected to be scarcer than read requests, it favours the performance of the latter over that of the former by design. A multitude of factors determine the performance of etcd. In this work we

[1] Application Programming Interface

focus on one of them: the *storage engine*. Currently, to persist key-value pairs to disk etcd uses *BoltDB*, which implements a read-optimized data structure called *B+ tree*.

The above might become an inhibitory factor for applications that want to integrate with etcd but exhibit mainly *write-intensive* workloads. In the age of Big Data, applications of this type are becoming increasingly common. Thus, the primary objective of this thesis is the optimization of etcd for write-heavy workloads via replacement of its current storage engine. *RocksDB*, a state-of-the-art local key-value store, was deemed to be an ideal fit for this case. It is based on an *LSM-tree*[2], which is an inherently write-optimized data structure.

In the next chapters, we thoroughly describe the design and implementation of the storage engine transition and present the obtained results.

1.2 Motivation

In this section we highlight the value of our contribution by mentioning specific types of applications that could benefit from it. In other words, we construct an indicative list of example applications that could leverage a distributed storage system like etcd and whose workload is known to be write-intensive.

- **Data collection systems:** data collection is of great significance for scientific research and everyday life. Sensor data collection, online surveys and traffic monitoring are concrete examples of this application category. They all involve voluminous amounts of incoming data that has to be efficiently loaded and reliably stored for subsequent analysis.

- **Online backup solutions:** they are characterized by frequent write operations aiming to keep the backed up data up to date with their local copy, while reads are only issued in case the local copy is damaged.

- **Mail servers:** The I/O operation mix that typically takes place in email message stores is reported to be write-heavy [2], [3]. Adequate write throughput is therefore essential for the provision of an acceptable quality of service.

[2]Log-Structured Merge-Tree

- **Social media websites:** these are yet another big data system whose ability to sustain a high write throughput is crucial.

- **Interactive games:** Maintaining player activity history entails large amounts of writes; often every single move has to be recorded.

- **Distributed logging:** various types of software employ event logs for purposes ranging from security management to debugging and user behaviour analysis [4]. If the insertion of new log records is not handled efficiently enough we run the risk of hampering the performance of the main application.

- **Online transaction processing systems:** examples include order processing, airline reservations, payroll and financial transaction systems. These are all scenarios with write-intensive workloads.

The following excerpt from the etcd developers' blog confirms that our effort is in the right direction: "The ideal key-value store ingests many keys per second, quickly persists and acknowledges each write, and holds lots of data. If a store can't keep up with writes then requests will time-out, possibly triggering failovers and downtime. If writes are slow then applications appear sluggish. With too much data, a store may crawl or even be rendered inoperable" [5].

What is more, Yahoo, reports a steady progression from read-heavy to read-write workloads, driven largely by the increased ingestion of event logs and mobile data [6], [7].

1.3 Existing Solutions

Our implementation is not the first one that integrates an LSM-tree-based storage engine into a distributed storage system. In this section we briefly describe other such approaches and mention their similarities and differences from our own.

1.3.1 CockroachDB

CockroachDB is a distributed SQL[3] database built on a transactional and strongly-consistent key-value store. It is developed by Cockroach Labs and can be used for building global, scalable cloud services that survive disasters [8]. It is a highly-available CP[4] system like etcd.

In the architecture of CockroachDB, the highest level of abstraction is the SQL layer. It connects to the underlying key-value store by constructing prefixes from the SQL table identifier, followed by the value of the primary key for each row. This way it manages to combine the rich functionality of SQL with the scalability common to NoSQL stores.

A typical setup consists of a few nodes. Nodes contain one or more stores. Each store should be placed on a unique disk and internally contains a single instance of RocksDB.

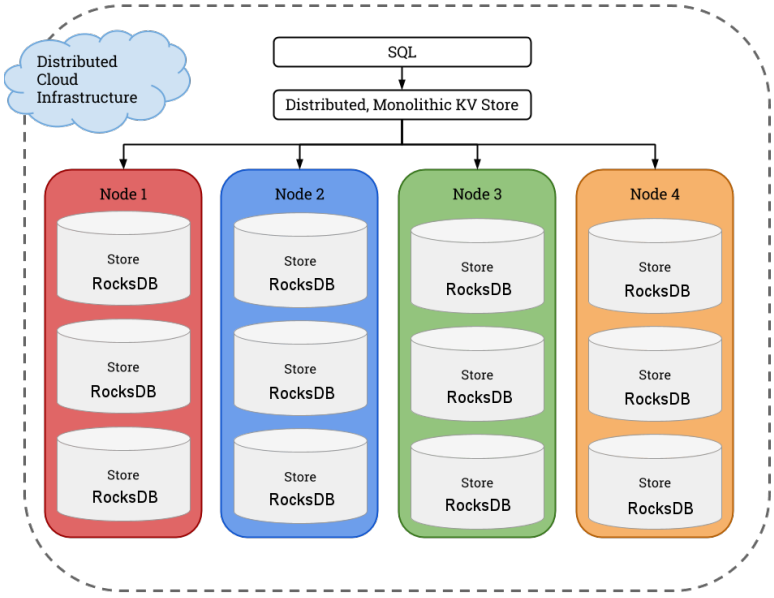


Figure 1.1: *Overview of CockroachDB architecture [9]*

CockroachDB scales horizontally, by dividing the key-value space in ranges and storing each range in a local instance of RocksDB. This technique is called *sharding*. Each range is replicated to a total of three or more CockroachDB servers. In order to ensure consistency between data ranges the *Raft* consensus protocol is used, just like in etcd.

[3]Structured Query Language
[4]Consistent and partition tolerant

In fact, those two projects share the same implementation of the Raft algorithm[5].

As data flows in, existing ranges split into new ones, aiming to keep a relatively small and consistent range size. Newly split ranges are automatically rebalanced to nodes with more capacity [8].

Compared to etcd, CockroachDB seems like a more ambitious project. Indeed, not only does it support automated load-balancing and failover, but also its support for horizontal scaling by partitioning data across multiple nodes, enables it to reliably handle database sizes of terabytes and above. Etcd on the other hand lacks keyspace sharding and is limited to several gigabytes. Adding more nodes to an etcd cluster, enforces its fault-tolerance, but does not increase its capacity, as is the case with CockroachDB.

However, sharding of data across multiple consistent replication groups introduces the need for a coordination protocol among them, which in turn induces longer latencies than those of etcd, which supports only a single replication group.

What is more, CockroachDB possesses an external SQL API with richer semantics than etcd's API, but at the cost of additional complexity for processing, planning, and optimizing queries [10]. Despite their similarities, each software has a different target group. Etcd is more appealing to users who need a simple, lightweight, flat key-value store, free from unnecessary overheads. Finally, etcd having been used in production for a longer period, is a more mature project than CockroachDB and already has a broad established user base.

1.3.2 TiKV

TiKV is yet another distributed key-value database that relies on the Raft algorithm for consensus and stores its data on RocksDB instances. This transactional and consistent store is developed by PingCAP and written in Rust.

TiKV provides horizontal scalability by following a sharding scheme very similar to that of CockroachDB. In this case, the partition unit is called a region. A key component of this key-value store is the Placement Driver, which manages region replication, stores metadata such as the region location of specific keys, and performs automatic load-balancing [11]. The Placement Driver is in fact an etcd cluster.

[5] `https://github.com/coreos/etcd/tree/master/raft`

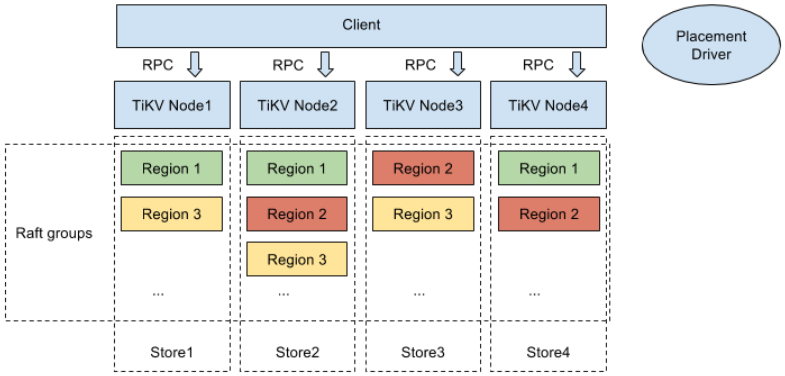


Figure 1.2: *Sharding and region replication in TiKV [11]*

TiKV is actually a component of TiDB[6], a distributed SQL database whose layered architecture bears a notable likeness to that of CockroachDB. TiDB works as the SQL layer and TiKV works as the key-value layer [12]. Even though it is possible to use TiKV as a standalone key-value store, the completeness of its API is nowhere near that of etcd.

1.3.3 Other Related Work

In this subsection we mention two more distributed storage projects and the ways in which they attempt to optimize their write performance.

MongoDB

MongoDB[7] is an open-source, distributed NoSQL database. It is horizontally scalable, consistent and stores data as documents in a binary representation. Just like the systems previously described it harnesses the innovations of NoSQL, while maintaining the semantic richness of relational databases.

Through the use of a *pluggable storage engine* architecture, it allows the user to select the most suitable storage engine based on their expected workload [13]. Currently, the default storage engine is *WiredTiger*[8]. It is a high-performance, transactional, NoSQL storage engine that offers a choice between B-tree and LSM-tree data structures. RocksDB is also a common option [14].

[6] `https://github.com/pingcap/tidb`
[7] `https://github.com/mongodb/mongo`
[8] `https://github.com/wiredtiger/wiredtiger`

MyCassandra

Cassandra[9] is another open-source, distributed NoSQL database with a custom query language interface, developed by the Apache Software Foundation. It is scalable, highly-available, decentralized and has an impressive user base. Internally it uses a storage structure similar to an LSM-Tree.

MyCassandra is an interesting effort to make a cloud storage system simultaneously read and write-optimized via adaptation of a modular design, analogous to that of MongoDB. The novelty of this approach resides in its proposal of a heterogeneous cluster, built from MyCassandra nodes with different storage engines. In this architecture, a proxy receives incoming requests and depending on their type (read or write query) routes them synchronously to nodes optimized for this query type, and asynchronously to the rest. To maintain consistency among replicated data a quorum protocol is employed. In an example cluster of three nodes, this introduces the need for at least one of them to perform well with both types of workloads. An in-memory storage engine node satisfies this requirement [15].

For example, a write query is synchronously routed to write-optimized and on-memory storage engine nodes and is asynchronously routed to a read-optimized storage engine node, as shown in Figure 1.3. .

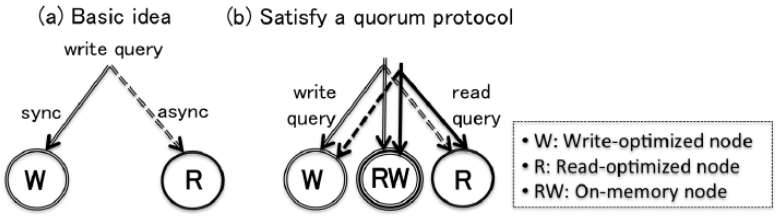


Figure 1.3: *Basic concept of a heterogeneous MyCassandra cluster [15]*

It is worth noting again that the aforementioned projects are far from being simple distributed key-value stores, as lightweight as etcd. Other interesting, distributed, plain key-value stores include project Voldemort[10], MemcacheDB[11]and Scalaris[12], but as far as we know there are none that are both persistent and write-optimized, with the ex-

[9] https://github.com/apache/cassandra
[10] https://github.com/voldemort/voldemort
[11] https://github.com/stvchu/memcachedb
[12] https://github.com/scalaris-team/scalaris

eption of Riak KV. Even in the case of Riak KV[13] though, or in cases that it is trivial to integrate new persistence mechanisms into those key-value stores, they happen to be fundamentally different than etcd regarding other important aspects, such as their consistency guarantees or the functionality exposed by their API.

1.4 Thesis Structure

The rest of the document is organized as follows:

- **Chapter 2:** presentation of the theoretical background and concepts that our work is founded upon.
- **Chapter 3:** analysis of the architecture of our solution and design decisions from a higher-level perspective.
- **Chapter 4:** demonstration of the focal points of our implementation, reference to the problems we faced during the development process, as well as the proposed workarounds, optimizations and testing.
- **Chapter 5:** experimental evaluation of our solution.
- **Chapter 6:** concluding remarks, suggested future improvements and alternative approaches.

[13] `https://github.com/basho/riak_kv`

2 Background

In this chapter we provide the key theoretical elements for the understanding of our work. First, we explain several fundamental principles from the area of distributed systems and storage systems in general. We move on to describe the architecture and functionality of etcd, the main software component of this project. Then, follows an analysis of BoltDB, the current storage engine of etcd and its proposed replacement, RocksDB. The analysis includes their respective on-disk data structures. This chapter concludes with a brief presentation of Go, the programming language in which etcd, BoltDB and our contribution are written.

2.1 Distributed Systems & Data Storage Concepts

2.1.1 An Overview of Distributed Storage

According to Wikipedia, "a *distributed data store* is a computer network where information is stored on more than one node, often in a replicated fashion. It is usually specifically used to refer to either a distributed database where users store information on a number of nodes, or a computer network in which users store information on a number of peer network nodes" [16].

A paradigm shift: from centralized to distributed

In the era of Big Data, Web 2.0, and the Internet of Things (IoT) storage systems are required to manage a huge amount of digital data which is created daily and accumulates to unprecedented amounts. A tipping point has been reached, at which the traditional approach of using a stand-alone storage box no longer works [17].

A distributed approach to storage has emerged to face the challenges that cannot be met by centralized systems. Distributed data stores, unlike their centralized counterparts, have an unmatched ability to *scale horizontally* (scale out), by adding more nodes to the system. Their **scalability** results in improved performance, as requests can be processed by many machines, instead of being limited to one. Moreover, when sharding is applied, the addition of extra nodes to a distributed storage network increases its capacity.

Scalability is not the only factor that has lead to this paradigm shift from centralized to distributed. Centralized systems expose by nature a single point of failure. If the central storage node is damaged, e.g., experiences a disk failure, then data will be irrevocably lost. There are applications though, for which reliability is crucial, introducing the need for data *replication*. Distributed solutions ensure that there exist always consistent copies of the data on different nodes, a fact that makes them **fault-tolerant**. In this case, even if one or more nodes fail, the data can be retrieved from another. Also, while repairing the failure, e.g., by replacing the failed node with a healthy one, the system remains capable of servicing requests. It is in other words **highly available**.

Another reason for the prevalence of distributed data stores lies in their **cost effectiveness**. Scaling out usually involves the addition of inexpensive, commodity hardware. On the other hand, centralized solutions in an attempt to handle ever-increasing amounts of data, resort to *vertical scaling* (scaling up). This means that they add resources to a single node, typically involving the addition of CPUs[1], memory or disks to a single computer. This requires the constant provision of new and often expensive hardware. For example, an increase in the number of CPUs of a machine may require a motherboard upgrade and introduce higher cooling and power requirements. Likewise, adding more disk trays might also require an extra storage controller. Today, system architects often prefer to configure tens or hundreds of low-cost computers

[1]Central Processing Units

into clusters to obtain aggregate performance that may supersede even that of a supercomputer, whose cost is prohibitively high [18]. In addition to that, scaling vertically usually requires downtime, while scaling horizontally can be done on-the-fly.

However, there are cases where centralized systems are still the preferred option due to their simpler programming design, security and smaller administrative overhead.

2.1.2 Key-Value Stores

A *key-value store* is a type of database that can store, retrieve and update information in the form of key-value pairs. The key is used to uniquely identify the value, which contains the actual data. Internally, this relationship is implemented via an associative array, more commonly known as a dictionary or hash. The key is represented by an arbitrary string, while the value can be any kind of data, e.g., an image, a document etc.

In general, key-value stores do not have a query language. Data management is done via simple get, put and delete commands. The *simplicity* of this model makes them *fast*, easy to use, scalable, portable and flexible [19]. Like all databases, they can run locally or in a distributed environment. Distributed key-value stores have been gaining a lot of popularity lately, as part of the broader NoSQL movement.

NoSQL versus relational databases

NoSQL databases provide a mechanism for storage and retrieval of data that is modelled in means other than the tabular relations used in relational databases. As it is descriptively explained in a MongoDB article on NoSQL databases [20], "in the case of *relational* databases individual records (e.g., "employees") are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., "manager", "date hired", etc.), much like a spreadsheet. Related data is stored in separate tables, and then joined together when more complex queries are executed. For example, "offices" might be stored in one table, and "employees" in another. When a user wants to find the work address of an employee, the database engine joins the "employee" and "office" tables together to get all the information necessary".

Furthermore, to access data stored in relational databases a structured query language

using select, insert, and update statements has to be used. NoSQL stores are accessed through object-oriented APIs. Relational databases were mainly developed in 1970s to deal with first wave of data storage applications, while NoSQL databases started being developed around the late 2000s to deal with the limitations of the former.

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address. Specifically, relational databases lack *agility*, as they need to know the type of data that is going to be stored in them in advance. Also, each time new features are added the database schema needs to change. NoSQL databases are free from these restrictions because they are built to allow the insertion of data without a predefined schema.

Relational databases do not have native support for sharding and replication, which makes their operation in distributed environments a very complex task that requires the development of additional application code. In other words, they do not have the inherent ability to *scale horizontally*. NoSQL databases, on the other hand, usually support auto-sharding and auto-replication, meaning that they natively and automatically spread data across an arbitrary number of servers, maintaining *availability* and providing *low latency* [20].

The basic NoSQL database types include key-value stores, which we have already described, and the following three:

- **Document databases:** they pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents and are often encoded in XML, YAML, JSON or BSON[2].

- **Graph stores:** this kind of database is designed for data whose relations are well represented as a graph consisting of elements interconnected with a finite number of relations between them. The type of data could be social relations, public transport links, road maps or network topologies [21].

- **Column stores:** data is stored in cells grouped in columns of data rather than as

[2]Extensible Markup Language, Yet Another Markup Language, JavaScript Object Notation, Binary JSON

rows of data. Columns are logically grouped into column families. The following example by G. Kumar illustrates the advantage of this type: "querying the titles from a bunch of a million articles would be a painstaking task while using relational databases as it would go over each location to get item titles. On the other hand, when using NoSQL the titles of all the items can be obtained with just one disk access" [22].

Obviously, key-value stores are the simplest among NoSQL types.

Use cases and notable implementations

The simplicity, low latency and flexibility of key-value stores render them ideal for numerous use cases. Among the most popular is *personalization* (storing user preferences and profiles) and *session management* (in web applications, mobile applications and multi-player online games).

The following are examples of prevalent key-value stores ordered by popularity, as reported in the monthly DB-Engines *Ranking of Key-value Stores* [23] for August 2017: Redis[3], Memcached, Apache Cassandra, Riak KV, BerkeleyDB[4], LevelDB, RocksDB, WiredTiger, TokyoCabinet[5], Scalaris, Project Voldemort etc.

2.1.3 CAP Theorem

The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:

- **Consistency:** any read operation that begins after a write operation completes must return that written value, or the result of a later write operation.
- **Availability:** every request received by a non-failing node in the system must result in a response (non-error or timeout) within a reasonable amount of time.
- **Partition Tolerance:** no set of failures less than total network failure is allowed to cause the system to stop functioning.

[3] https://github.com/antirez/redis
[4] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html
[5] http://fallabs.com/tokyocabinet/

A network partition is a damage to the network connecting the components of a distributed system, that results in messages sent from one node of the system to other nodes being delayed or dropped.

The CAP theorem was proposed by Eric Brewer in 2000. Seth Gilbert and Nancy Lynch published a formal proof [24] in 2002. The theorem has often been characterized as misleading, a common misconception being that a distributed system needs to pick two out of the three guarantees for the whole duration of its operation. But what it states is in fact that *in the presence of a network partition*, one has to choose between consistency and availability. In absence of network failure, that is, during normal operation of the distributed system, both availability and consistency can be satisfied [25].

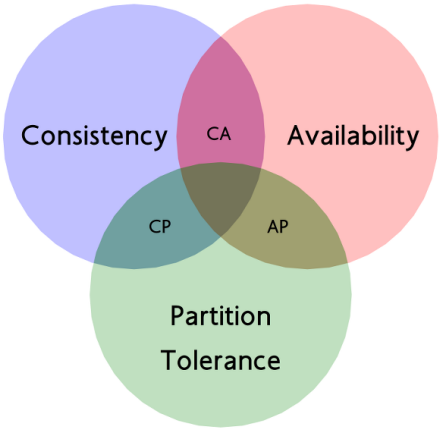


Figure 2.1: *Visualization of the CAP theorem [26]*

CP systems

If availability is not required, then it is easy to achieve consistency and partition tolerance. According to S.Gilbert and N. Lynch [24], "many distributed databases provide this type of guarantee, especially algorithms based on distributed locking or quorums: if certain failure patterns occur, then the liveness condition is weakened and the service no longer returns responses. If there are no failures, then liveness is guaranteed".

AP systems

It is possible to provide high availability and partition tolerance, if consistency is not required. The system will return the most recent available version, at the risk of returning stale data [24].

CA systems

If there are no partitions, it is clearly possible to provide consistent, available data [24]. However, as distributed systems are generally not safe from network partitions, their designers are forced to chose if they will belong in the CP or AP category.

Constructive criticism of the CAP theorem, as expressed by Martin Kleppmann in a 2015 article [27], states that the theorem's narrow definitions of consistency and availability fail to describe the variety of guarantees offered by modern distributed systems. Indeed, there are many widely used systems that cannot be classified neither as CP or AP.

2.1.4 ACID Properties

ACID is an acronym made up of a set of properties of database transactions[6], that allow the safe sharing of data: Atomicity, Consistency, Isolation and Durability. According to G.Kohad et al., these properties are defined as follows [28]:

- **Atomicity:** either all of the tasks of a transaction are performed or none of them are. Atomicity states that database modifications must follow an "all or nothing" rule. If some part of a transaction fails, then the entire transaction fails, and vice versa.
- **Consistency:** the database remains in a valid state, despite the transaction succeeding or failing and both before the start of the transaction and after the transaction is over.
- **Isolation:** other operations cannot access or see the data in an intermediate state during a transaction. The Isolation property helps implement concurrency of database.
- **Durability:** once a transaction is committed, its effects are guaranteed to persist even in the event of subsequent failures.

At this point it is worth noting that consistency is defined here in a different way than within the context of the CAP theorem.

[6]A transaction is a sequence of requests that is treated as a single unit.

2.1.5 Consistency Models & Isolation Levels

The consistency model of a distributed system, whether strong or weak, is a significant design choice. In this section we limit the discussion to the models directly relevant to the subject of our thesis.

Strict consistency or linearizability

This is the strongest consistency guarantee. The following is an informal definition by M. Kleppmann [27]: "if operation B started after operation A successfully completed, then operation B must see the the system in the same state as it was on completion of operation A, or a newer state". This is the same notion as consistency in the context of the CAP theorem, also known as atomic consistency.

Sequential consistency

This model is weaker than strict consistency and was first proposed by Leslie Lamport in 1979 [29]. According to him, "the result of any execution is the same as if the operations by all processors were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program".

What this definition means is that when processes run concurrently on (possibly) different machines, any valid interleaving of read and write operations is acceptable behaviour, but *all processes see the same interleaving of operations* [30]. Thus, sequential consistency offers a guarantee of ordering (also offered by strict consistency) rather than recency. *Serializability* is a another term often used in literature to describe this consistency model.

Eventual consistency

This consistency model defines that if no update takes place for a long time, all replicas eventually become consistent [30]. The only requirement of eventual consistency is that updates propagate to all replicas. If this requirement is met, in the absence of updates, all replicas will converge.

Read-your-writes consistency

A data store is said to provide read-your-writes consistency if the effect of a write operation by a process on a data item x will always be seen by a successive read operation on x *by the same process* [30].

Other consistency types include causal consistency, processor consistency, PRAM[7] consistency and continuous consistency.

Isolation, like consistency, has a variety of levels. A lower isolation level makes concurrent access to data by multiple users easier, but does not offer protection against undesirable concurrency effects, such as dirty reads and lost updates. On the other hand, a higher isolation level eliminates concurrency effects, but consumes more system resources and causes transactions to often block one another [31].

Serializable isolation

This is the highest isolation level and it guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation. It does this by performing locking [31]. This is the same notion as the "I" in ACID. This isolation level provides complete protection from concurrency effects.

Repeatable reads

In this isolation level, read and write locks are kept until the end of a transaction. However, range-locks are not managed, so *phantom reads*[8] can occur.

Other isolation levels include read committed isolation and read uncommitted isolation.

[7] Pipelined Random Access Memory

[8] A phantom read occurs when the same query executes twice in a transaction, and the second result set includes rows that were not visible in the first result set. This situation is caused by another transaction inserting new rows between the execution of the two queries.

2.1.6 Write-Ahead Logging

Write-ahead logging is a technique used in databases to provide atomicity and durability (two of the ACID properties). When using this technique, all changes are first written to the write-ahead log (WAL), which resides on stable storage, and then they are applied to the database.

Providing atomicity

To illustrate how the WAL helps achieve atomicity we borrow an example scenario from Wikipedia [32]: "imagine a program that is in the middle of performing some operation when the machine it is running on loses power. Upon restart, that program might well need to know whether the operation it was performing succeeded, half-succeeded, or failed. If a write-ahead log is used, the program can check this log and compare what it was supposed to be doing when it unexpectedly lost power to what was actually done. On the basis of this comparison, the program could decide to undo what it had started, complete what it had started, or keep things as they are".

Apparently, in some cases the WAL also ensures *consistency*. For example, if in the above scenario there is a database constraint according to which variables x and y must always have the same value, the operation is a transaction containing the commands `set x to 5` and `set y to 5` and the power outage happens just after the execution of the first command.

Providing durability

When using a log our database need not flush data pages to disk every time a transaction is committed. In the event of a crash the changes that may be lost can be replayed from the WAL. This has an added benefit: it reduces the number of disk writes. The WAL can also ensure durability for in-memory databases.

Additionally, the WAL makes it possible to support point-in-time recovery. Log *compaction* is discussed in subsection 2.1.8 within the context of Raft consensus algorithm and is the same for the general case. Furthermore, logs play a vital role in the performance of a database. Optimizations such as batching entries or storing the log on a

dedicated disk to avoid competition between logging and other I/O operations can make a considerable difference.

2.1.7 Multi-Version Concurrency Control

What is concurrency control?

In the words of Philip A. Bernstein and Nathan Goodman, "concurrency control is the activity of synchronizing operations issued by concurrently executing programs on a shared database. The goal is to produce an execution that has the same effect as a serial (non-interleaved) one" [33].

Concurrency control is necessary in the presence of transaction concurrency in order to provide *isolation* that will prevent problems like:

- **Lost updates:** these happen when two queries access and update the same data item in a database. Then, the first of these updates is lost for other concurrent transactions that need to read its value for their correct execution.
- **Dirty reads:** these occur when a transaction updates a database item and then it is rolled back. The updated item is accessed by another transaction before it is reverted to its original value.
- **Incorrect summaries:** if a transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the result will not be correct as it will depend on the timing of the updates (some will be included and some not).

There are two main categories of concurrency control mechanisms:

- **Optimistic:** conflicts are assumed to be rare. Optimistic concurrency control mechanisms allow concurrent transactions to proceed without blocking any of their operations and only check for violations at commit time. Then, if a conflict is found the transaction is rolled back. This approach favours throughput when conflicts are indeed rare ; if they are not though, aborted transactions need to be restarted, which involves an extra overhead.

- **Pessimistic:** conflicts are assumed to be frequent. This category of concurrency control mechanisms uses locking or a time-stamping technique to detect conflicts from the beginning of a transaction. Operations that cause violations are blocked until it is safe for them to be executed. This incurs delays but is usually a good strategy in high-contention environments, as the cost of protecting data is less than the cost of rolling back and restarting transactions.

Common mechanisms for concurrency control include locking[9], timestamp ordering[10] and multi-version concurrency control (*MVCC*), which is described below.

MVCC

The central concept of MVCC is summarized descriptively by Wikipedia: "when an MVCC database needs to update an item of data, it will not overwrite the old data with new data, but instead mark the old data as obsolete and add the newer version elsewhere. Thus, there are multiple versions stored, but only one is the latest" [34].

MVCC provides point-in-time consistent views of the database (snapshot isolation). The main advantage of the MVCC technique is that *readers never block writers* and vice versa. While a write transaction is in progress, readers can still access the previous version of the data. Consequently, a read-only transaction never needs to wait and in fact, does not have to use locking at all. Adoption of MVCC has to be accompanied by a mechanism that compacts history (i.e., removes obsolete versions).

Apart from its contribution to concurrency control, access to a revision history of the data is useful in itself for some types of applications. MVCC is supported by MySQL[11], PostgreSQL[12], SAP HANA[13], BerkeleyDB and Cassandra among many other databases.

[9] An operation cannot read or write data until it acquires an appropriate lock on it.

[10] Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction.

[11] `https://www.mysql.com/`

[12] `https://www.postgresql.org/`

[13] `https://www.sap.com/products/hana.html`

2.1.8 The Raft Consensus Algorithm

What is consensus?

State-machine replication, the technique used to ensure fault-tolerance in distributed data stores, introduces the need for consensus. *Consensus* is the process of agreeing on the values of the stored data among different nodes of a distributed system. The problem of maintaining consensus becomes more complex when failures occur, either on the nodes or on the network they use to communicate.

Replicated state machines are typically implemented using a *replicated log*. The log is stored on each server of a cluster and contains a series of commands, which the server's state machine executes in order. Consensus algorithms make sure that the replicated log is kept consistent (contains the same commands in the same order) across all servers [35].

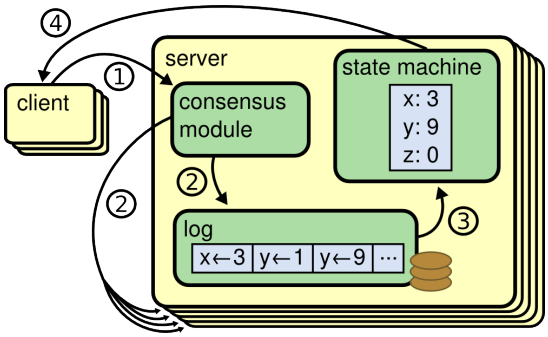


Figure 2.2: *Replicated state machine architecture [35]*

The Raft protocol

The Raft algorithm was introduced by Diego Ongaro and John Ousterhout in 2013. It is a consensus algorithm for managing a replicated log. Consensus is achieved by electing a leader who has then full responsibility for the management of the replicated log. The leader accepts log entries from clients, replicates them on the other servers of the cluster, and tells servers when it is safe to apply log entries to their state machines. In case a leader fails or becomes disconnected from the other servers, a new one is elected.

At any given time, each server is in one of the following three states: *leader*, *follower*,

or *candidate*. Normally, there is exactly one leader and all of the other servers are followers. Followers do not issue requests on their own but simply respond to requests from leaders and candidates. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). The third state, candidate, is used for leader election [35].

A *term* is in Raft terminology a time unit of arbitrary length denoted by a term number and characterized by a certain election. Communication between the Raft servers is performed via RPCs[14] and involves two basic types of messages: RequestVote RPCs used by candidates to initiate an election and AppendEntries RPCs, used by leaders both for log replication and as a heartbeat[15] when they carry no log messages [35].

Raft makes progress as long as any majority of the cluster's servers is available. It decomposes the consensus problem into three relatively independent subproblems: *leader election*, *log replication* and *safety*. Each of them is analysed below:

- **Leader election:** servers begin in the follower state and remain in it as long as they receive heartbeat messages from a leader in a periodic fashion or RequestVote messages from a candidate. If a follower does not receive such communication during an *election timeout*, it becomes a candidate. Then, it may either win the election if it receives votes from the majority of the cluster members, thus becoming the leader, or be notified that someone else has won. Requiring a majority of votes ensures that only one leader can be elected at a time. Votes are granted in a first-come-first-served basis and a candidate always votes for itself. The election time out should be configured to be an order of magnitude greater than the round trip time for heartbeat messages to prevent followers from starting unnecessary elections.

  In case of a *split vote* (no candidate establishes a majority of votes) the election starts anew. To prevent split votes from occurring indefinitely Raft uses randomized election timeouts. This increases the possibility of only one server timing out at a time and winning the election before any other has timed out.

  Furthermore, conflicts between nodes that happen to be simultaneously in the leader state, or between a leader an a candidate are resolved based on who has

[14] Remote Procedure Calls

[15] A heartbeat is a periodic signal generated by hardware or software to indicate normal operation or achieve synchronization [36].

the largest term number [35]. For example, when a network partition happens and the leader at term x happens to be in the minority partition, a new leader with term $x+1$ will be elected among the nodes in the majority partition. When the partition is healed the old leader will recognize that there is a leader with a higher term number than its own and step down. Figure 2.3 contains a schematic representation of the leader election process.

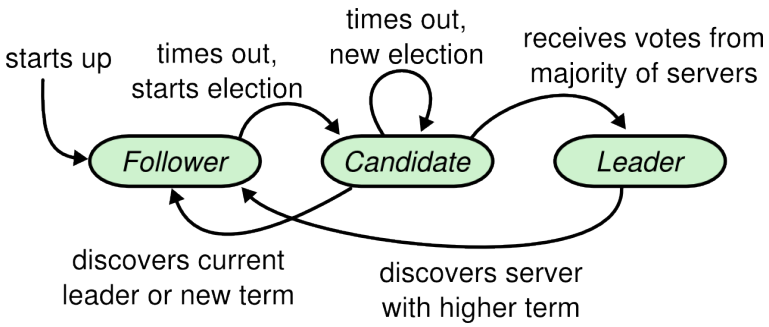


Figure 2.3: *The leader election process in the Raft consensus algorithm [35]*

- **Log replication:** the leader accepts client requests each of which contains a command to be executed by the replicated state machines. First, it appends the command to its log as a new entry. Then, it issues AppendEntries RPCs in parallel to each of the other servers in order to replicate the entry. Once the entry has been replicated on a majority of servers the leader applies it to its state machine and returns the result of that execution to the client. In Raft terminology the entry is now *committed*. The leader also notifies the other servers that the entry has been committed via AppendEntries RPCs so that they can apply it to their state machines as well. RPCs that are not received (e.g., because a follower has crashed) are resent indefinitely [35].

  The log structure is shown in Figure 2.4. In each log entry a command is stored along with the number of the term in which the entry was received by the leader. In addition, entries are numbered sequentially by the log index. The term number is used to recover from inconsistencies between logs. Inconsistencies may be caused by leader crashes and they are resolved by the new leader by forcing the followers to duplicate its log [35].

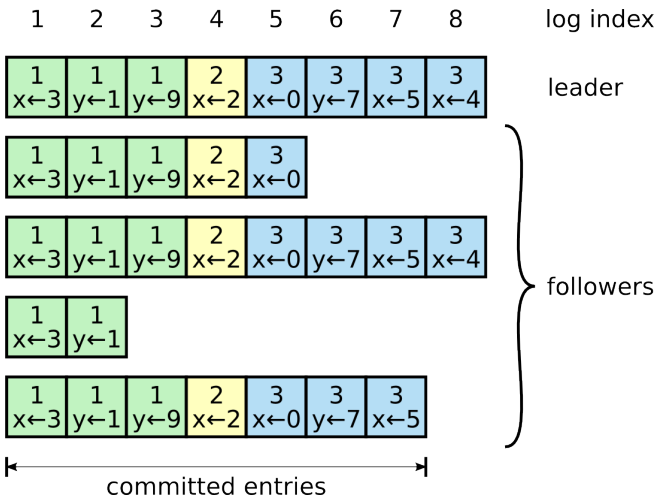


Figure 2.4: *Replicated Raft logs [35]*

- **Safety:** the essence of safety in the context of the Raft algorithm is that each state machine executes exactly the same commands in the same order[16]. To ensure that, Raft imposes a restriction according to which only a server whose log contains all committed entries up to the current term may be elected leader. The following is an illustrative example of the kind of situations this restriction helps avoid, as described by D. Ongaro and J. Ousterhout in their paper [35]: "a follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences". The restriction is implemented as part of the election process, by not allowing servers to vote for candidates whose log is not at least as up-to-date as their own.

Another interesting issue arises within the context of Raft with regard to the size of the on-disk log. As client requests flow in, the log size increases but apparently it cannot be allowed to grow without bound. This would cause it to occupy all available disk space and log replay time would become unacceptably long. A technique called **log compaction** is employed to deal with this problem. *Snapshots*[17] are the simplest approach to compaction. Each time the log reaches a fixed size, the entire current system state is written to a snapshot on stable storage and the log up to that point is discarded, as depicted in Figure 2.5. Moreover, there's an additional use for snapshots: they are

[16] This is equivalent to saying that if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index.

[17] A snapshot is (a copy of) the state of a system at a particular point in time.

sent to new cluster members or followers that are too far behind progress-wise so that they can catch up with the leader [35].

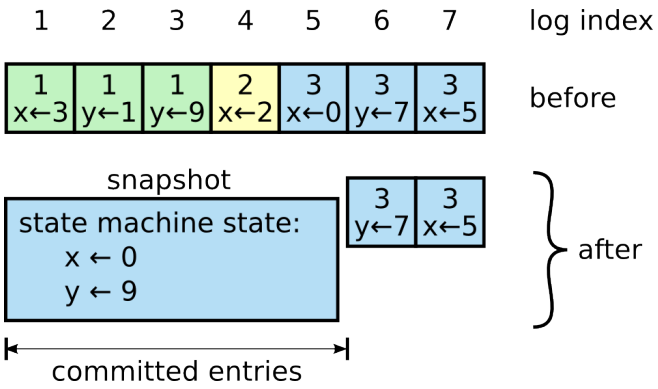


Figure 2.5: *Log compaction in Raft [35]*

Other consensus algorithms

Raft was actually developed in an attempt to provide a simpler, more *understandable* alternative to *Paxos*, the consensus algorithm devised by Leslie Lamport in 1989 [37]. Paxos is equivalent to Raft regarding their produced results and efficiency, but its complexity makes it exceptionally difficult to understand and does not provide a good foundation for the implementation of practical systems. The most notable difference between Raft and Paxos is Raft's strong-leadership; in Paxos leader election only serves as a performance optimization [35]. Other methods to achieve consensus across a set of nodes include the two-phase commit protocol (2PC), the three-phase commit protocol (3PC), Oki and Liskov's Viewstamped Replication and Chandra-Toueg consensus algorithm.

Implementations of Raft

At least 89 open-source implementations of Raft exist today [38] and are in various stages of development. It is worth mentioning LogCabin[18], D. Ongaro's implementation of Raft in C++. Among software systems that make use of Raft to achieve consensus etcd, RethinkDB[19], TiKV and Consul[20] stand out.

[18] `https://github.com/logcabin/logcabin`
[19] `https://github.com/rethinkdb/rethinkdb`
[20] `https://github.com/hashicorp/consul`

2.2 etcd Distributed Key-Value Store

2.2.1 Overview

etcd[21] is an open-source distributed key-value store that provides a reliable way to store data across a cluster of machines. It is a consistent, fault-tolerant and highly-available system. In CAP theorem parlance, etcd is a CP system. Etcd runs on each machine in a cluster and uses the *Raft* consensus algorithm to handle communication between machines and achieve consensus. The cluster acts as a replication group. Each modification is assigned a global unique ID, called a *revision*. An incoming modification request first has to pass through the Raft protocol and then it can be committed. Applications running on client machines can issue read and write requests to etcd.

Etcd is written in *Go* and its name derives from the Unix "`/etc`" directory plus "d" from distributed systems. The `/etc` directory is used to store system configuration so the combination of this notion with that of distributed systems accurately epitomizes the intended usage of etcd: to store configuration information for large-scale distributed systems [1], [10].

Etcd is developed by the *CoreOS* company, a team of developers who have also built an open-source operating system called Container Linux. This lightweight operating system is intended for large server deployments and is optimized for containers[22].

Guarantees

- **Atomicity:** all API requests are atomic; an operation either completes entirely or not at all.

- **Consistency:** as stated in etcd documentation [39], "All API calls ensure *sequential* consistency[23], the strongest consistency guarantee available from distributed systems. No matter which etcd server a client makes requests to, a client reads the same events in the same order". "As with all distributed systems, it is im-

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

[22] A container is one of multiple isolated user-space instances running on a machine and sharing the same kernel. From the point of view of a program running in it, it looks like a real computer.

[23] See subsection 2.1.5

possible for etcd to ensure strict consistency[24]. Etcd does not guarantee that it will return to a read the most recent value (as measured by a wall clock when a request is completed) available on any cluster member".

- **Partition tolerance:** if the cluster is made up of an odd number of servers it is certain that in occurrence of a network partition there will exist a majority partition able to resume operation normally. A leader election may be needed if the old leader is not in the majority partition.

- **High availability:** since the key-value data stored within etcd is automatically distributed and replicated with automated master election and consensus establishment using the Raft algorithm, all changes in stored data are reflected across the entire cluster, while the achieved redundancy prevents failures of single cluster members from causing data loss. More specifically, a cluster will remain available as long as any server majority is functional. Being both CP and highly-available does not contradict the CAP theorem. This is because in the context of the CAP theorem availability is regarded as a binary property, whereas here it is perceived as a spectrum (e.g., a system is available 99.99% of the time). In other words, etcd being both CP and highly-available means that whenever a majority of replicas can talk to each other, they should be able to make progress. Availability in the CAP sense of the term also differs from this definition because it requires that "every request received by a non-failing node in the system must result in a response (non-error or timeout) within a reasonable amount of time", as discussed in subsection 2.1.3.

- **Isolation:** etcd ensures *serializable* isolation[25], which is the highest isolation level available in distributed systems. Read operations will never observe any intermediate data.

- **Durability:** any completed operations are durable. etcd stores key-value pairs in a persistent storage engine. What is more, it persists the Raft log to disk and can replay it after power loss.

- **Simplicity:** being a key-value store etcd is inherently simple. In addition to that, it has a well-defined, well-documented API.

[24] See subsection 2.1.5
[25] See subsection 2.1.5

- **Security:** etcd provides automatic TLS[26] with optional client certificate authentication, as well as user and role-based access control.
- **Performance:** latest benchmarks show that etcd can support tens of thousands of writes per second.

Data model

By employing *MVCC*, etcd is capable of facilitating inexpensive snapshots and access to revision history. Each mutative operation creates a new revision on the key space.

The store's logical view is a flat binary key-space that is lexically sorted. Etcd stores the physical data as key-value pairs in a persistent B+ tree, implemented by its storage engine, *BoltDB*. The key-value pairs inserted in the B+ tree are slightly different from the ones inserted in the database by the client application. As described in the documentation of etcd [40], each key is a 3-tuple (major, sub, type). Major is the store revision holding the key. Sub differentiates among keys within the same revision (i.e., keys that have been updated in the same transaction). Type is set to `t` if the value contains a tombstone or is empty for a put operation. The value contains the actual key-value pair inserted in the database, which is the delta from the previous revision, along with other information. The B+ tree is ordered by key in lexical byte-order.

Etcd also keeps a secondary in-memory B-tree index to speed up range queries. The keys in this B-tree index are the keys that are exposed to the user. The values are pointers to the modifications of the persistent B+ tree [40].

Maintenance

- **History compaction:** to prevent the data store from growing excessively over time due to the accumulation of past revisions, the option of periodic or manual compaction is offered. Compaction discards the oldest versions of data when they are no longer needed.
- **Backend defragmentation:** after numerous write and delete operations (e.g., after a compaction), the backend database may exhibit internal fragmentation.

[26] Transport Layer Security

This means that even though disk space has been freed up by BoltDB it is unavailable to the host file system (it is still available to etcd though). The reason why this happens is explained in subsection 2.3.3. Etcd enables the user to issue a defragmentation in order to release storage space back to the file system.

- **Storage space quota:** According to the developers of etcd, the existence of a space quota is necessary to prevent performance deterioration caused from excessive keyspace growth, or space exhaustion. For this reason, etcd has a hard-coded quota with a default value of 2GB, configurable up to 8GB. When it is exceeded, a cluster-wide alarm is raised and the cluster no longer accepts write requests [41]. An additional reason behind the existence of the backend size restriction is to keep the *mean time to recovery* (MTTR) low. The impact of the MTTR becomes evident when a cluster member crashes and has to be replaced by a new one. Having to wait for a very large snapshot to be transferred to the new member would hurt the availability of etcd.

API

Interaction with etcd usually happens through *etcdctl*, a simple command line client. Here, we mention the functionality exposed by the gRPC[27] API of etcd [42]:

- **KV:** range, put and delete range requests. Transactions and history compaction. Range requests can also be used for point lookups if the range end is not specified. There are two types of range requests: linearizable and serializable. *Linearizable* reads have higher latency and lower throughput than serializable requests since they have to go through a quorum of cluster members, but reflect the current consensus of the cluster. *Serializable* reads do not require consensus and can be processed by any cluster member. They offer better performance as they can be served by any single etcd member, but may return stale values. The default option for range requests is linearizable. For a more thorough analysis of linearizability and serializability see subsection 2.1.5.

- **Watch:** a watch request watches for events happening or that have happened and are related to a key or set of keys.

[27] An open source remote procedure call (RPC) system initially developed at Google.

- **Lease:** granting, revocation and renewal of leases[28]
- **Maintenance:** alarm[29] activation and deactivation, member status queries, backend defragmentation, backend hash computation, snapshot.
- **Authentication:** configuration (enable/disable), creation, deletion and listing of users and roles, granting of permissions.
- **Cluster:** addition, removal and listing of cluster members.
- **Lock:** the lock service exposes client-side distributed shared locking facilities as a gRPC interface.
- **Election:** the election service exposes client-side election facilities as a gRPC interface.

Disaster recovery

An etcd cluster can tolerate up to up to $(N-1)/2$ server failures in a cluster of N members, leader failure and network partitions. However, when a majority of its servers fails, the cluster refuses to accept updates. To recover from this situation a *snapshot* file is required. Snapshots serve as backups of the etcd keyspace and the whole cluster can be restored from a single snapshot file [43]. This is the same snapshot file that is sent over the network to a new cluster member or to a slow follower to help it catch up. It is useful here to *disambiguate* between this kind of snapshot and the Raft snapshot which enables log compaction and is internal to etcd (i.e., not exposed by the API). The latter is further explained in subsection 2.2.2.

Recommended cluster sizes

As we have noted before, because etcd does not implement sharding, it does not scale in capacity as we add more nodes to the cluster. Its performance does not improve either, as every request still has to go through the leader and having to replicate data across more machines induces extra latency for writes. The reason for adding more

[28] Leases are a more efficient implementation of TTL (time to live). Instead of assigning a TTL to each key, keys with the same TTL are attached to a common lease. When a lease expires all keys attached to it are deleted.

[29] The etcd server raises an alarm whenever the cluster needs operator intervention to remain reliable.

nodes to an etcd cluster is to increase the replication level and therefore enforce its *fault tolerance*. A 5-member cluster offers a good trade-off between fault tolerance (can tolerate 2 member failures) and performance (the leader has to replicate to 2 members before responding to the client). 3-member clusters are also very common. The reason that *odd* cluster sizes are preferred is that for any such cluster adding one more node will always increase the number of nodes necessary for quorum[30]. According to the documentation of etcd [44], "although adding a node to an odd-sized cluster appears better since there are more machines, the fault tolerance is worse since exactly the same number of nodes may fail without losing quorum but there are more nodes that can fail". The following table gives a better idea of how an odd-sized cluster tolerates the same number of failures as an even-sized cluster but with fewer nodes.

Cluster Size	Majority	Fault Tolerance
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3
9	5	4

Table 2.1: *Quorum and fault tolerance in relation to etcd cluster size*

Another reason in favour of odd-sized clusters is that during a network partition they guarantee that there will be a majority partition that continues to operate. In the case of even-sized clusters this does not hold and a network partition that separates $N/2$ nodes from the rest $N/2$ can render the whole cluster unavailable [44].

2.2.2 Evolution of the Storage Backend

In its previous version (v2) etcd stored the key-value pairs in an *in-memory* engine. Among the changes made in v3 was the transition to a B+tree-based on-disk storage engine with full MVCC support, *BoltDB*. This enables etcd to handle larger datasets, while hot data is still retained in memory for fast access.

[30]The number of active members needed for consensus to modify the cluster state. Etcd requires a member majority to reach quorum.

In the latest releases of etcd the v2 backend is still accessible through the v2 API for backward compatibility reasons. The new on-disk store and the old in-memory one are separate and isolated. According to development plans, v2 backend will eventually be deprecated.

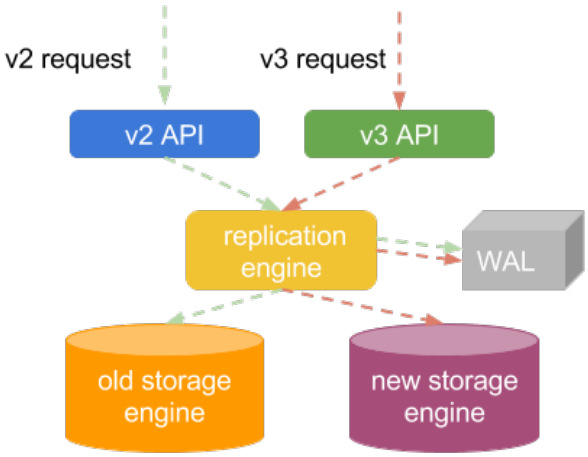


Figure 2.6: *v2 and v3 storage engines in etcd [45]*

Incremental snapshots

The new disk-backed storage engine allows etcd to perform incremental snapshots. While in v2 a snapshot was a separate file, in v3 the on-disk *database file* works as the snapshot. This snapshot is updated via commits to the backend, which only write updates (the delta) and without stopping the world, thus providing incremental snapshots. This leads to less I/O and CPU cost per snapshot.

2.2.3 Use Cases

"Etcd was designed to be the backbone of any distributed system", as its developers state. Its most common use cases up to date are:

- **Shared configuration:** by storing configuration in a globally accessible store, one can offer the same options to each instance of a cluster with no additional work. Some examples of shared configuration are: cluster membership details (e.g., IP addresses), environment variables, cluster-wide alerts, database connection details, cache settings, feature flags or other application-specific settings.

This way, a configuration change can be automatically reflected across a cluster of servers or containers. With etcd these values can be watched, allowing an application to reconfigure itself when they change.

- **Service discovery:** service discovery is the automatic detection of services offered on a computer network. Service discovery tools are generally implemented as a globally accessible registry that stores information about the instances or services that are currently operating. At its core, it is about knowing when any process in a cluster is listening on a TCP or UDP port, and being able to look up and connect to that port by name. As we read in an article on service discovery by J. Ellingwood [46], "each service, as it comes online, registers itself with the discovery tool. It records whatever information a related component might need in order to consume the service it provides. For instance, a MySQL database may register the IP address and port where the daemon is running, and optionally the username and credentials needed to sign in. When a consumer of that service comes online, it is able to query the service discovery registry for information at a predefined endpoint. It can then interact with the components it needs based on the information it finds. One good example of this is a load balancer. It can find every backend server that it can feed traffic to by querying the service discovery portal". Changes can be dynamically reflected in the registry, a very useful feature for modern, cloud-based microservices[31], applications that have dynamically assigned network locations. Service discovery also offers a way of *monitoring machine liveness*. If a component fails, the discovery service will be updated to reflect the fact that it is no longer available [46].

- **Distributed locking:** "the purpose of a distributed lock is to ensure that among several nodes that might try to do the same piece of work, only one actually does it (at least only one at a time)", in the words of M. Kleppmann. Examples of such work are: *synchronizing access to shared resources* (e.g., writing data to a shared storage system) or performing a critical update. In the latter case, if all the servers of a cluster performed the update and rebooted at the same time, the cluster would be unable to answer client requests during that interval. In-

[31] As Martin Fowler has put it "the microservices architectural style is an approach to developing a single application as a suite of small services, each running its own process and communicating with lightweight mechanisms".

stead, if only one server at a time is updating, the system retains its availability, performing what is called a *rolling update*.

- **Leader election:** large-scale systems that operate in a cluster and have a single leader, such as HDFS[32], typically use a separate replicated state machine to manage leader election and store configuration information that must survive leader crashes [35].

Specific projects that utilize etcd in a production environment include:

- **Container Linux by CoreOS:** applications running on Container Linux get automatic, zero-downtime Linux kernel updates. To coordinate updates Container Linux uses locksmith, which implements a distributed semaphore over etcd to ensure that only a subset of a cluster is rebooting at any given time [10]. This is the use case that inspired the development of etcd in the first place. Container Linux runs etcd as a daemon across all computers in a cluster. It also uses etcd to provide a dynamic configuration registry, allowing various configuration data to be easily and reliably shared between the cluster members.

- **Kubernetes[33]:** it stores configuration data into etcd for service discovery and cluster management; the consistency of etcd is of vital importance for the correct scheduling and operation of services. The Kubernetes API server persists cluster state into etcd and uses the watch API of etcd to monitor the cluster and apply critical configuration changes [10].

- **CloudFoundry[34]:** it uses etcd to store cluster state and configuration and provide a global lock service. etcd is also used, to a much lesser extent, as a discovery mechanism for some components.

- **TiDB:** as we have already mentioned in subsection 1.3.2, the Placement Driver (PD) is the central controller in the TiDB cluster and is implemented as an etcd

[32]Hadoop Distributed File System (HDFS) is designed to reliably store very large files across machines in a large cluster.

[33]Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers. `https://github.com/kubernetes/kubernetes`

[34]Cloud Foundry is an open source, multi cloud platform as a service (PaaS) on which developers can build, deploy, run and scale applications.

cluster. It saves the cluster meta information, allocates the global unique timestamp for the distributed transactions, etc. The PD embeds etcd to supply high availability and automatic failover [47].

2.2.4 Similar Systems

ZooKeeper

ZooKeeper is a distributed, open-source coordination service for distributed applications [48]. To achieve consensus it uses Zookeeper Atomic Broadcast (ZAB), a custom algorithm based on Paxos. ZooKeeper servers keep their entire state machine *in memory*, but every mutation is written to a durable WAL. ZooKeeper is a mature project used by many big companies like Yahoo and Rackspace. The developers of etcd admit that the lessons learned from ZooKeeper have influenced the design of etcd. It is very similar to etcd but lacks some features that etcd offers: stable reads/writes under high load, the MVCC data model, an API for safe distributed shared locks, etc. Also, ZooKeeper is a very *complex* system related to etcd [10].

Consul

Consul is a distributed key-value store for service discovery and health checking. Its similarities to etcd include also being a CP system, using *Raft* for consensus and being written in Go. Consul stores its key-value pairs in an *in-memory* database. Consul's main difference from etcd and ZooKeeper is that it is heavily focused on *service discovery*. While etcd and ZooKeeper only provide a plain key-value store and require application developers to build their own system on top of that to provide service discovery, consul also offers its own framework for service discovery, accessible via a DNS or HTTP interface. Consul also uses a feature-rich gossip system that links server nodes and clients. The gossip protocol implements efficient health checking and allows clients to check that a web server is returning 200 status codes, that memory utilization is not critical, that there is sufficient disk space, etc. [49]. However, as can be seen in Figure 2.7, Consul's performance as a key-value store cannot scale as well as that of etcd with the number of clients. What is more, its key-value API is not as rich as that of etcd, as it does not support MVCC, conditional transactions or

watches. To sum up, etcd and Consul solve different problems. The use of consul is advised when looking for end-to-end cluster service discovery, while etcd is a better choice when looking for a distributed consistent key-value store [10].

2.2.5 Performance

The performance of etcd is determined by multiple factors [50]:

- **Disk I/O latency:** maintaining the Raft WAL requires frequent `fsync` operations. The typical `fsync` latency for a spinning disk is about $10ms$. For SSDs[35], the latency is often lower than $1ms$. To attain high throughput despite heavy load, etcd batches multiple requests together and submits them to Raft. This way, the cost of `fsync` is split among multiple requests. However, write performance in etcd is *dominated by logging* consensus proposals.

- **Network latency:** the minimum time required to complete an etcd request equals the Round Trip Time (RTT) between members, plus the duration of the `fsync` operation that commits the data to permanent storage. Within a datacenter the RTT is in the order of *several hundred microseconds*. A typical RTT within the United States is around $50ms$, whereas between continents it is around $400ms$.

- **Storage engine latency:** each etcd request must pass through the BoltDB-backed MVCC storage engine of etcd, which usually takes *tens of microseconds*.

- **Snapshots:** periodically etcd incrementally snapshots its recently applied requests, merging them back with the previous on-disk snapshot, a process that may lead to a latency spike.

- **Compactions:** ongoing compactions impact the performance of etcd. Fortunately, their impact is often insignificant since they are staggered and do not compete for resources with regular requests.

- **gRPC API:** this introduces a small additional latency.

[35] Solid-State Drives

In the tables below we present some performance numbers, as measured by the developers of etcd using the built-in CLI[36] benchmark tool. Tests were run on a cluster of 3 VMs[37] running Ubuntu 17.04, on Google Cloud Compute Engine. Each VM has 8 vCPUs, 16GB memory, and a 50GB SSD. The client machine had 16 vCPUs, 30GB memory and a 50GB SSD. The version of etcd was 3.2.0, compiled with Go 1.8.3. The key size was 8 bytes and the value size was 256 bytes.

Number of keys	**Number of clients**	**Average write throughput**	**Average latency per request** (ms)
$10,000$	1	583	1.6
$100,000$	1000	$50,104$	20

Table 2.2: *Write performance of etcd*

Number of keys	**Number of clients**	**Consistency**	**Average read throughput**	**Average latency per request** (ms)
$10,000$	1	Linearizable	$1,353$	0.7
$10,000$	1	Serializable	$2,909$	0.3
$100,000$	1000	Linearizable	$141,578$	5.5
$100,000$	1000	Serializable	$185,758$	2.2

Table 2.3: *Read performance of etcd*

Performance comparison with similar systems

Here we present a write performance comparison between etcd, ZooKeeper and Consul, performed by the etcd developer team with dbtester[38]. Tests were run on Google Cloud Platform Compute Engine virtual machines with Ubuntu 16.10. Each cluster used three VMs, each of which had 16 dedicated vCPUs, 30GB memory, and a 300GB SSD with 150 MB/s sustained writes.

The chart below shows how scaling client concurrency impacts writes. As expected, when concurrency increases, write throughput, tends to increase in order to match request pressure. In the case of etcd it grows steadily. Zookeeper, on the other hand,

[36]Command Line Interface
[37]Virtual Machines
[38]`https://github.com/coreos/dbtester`

loses its write rate on account of writing out full state snapshots; a much more expensive procedure than the incremental snapshots of etcd, which write only updates and without stopping the world, as G. Lee explains [5]. The throughput of Consul also drops to low rates under concurrency pressure.

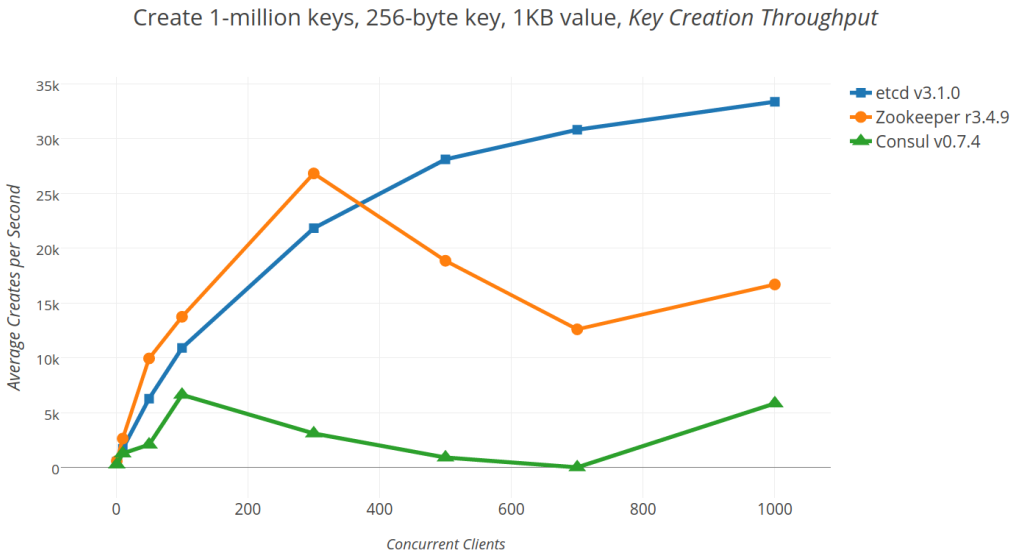


Figure 2.7: *Average throughput for creating one million keys as clients scale: etcd vs ZooKeeper vs Consul [5]*

2.3 BoltDB Storage Engine

A *storage engine* is a software component whose purpose is to manage data stored in memory or on disk. It is usually integrated into other application software that requires access to data. Each storage engine implements an *indexing* algorithm [15].

2.3.1 B+ Trees

The B+ tree, which is a variation of the B-tree, is the underlying indexing structure of BoltDB. Below, we list the definition and attributes of both of these data structures.

B-trees

B-trees have become a de facto standard for file systems (e.g., NTFS, BTRFS, Ext4) and databases. A *B-tree* is self-balancing and keeps data sorted. Operations (searches,

insertions, deletions) on a B-tree complete in *logarithmic* time. It is a generalization of the binary search tree in the sense that a node can have more than two children [51].

According to Knuth's definition [52], a B-tree of order m is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k-1$ keys.
5. All leaves appear in the same level.

The literature is not uniform in its terminology concerning B-trees. More specifically, Comer (1979) [51] as well as Bayer & McCreight (1972) define the order d of the B-tree as the minimum number of keys in a non-root node (so the maximum is $2d$), unlike Knuth (1998) who gives the definition we presented above. From this point onwards, we will follow Comer's definition as it is more convenient for the complexity analysis of operations.

The length h of the path from the root to any leaf is called the *height* of the tree. In the worst case it is equal to $\log_d n$, where n is the number of keys in the tree. In the B-tree, each internal node's keys act as separation values which divide its subtrees. Wikipedia provides an enlightening example: "if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2" [53].

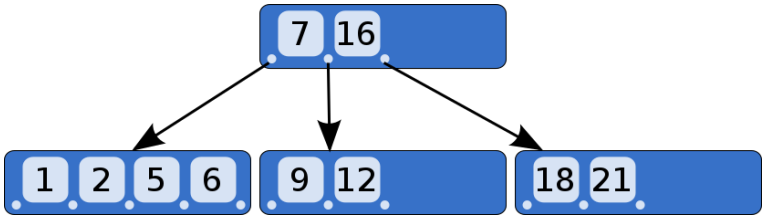


Figure 2.8: *A B-tree of order 5 [53]*

Search: searching a B-tree is similar to searching a binary search tree. Starting from the root and comparing the query key to each key stored in a node, the appropriate path is chosen to proceed from. The decision procedure is repeated at each node until

an exact match occurs or a leaf is encountered. A search operation in a B-tree of order d that stores n keys will not need to visit more than $1 + \log_d n$ nodes [51].

Insertion: this is a two-step process. First, a search must be performed to locate the proper leaf for insertion. Then, the insertion is performed and balance is restored if necessary. If the leaf is able to accommodate another key without violating the first constraint of Knuth's definition, nothing else has to be done. If however the leaf is already full, a *split* occurs: the first d of the $2d + 1$ keys are placed in one node, the last d in another, and the remaining key is promoted to the parent node. As D. Comer explains, "if the parent node happens to be full too, then the same splitting process is applied again. In the worst case, splitting propagates all the way to the root and the tree increases in height by one level" [51]. The procedure of insertion requires $\mathcal{O}(2 \log_d n)$ node accesses.

Deletion: this operation also requires a search first, to locate the key that will be deleted. In case the key resides in a non-leaf node, a new separator key must be found and swapped into the vacated position so that subsequent searches will work as expected. This key is found in the leftmost leaf of the right subtree of the now empty slot. Once this has been done (or in the case that the key to be deleted resides on a leaf node), we must check that at least d keys remain in the leaf. In case of an *underflow*, to restore balance a key can be transferred from a neighbouring leaf that has more than d keys. If no leaf can spare a key, then the deficient leaf must be *merged* with another one. This causes the parent to lose a separator key and rebalancing may have to be reapplied, continuing up to the root in the worst case. Just like insertion, the procedure of deletion requires $\mathcal{O}(2 \log_d n)$ node accesses [51].

Sequential accesses: so far we have only considered random searches. If a user requests all the records in key-sequence order using a *next* operator the B-tree will not perform so well. In fact, it may need up to $\log_d n$ accesses to process a "next" operation. Later we will see how this problem is solved by B+ trees [51].

An interesting property of B-trees is that their height and consequently the expensive node accesses can be reduced by maximizing the number of keys within each node.

B+ trees

The B+ tree is a data structure very similar to B-trees but with a couple of substantial differences [51]:

1. All keys reside in the leaves. The upper levels, which are organized as a B-tree, consist only of an index, a roadmap to enable rapid location of actual keys.
2. Leaf nodes are linked together from left to right, providing efficient sequential accesses.

As can be seen in Figure 2.9, the leaves may also contain pointers to the values that correspond to their keys.

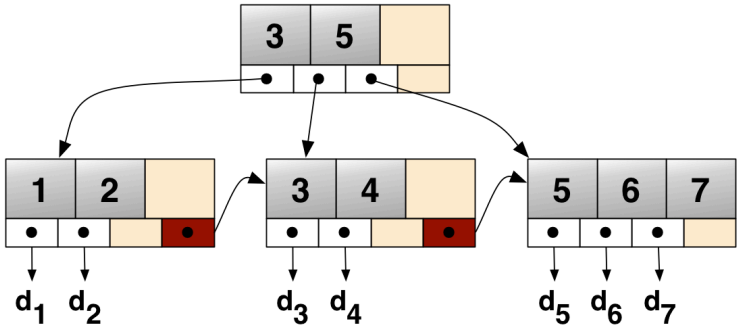


Figure 2.9: *A B+ tree of order 4 [54]*

Decoupling the keys from the indices in B+ trees simplifies the rebalancing process after deletions as non-key values can now be left in the index part. Insertions and searches remain the same as in the B-tree for the most part, with a few exceptions. For example, during an insertion that happens to overflow a leaf, when the leaf splits in two instead of promoting the middle key, a copy of it is promoted, retaining the original in the right leaf. What is more, search operations do not stop if a match is found in the index part of the tree. Instead, the nearest right pointer is followed and the search proceeds all the way to a leaf [51].

The number of accesses required in the worst case for searches, insertions and deletions is the same as in the case of the B-tree, which is $\mathcal{O}(\log_d n)$. However, when it comes to *sequential* searches the B+ tree demonstrates a clear advantage. Thanks to its linked list at the leaf level, it requires at most 1 access to satisfy a "next" operation. This means that accessing all the keys sequentially would require $\mathcal{O}(\log_d n + n)$ accesses, while in a

B-tree it would require $\mathcal{O}(n \log_d n)$. Therefore, B+ trees are a good fit for applications which entail both random and sequential processing [51].

Operation \ Structure	**B-tree**	**B+ tree**
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Insertion	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Deletion	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Next	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Table 2.4: *Complexity of basic operations in B-trees and B+ trees*

Both in B-trees and B+ trees, given that the block size of a storage system is B and the size of the keys to be stored is k, the most efficient tree is the one with an order (in the sense of Knuth's definition) of $m = \frac{B}{k}$. This takes into account the extra space needed by the pointers stored in the node. To adjust this to a RAM environment it would suffice to set B equal to the size of the processor's cache line [54].

Copy-on-write B+ trees

What BoltDB uses as its underlying indexing structure is actually a copy-on-write B+ tree, also known as an append-only B+ tree. This way it avoids random in-place updates replacing them with *sequential writes* at the end of a file. To understand how this structure works we will consider the 3-level B+ tree of Figure 2.10. Copy-on-write B+trees do not generally support links between the leaves, as that would require the whole tree to be rewritten on each update.

In this example, borrowed from M. Hedenfalk [55], we assume that each node corresponds to a page. In the database file the pages are stored sequentially as can be seen in Figure 2.11. The meta page contains a pointer to the root page, a hash and statistic counters. When the file is opened it is scanned backwards page by page until a valid meta page is found that leads to the root.

When updating a value in leaf page 8, instead of changing the page in-place, a whole new page is appended to the file (here as page 12). Because the location of the page

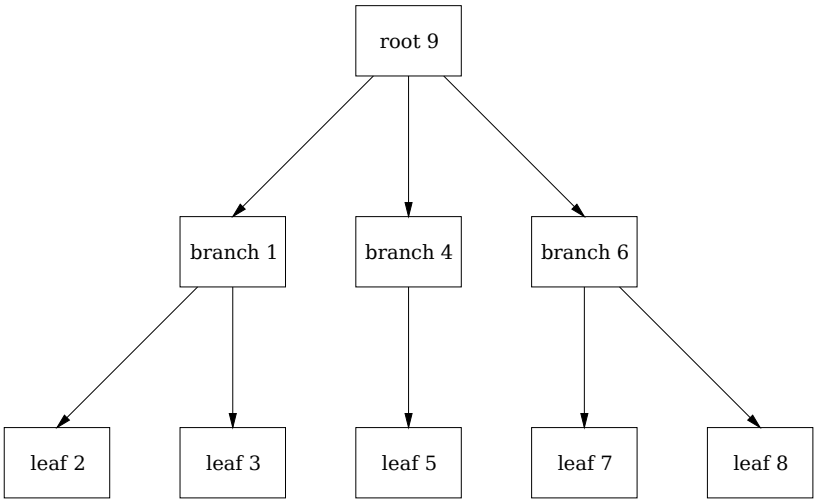


Figure 2.10: *A copy-on-write B+ tree with 3 levels [55]*

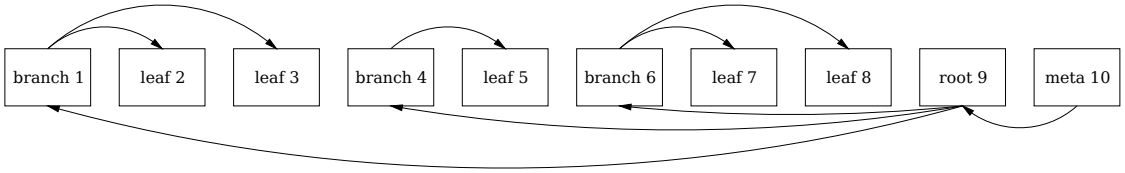


Figure 2.11: *On-disk layout of the pages of a B+ tree [55]*

is changed, the parent page needs to be updated to point to the new location and this process is repeated until the root is reached. Any readers still holding a pointer to the old root page can traverse the tree unaffected by the change. They will see a consistent snapshot of the database. Dashed pages and pointers in Figure 2.12 still exist in the file, they just do not reflect the latest version.

Changes are written sequentially by appending new pages to the file. Written pages are immutable. After a new revision of the tree is created, a meta page pointing to the new root is written. In our example changing one page resulted in 4 new pages being appended to the file. This introduces significant write and space amplification, but writing consecutive pages to disk is more efficient than writing at random locations.

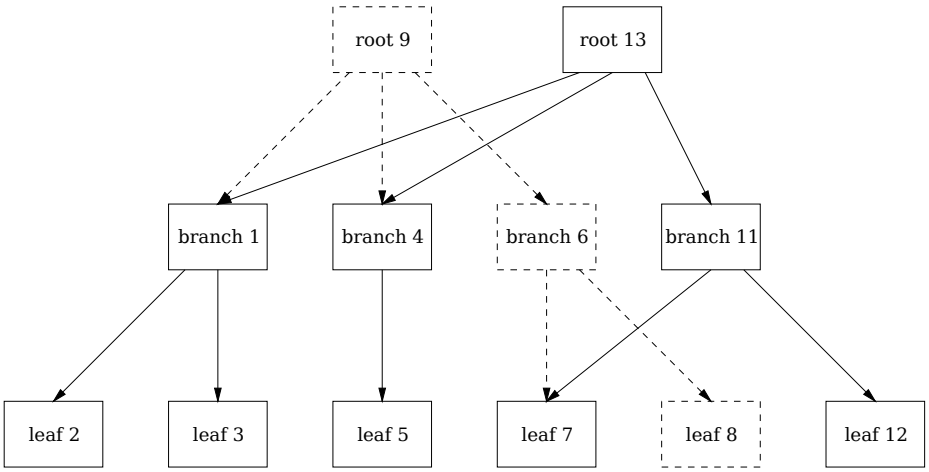


Figure 2.12: *A copy-on-write B+ tree after updating a value [55]*

2.3.2 Basic Concepts & API

BoltDB[39], is an embedded key-value store developed by Ben Johnson. It initially started as a port of LMDB[40] to Go, but the two projects diverged. They share the same architecture but BoltDB is focused on *simplicity* and ease of use while LMDB is focused on performance. LMDB is a high-performance embedded transactional key-value store written in C.

BoltDB stores everything in a *memory-mapped*[41] file, implementing a copy-on-write B+ tree that supports MVCC. This makes reads very fast, as they can be executed concurrently with writes without the need for a lock. Only one writer at a time is allowed, but as many readers as necessary. BoltDB does not have a write ahead log. It provides *ACID* transactions with serializable isolation. *All* commands must go through a transaction. It is a project of amazing simplicity (its code base is smaller than 3KLOC[42]), requires no prior configuration or tuning and has a small, well-documented API [57].

The basic concepts of BoltDB as well as the functionalities offered by its API are summarized in the following list [58]:

[39] `https://github.com/boltdb/bolt`

[40] Lightning Memory-Mapped Database
`https://symas.com/lightning-memory-mapped-database/`

[41] According to Wikipedia [56], "a memory-mapped file is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource". "Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory", thus increasing I/O performance.

[42] 3 Thousand Lines of Code

- **DB:** in BoltDB the top level object is a `DB`, which represents a single memory-mapped file on disk. Functions applicable on the `DB` include `bolt.Open()` and `DB.Close()`. The `DB` can only be opened by one process at a time. However, it does have the ability to be opened by multiple processes in `ReadOnly` mode.

- **Transactions:** only one *read-write* transaction is allowed at a time but as many *read-only* transactions as the user wants. Each transaction has a *consistent view* of the database as it was when the transaction started. Transactions may be created, committed or rolled back manually (with functions `DB.Begin()`, `Tx.Commit()`, `Tx.Rollback()`), or by using BoltDB's wrapper functions `DB.Update()` and `DB.View()` that take care of all transaction management details.

- **Buckets:** the storage space in BoltDB is divided into buckets. Buckets are collections of key-value pairs within which all keys must be unique, so in a sense, buckets represent separate namespaces. Buckets may be created (`Tx.CreateBucket()`), deleted (`Tx.DeleteBucket()`) or retrieved (`Tx.Bucket()`) by the user in the context of a transaction with the purpose of subsequently putting, getting or deleting key-value pairs from them. Nested buckets are also supported.

- **Key-value pairs:** to save a key-value pair to a bucket the `Bucket.Put()` function is called. Likewise, to retrieve a key-value pair the `Bucket.Get()` function is used and deletion is performed by calling `Bucket.Delete()`.

- **Cursors:** keys are stored in byte-sorted order within a bucket. This makes sequential iteration very fast. To iterate over keys a `Cursor` is used. Operations applicable on a `Cursor` include `Bucket.Cursor()` for its creation, `Cursor.First()`, `Cursor.Last()`, `Cursor.Seek()`,`Cursor.Next()` and `Cursor.Prev()`. There is also a function called `Bucket.ForEach()` that can be used to apply a user-defined function on every key-value pair in a bucket.

- **Backups:** being a single file, BoltDB is easy to backup. The `Tx.WriteTo()` function can be called from within a read-only transaction to write a consistent view of the database to a writer. The writer can be anything that implements Go's `io.Writer` interface, e.g., a Go pipe, a file etc. [58].

The following is part of a description of the query path when using BoltDB as an embedded key-value store, as given by its creator, B. Johnson [59]:

1. "Start a transaction. This involves acquiring a single `sync.Mutex` lock which takes around $50ns$. After the transaction starts, the mutex is released and no additional locks are required during execution."
2. "Traverse through a B+ tree to find your key-value pair. Many times your branch data is cached in-memory so only the leaf value needs to be fetched from disk. This operation can take $1\mu s$ if all pages are cached or a couple hundred μs if pages need to be fetched from an SSD."

Use cases

Admittedly, BoltDB is a better fit for read-heavy projects. It is currently used in high-load production environments serving databases as large as 1TB. Some of the open-source projects that embed BoltDB are etcd, Consul, Heroku[43] and InfluxDB[44].

Embedded versus standalone databases

An embedded database is a library included in and compiled with application code. Using a standalone database running in a remote server incurs a network transport overhead not present in embedded databases. On the other hand, standalone databases give systems flexibility to connect multiple application servers to a single database server [58]. Also, it is easier for them to scale out. Finally, embedded databases tend to be simpler to configure and use.

2.3.3 Caveats & Limitations

- BoltDB is a good fit for read-intensive workloads. Sequential write performance is also fast but *random writes* can be slow, especially as the database size grows [58].

[43] `https://www.heroku.com/`
[44] `https://github.com/influxdata/influxdb`

- BoltDB uses a memory-mapped file so the operating system handles the caching of the data. Typically, the OS will cache as much as possible of the file and release memory to other processes when needed. Consequently, when working with large databases BoltDB may exhibit *high memory usage*. Nevertheless, it can handle databases much larger than the available physical RAM, provided that the memory-mapped file fits in the virtual address space of the process. This means that on 32-bit systems there is a 2GB restriction on the database size [58].

- BoltDB exhibits both external *fragmentation* and a kind of internal fragmentation. As far as the latter is concerned, BoltDB works by allocating 4KB pages and organizing them into a B+ tree. New pages are written to the end of the file as needed. When first created, a BoltDB database file has a size of 1MB. This provides space for some metadata pages (`M`), a page for storing free pages (`F`), some pages with actual data (`D`), and some unallocated page slots ( ). As more data is written, the original 1MB will eventually be exhausted. BoltDB will then remap the database to give the user 2MB. In other words, it keeps doubling the file size until it reaches 1GB and from then on it applies 1GB increments. As a result, the database may reserve disk space that it is not actually using [60].

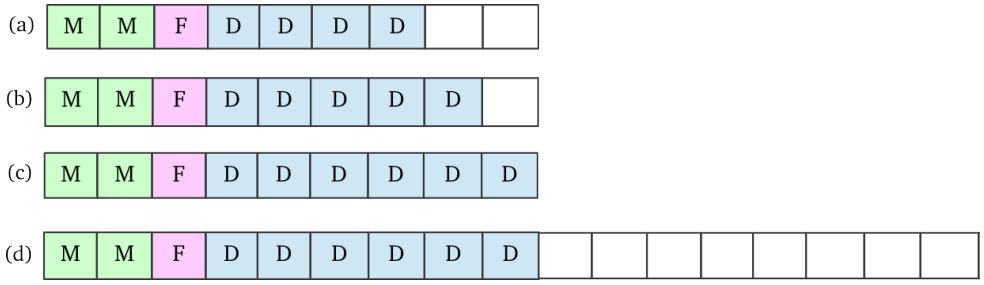


Figure 2.13: *Internal fragmentation in BoltDB*

External fragmentation happens due to the arrangement of pages on disk. According to its documentation [58], BoltDB cannot *truncate* data files and return free pages back to the disk. Instead, it maintains a free list of unused pages within its data file, which can be reused by future transactions. This is an adequate approach for many use cases as databases generally tend to grow. However, it is important to note that deleting large chunks of data will not allow the user to reclaim their space on disk. *Defragmentation* can only be achieved by copying the whole database to a new file. This operation is not yet supported by BoltDB

but it is relatively easy to implement on top of it.

More precisely, when deleting data in BoltDB that causes a page to be removed, that page could be located anywhere in the file. This is clearly illustrated in Figure 2.14. BoltDB is unable to truncate the file because the free page is in the middle of it. Compacting pages from the end of the file to the beginning would be complicated as it would require updating all references to each moved page in its parent pages [60].

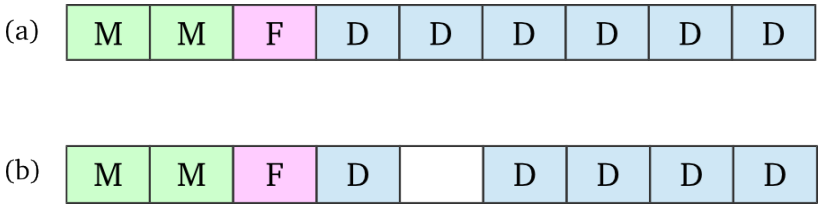


Figure 2.14: *External fragmentation in BoltDB*

2.4 RocksDB Storage Engine

2.4.1 Log-Structured Merge-Trees

RocksDB uses an LSM-tree as its indexing structure. The LSM-tree (Log-Structured Merge-Tree) is a data structure proposed by O'Neil et al. in 1996. Its basic components are the memtable and a set of Sorted String Table files.

A *Sorted String Table* (SSTable or just SST file) is a file that contains a set of arbitrary, sorted key-value pairs. SST files are stored *on disk* and they may contain duplicate keys. One of the most important properties of SSTs is that they are *immutable*. They may be implemented as B-trees.

The *memtable* is a tree-like structure that resides *in memory*. The simplest version of the LSM-tree is a two-level LSM-tree, like the one in Figure 2.15. C_0 corresponds to the memtable and C_1 to the on-disk part of the tree. Although the C_1 component is disk resident, frequently referenced page nodes in it will be cached in memory [61].

Most LSM-trees used in practice employ multiple levels of increasing size. Having more than two on-disk levels reduces the number of files per level, hence leading to more performant reads, as well as efficient merges from one level to the next.

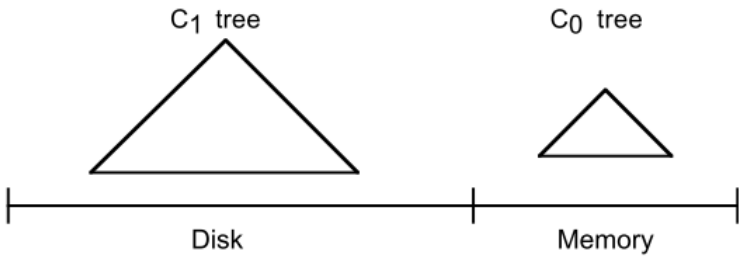


Figure 2.15: *Schematic diagram of an LSM-tree of two components [61]*

Insertion: when using an LSM-tree, all writes go to the memtable. They are always performed in memory and hence are very fast ($\mathcal{O}(1)$). The LSM-tree takes its name because of the memtable's log-like behaviour.

Search: when a read operation is requested, the system first checks the memtable. If the key is not found there, the on-disk files will be inspected one by one, in reverse chronological order, until the key is found [7]. In the worst case all levels may need to be checked before either finding the key or deciding that it does not exist. Range queries on LSM-trees tend to be slow as the keys of a range may be scattered across multiple levels of the tree, thus requiring a lot of file accesses.

Updates and deletions: updates are not performed in place, since as we have already mentioned SST files are immutable. Instead, they are stored in the memtable. Also, when deletions happen, a "tombstone" record is stored in the memtable. Upon a read request, the most recent versions (or tombstones) are returned, since the memtable is accessed first [62].

Obviously, the memtable's size is limited by a system's memory capacity. Once it reaches a certain size threshold it is *flushed* to disk as a new immutable SST file, and a new memtable takes its place. Therefore, in LSM-trees the cost of writes is amortized by batching them. Although in an LSM-tree a write may be issued multiple times as it moves to lower levels of the tree, because the I/O cost is divided among a large batch, the cost per insert ends up being smaller than one I/O operation.

When the number of files in a level exceeds a certain threshold, a *compaction* is performed. During a compaction on-disk SST files are *merged* together into bigger files and moved to the next level to keep the number of files low (a large number of files degrades read performance). During this process deduplication is also performed,

which means that recent updates and delete records overwrite and remove the older data. Because the SST files are sorted the process of merging the files is quite efficient [7]. However, it induces a periodic I/O penalty.

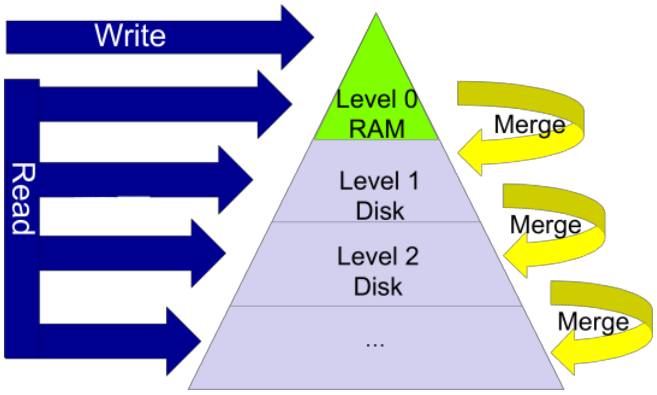


Figure 2.16: *LSM-tree storage [63]*

An entry inserted in an LSM-tree will start in the memtable and eventually migrate to the bottommost on-disk level through a series of asynchronous merge steps [61]. It is worth noting that the system is only performing sequential I/O as files are not updated in place. In that sense, we can say that an important property of the LSM-tree is that it *converts* random writes to sequential writes.

Often an LSM-tree is complemented by a *WAL* to ensure durability. By writing key-value pairs to an append-only log upon each insert and before storing them in the memtable, we ensure that in case of power-loss the data of the memtable that has not yet been flushed to disk will not be lost. On restart, the memtable can be reconstructed by simply replaying the WAL.

All in all, the LSM-tree is most useful in applications where inserts are more common than retrievals. BigTable[45], HBase[46], Cassandra, RocksDB, MongoDB, WiredTiger and InfluxDB all use LSM-trees or variants.

Read, write and space amplification

Read, write and space amplification are notions of utmost importance for the design decisions that must be made when implementing an LSM-tree. *Write amplification* is

[45] https://cloud.google.com/bigtable/
[46] https://hbase.apache.org/

the ratio of bytes written to storage versus bytes written to the database. For example, if data is written to a database at a rate of 10MB/s but a 30MB/s disk write rate is observed, the write amplification is 3. High write amplification is undesirable not only because it hampers write performance, but also because it is detrimental to SSDs. *Read amplification* refers to the number of disk reads per read query. For example, if 5 pages need to be read in order to answer a query the read amplification is 5. *Space amplification* describes how much extra space a database will use on disk compared to the size of the data stored in the database. If 10MB is stored in the database but it uses 100MB on disk, then the space amplification is 10. Compression is a means to reduce space amplification [64].

Bloom filters

As we have seen, the LSM-tree has a relatively high read amplification. This may not be acceptable for applications where read performance is critical. Fortunately, there are a number of optimizations that can be applied in this direction, including Bloom filters and maintaining in-memory page indices for each file. Bloom filters offer a way to know if a file does not contain a certain key, without having to search through the file. By keeping Bloom filters in memory when using an LSM-tree, disk accesses to SST files are reduced substantially as reads from files that are known not to contain a given key are prevented.

Bloom filters are memory efficient probabilistic data structures implemented as *bit arrays* of m bits. At first all bits are set to 0. When an element is added to the file it is fed to k different *hash functions*, each of which maps it to one of the m array positions. The bits of those positions are then set to 1. To test whether an element is in the file, it is fed to the k hash functions and k array positions are returned. If any of the bits at these positions is 0, the element is *definitely* not in the set. If all the bits at these positions are 1, then either the element is in the file, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a *false positive*. False positives are possible but false negatives are not [65].

Elements cannot be removed from this simple version of the Bloom filter, as this would introduce the possibility of false negatives. As it is explained in Wikipedia [65], "an element maps to k bits, and although setting any one of those k bits to zero suffices to

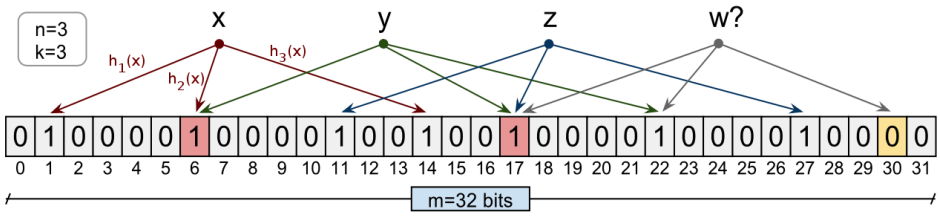


Figure 2.17: *A Bloom filter with a 32-bit array and 3 hash functions [66]*

remove the element, it also results in removing any other elements that happen to map onto that bit".

Like all probabilistic data structures, Bloom filters trade a small margin of error for space-efficiency. The probability of false positives is given by the formula $p = (1 - e^{-\frac{kn}{m}})^k$, where n is the number of elements inserted in the file. Using this formula one can compute the optimal number of hash functions k given n and m, or the length of the bit-array m that achieves the desired false positive possibility given n and k [66], [65].

Levelled & universal compaction

So far in the description of the LSM-tree it has been assumed that universal compaction is the compaction type employed. According to the documentation of RocksDB [67], when using this compaction style the SST files are arranged in *time order*. Each of them is organized as a sorted run covering the whole key space and data generated during a specific time range. Different sorted runs never overlap on their time ranges but they can contain overlapping keys. Compaction can only happen among two or more sorted runs that are chronologically adjacent. The output of this merge is a single sorted run whose time range is the combination of the input sorted runs and is stored in the next level of the tree. After any compaction, the condition that sorted runs do not have overlapping time ranges still holds.

Obviously, SST files stored in the lower levels of the tree are bigger, since they have been created by merging the files of the higher levels and removing duplicates by keeping only the most recent update on a given key-value pair. Often, when universal compaction is used there is only one on-disk level. This compaction style targets use cases that require lower write amplification, trading off read and space amplification. For

a read request to be answered SST files are consulted in reverse chronological order, but since their key ranges are overlapping, all of them may have to be consulted in the worst case. On the other hand, every update will be written at most n times, where n is the number of the on-disk levels of the LSM-tree.

Levelled compaction appears in newer implementations like RocksDB and Cassandra and manages to reduce the number of files that must be consulted for the worst case read. It also reduces the relative impact of compaction by spreading it over time, as well as space usage. However, the write amplification is higher in this case. The main differences of levelled compaction from universal compaction are that:

1. The more recent data resides in the first level and the oldest data is in the bottom-most level. Each level has a size threshold and is guaranteed, as a whole, to not have overlapping key ranges within it (except for the first on-disk level, which is structured in the same way as in universal compaction). That is to say the keys are partitioned across the available files. Thus, the whole level can be seen as *one sorted run* and to find a key in it only one file needs to be consulted. To identify a position for a key, we first perform a binary search over the key range of all files to identify which file possibly contains the key, and then perform a binary search inside the file to locate the exact position [68]. The number of SST files that will have to be consulted in the worst case when performing a read is equal to $number\_of\_L0\_files + (n-1)$, where n is the number of the on-disk levels.

2. SST files are of fixed size (e.g., 2MB). A compaction is triggered when the files in Level 0 reach a predefined number. Then, all files in Level 0 are merged in Level 1. Files are also merged into lower levels *one at a time*, whenever the size threshold of a level is exceeded, in order to create space for more data to be added [7]. In the general case, a compaction process picks one file in level L_n and all its overlapping files in L_{n+1} and *replaces* them with new files in L_{n+1} [69]. Typically, each level is 10 times bigger than the previous one.

The write amplification is greater in this case, because upon compaction, instead of only writing the file that is merged from L_n to L_{n+1}, the overlapping files of L_{n+1} are also rewritten. Table 2.5 presents a comparison of read and write amplification between universal and levelled style compaction.

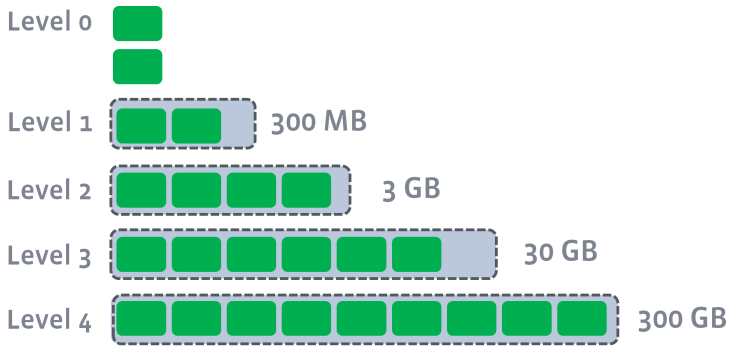


Figure 2.18: *Levelled compaction layout in LSM-trees [68]*

	Universal Compaction	**Levelled Compaction**
Read amplification (worst case)	Greater, as every SST file in every level will be consulted	Equal to $number\_of\_L0\_files+(n-1)$, where n is the number of on-disk levels
Write amplification	Every update will be written at most n times	Greater, because upon compaction, instead of only writing the file that is merged from L_n to L_{n+1}, the overlapping files of L_{n+1} are also rewritten

Table 2.5: *Read & write amplification in universal and levelled style compaction*

Number of on-disk levels

It is rather common for someone recently acquainted with the structure of LSM-trees to wonder why one on-disk level is not enough. In the case of levelled compaction, having more than one on-disk level reduces the number of files that must be consulted for the worst case read. If there was no Level 1, the number of files in Level 0 would gradually grow very large. Of course, adding more levels sacrifices write performance. In the case of universal compaction, having more than one on-disk level reduces space amplification, as duplicates and deleted key-value pairs are removed during the compaction process. Obviously, in this case as well, this increases write amplification.

Likewise, one could ask themselves what the point of having more than two on-disk levels is. The answer is that it makes compactions more efficient. If there is no Level

2, and keeping in mind that the size of the SST files is fixed, the ranges of the files in Level 1 will gradually become very fine grained. Then, the keys of a file on Level 0 will overlap with many files in Level 1. As a result, a compaction will require opening, reading and rewriting a lot of files in order to merge them with the Level 0 file.

There exist LSM-tree variations with SST files of variable size. In that case, not having a third on-disk level would make the files of Level 1 gradually grow very large, which would again be detrimental for read performance. Yet another explanation, is that an LSM-tree with multiple on-disk levels that is employing levelled compaction acts as a multi-level cache structure in which the most recently updated data is located in higher levels.

Read I/O cost analysis

In an LSM-tree, read queries on SST files are performed using binary search. Searching a file requires $\mathcal{O}(\log \frac{N}{B})$ I/Os, where N is the database size and B is the read block size. If there are multiple SST files in a level, then a query may require an I/O operation for each of them. In the case of LSM-trees following the levelled compaction style, the largest level's size is $\mathcal{O}(N)$, thus requires $\mathcal{O}(\log \frac{N}{B})$ I/Os. The level before that has a size of $\mathcal{O}(\frac{N}{k})$, where k is the level-to-level size multiplication factor. It follows that searching this level requires $\mathcal{O}(\log \frac{N}{kB}) = \mathcal{O}(\log \frac{N}{B}) - \mathcal{O}(\log k)$ I/Os. The next level requires $\mathcal{O}(\log \frac{N}{k^2 B}) = \mathcal{O}(\log \frac{N}{B}) - \mathcal{O}(2 \log k)$ I/Os and so on. So the total number of I/O operations is asymptotically $R = (\log \frac{N}{B}) + (\log \frac{N}{B} - \log k) + (\log \frac{N}{B} - 2 \log k) + \ldots + 1$, which has a solution of $R = \mathcal{O}((\log \frac{N}{B})(\log_k \frac{N}{B})) = \mathcal{O}(\frac{(\log^2 \frac{N}{B})}{\log k})$ [70].

Comparison with B+ trees

Here we sum up the different traits of LSM-trees and B+trees:

1. LSM-trees are optimized for writes.
2. B+ trees are optimized for reads.
3. The SST files an LSM-tree creates are immutable. This makes the locking semantics over them much simpler [7].
4. LSM-trees may put more pressure on memory.

5. LSM-trees utilize disk space more efficiently as they do not exhibit fragmentation like B+ trees do.

6. The performance of B+ trees tends to degrade as the dataset becomes larger than the available memory.

7. Copy-on-write B+ trees achieve better write throughput than regular B+ trees by avoiding random writes. Still, their write amplification is significantly larger than that of LSM-trees, as part of the structure has to be rewritten on each update. Both in copy-on-write and regular B+ trees, when updating an element at least one whole page has to be written to disk.

8. B+ trees can handle large values more efficiently. In LSM-trees insertion of large values might trigger repeated compactions which will incur extra delays. In B+ trees write amplification is inversely proportional to the value size. In LSM-trees on the other hand, write amplification does not depend on the value size. Consequently, as one moves from smaller to larger values, there is a point where B+ trees will become better [71].

Interesting variants of LSM-trees and B-trees

- **bLSM-trees:** they are a variant of LSM-trees that attempts to combine the read and scan performance of B-trees with the fast writes of LSM-trees. bLSM-trees ensure reads can stop after finding one version of a record, thus avoiding multiple SST file lookups. Also, they ensure that SST file merges make steady progress and do not block writes for a long time [6].

- **B^{ε}-trees / Fractal trees:** they are a write optimized B-tree variation whose innovation is that as the working set grows larger than main memory, write performance stays consistent. Internal nodes of the tree have buffers for each child that batch write operations. When a data record is inserted into the tree, instead of traversing the entire tree the way a B-tree would, it is simply added to the buffer at the root of the tree. Buffers are flushed to lower levels of the tree when they become full, until eventually data is flushed all the way down to leaves [72].

2.4.2 Basic Concepts & API

RocksDB[47] is a library that provides an embedded, persistent *key-value store*, written in C++ and developed by Facebook. It builds upon earlier work on LevelDB, developed by Google. RocksDB has an *LSM-tree* design with flexible configuration settings that may be *tuned* to run on a variety of production environments, and to achieve the desired trade-off between write, read and space amplification. RocksDB is optimized for flash storage, delivering extremely low latencies. It also has the ability to scale linearly with the number of CPUs.

RocksDB differs from existing key-value stores due to its focuses. It is a low-level data engine whose primary design point is performance for fast storage and server workloads. It is not a distributed database, it does not offer failure-tolerance or high-availability but these features can be built on top of it if necessary.

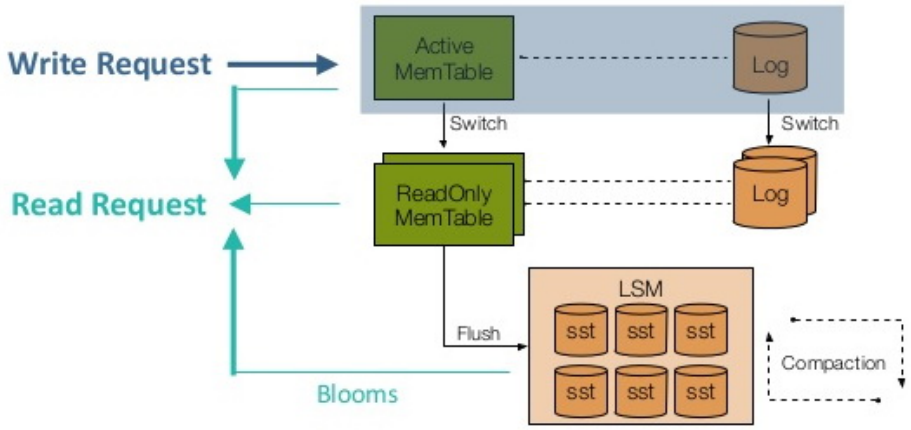


Figure 2.19: *The architecture of RocksDB [73]*

Following the LSM-tree design, the three basic constructs of RocksDB are the *memtable*, the *SST file* and the *logfile* (WAL). When the memtable fills up and is flushed to an SST file on storage, its corresponding log file can be safely deleted. RocksDB also supports *memtable pipelining* to increase write throughput. It may be configured with an arbitrary number of memtables. When one of them becomes full, it becomes immutable and a background thread starts flushing its contents to storage. Meanwhile, new writes are directed to a newly allocated memtable instead of being stalled [69].

[47] `https://github.com/facebook/rocksdb/`

The basic concepts of RocksDB as well as the functionalities offered by its API are summarized in the following list [69]:

- **DB:** the name given to the database corresponds to a file system directory. Functions that can be used upon the `DB` include `Open()` and `DestroyDB()`. A database may be opened in read-only mode, ensuring higher read performance as locks are avoided. A RocksDB database can be opened by multiple processes in read-only mode but not in normal mode.

- **Key-value pairs:** keys and values in RocksDB have the form of arbitrary byte arrays. Key ordering can be specified by the user via a comparator function. The `Get()` function allows an application to fetch a single key-value pair from the database. The `MultiGet()` function atomically retrieves a bunch of key-value pairs. Bloom filters may be enabled to facilitate reads.

  The `Put()` function inserts a single key-value pair to the database. Likewise, the `Delete()` function removes a key-value pair from the database. The `Write()` function allows multiple key-value pairs to be *atomically* inserted, updated or deleted. The set of these key-value pairs is called a `WriteBatch`. In other words, a `WriteBatch` holds a sequence of updates to be made to the database and applies them in order when the `Write()` function is called. Using a `WriteBatch` to update multiple keys usually performs better than using `Put()` for each of them, as the cost of *synchronously*[48] writing to the WAL is *amortized* across all of the updates in the batch.

  `WriteBatchWithIndex` is a variation of `WriteBatch` that serves the *read-your-own-writes* scenario, where a reader needs to have access to a transaction's uncommitted writes. It achieves that by maintaining an internal buffer for all the written keys, in the form of a searchable index that supports iteration. When `Get()` is called on a `WriteBatchWithIndex`, before checking data in RocksDB, it is first checked whether the value exists in the `WriteBatchWithIndex`, and if it is, it is returned from there. In case of a range query, a super iterator is internally created, which *merges* the results of RocksDB with those of the `WriteBatchWithIndex`'s buffer [74].

[48] Asynchronous writes return as soon as they have been delegated to the OS. A synchronous write blocks until it has reached persistent storage. Adding updates to the `WriteBatch` is an asynchronous operation, while calling `Write()` is synchronous.

- **Iterator:** the `Iterator` API is used to perform range scans on the database. The `Iterator` can seek to a specified key and then start scanning one key at a time from that point. All keys returned via the `Iterator` are from the same consistent view of the database.

- **Options:** RocksDB uses options structs for a variety of purposes. There are `DB Options` used when opening the database, `ReadOptions` and `WriteOptions` used for each read or write, `TransactionOptions` and so on.

- **Snapshot:** a `Snapshot` allows an application to create a *point-in-time view* of the database. Then, the `Get` function and the `Iterator` can be configured to read data from a specified `Snapshot` by passing it in their options struct. Snapshots are a logical concept and are *not persisted* across database restarts.

- **Backup:** creates a point-in-time snapshot that the database state can later be rolled back to. The memtable is flushed before taking the snapshot so that its changes are included. Backups work by *copying* the database and are normally incremental. The database files can even be copied to a remote file system (e.g., HDFS).

- **Checkpoint:** a checkpoint creates a physical mirror of a running database in an other directory of the same file system. SST files are *hard-linked*[49], rather than copied, if the file system supports it, so it is a very lightweight operation [76]. Checkpoints are point-in-time snapshots and can be used as read-only copies of the database or can be opened as standalone databases.

- **Transactions:** transactions have a simple `Begin()`, `Commit()`, `Rollback()` API and provide *a way to guarantee that a batch of writes will only be written if there are no conflicts.* RocksDB supports both optimistic and pessimistic transactions. According to the documentation of RocksDB [77], when using *pessimistic transactions* conflict detection is performed by internally locking all keys that are written. In case a key fails to be locked the operation returns an error. *Optimistic transactions* perform conflict detection at commit time to validate that no other writers have modified the keys being written by them. If a conflict

[49] According to the Linux Information Project's definition [75], "a hard link is merely an additional name for an existing file on Linux or other Unix-like operating systems". "The original file name and any hard links all point to the same inode."

is found the commit operation returns an error and no keys are written. For a more thorough explanation of optimistic and pessimistic concurrency control, see subsection 2.1.7. Transactions also support easy reading or iteration over the keys currently batched in a transaction but not yet committed [77].

- **Compactions:** in RocksDB compactions can be processed concurrently by multiple threads, thus affecting positively the overall write throughput. Two styles of compaction are supported: universal style and levelled style. Both of these have been outlined in subsection 2.4.1. Also, a number of threads can be reserved for the purpose of flushing the memtable(s) to disk, so that incoming writes are not stalled in case the memtable(s) are full and all threads are performing compactions.

- **Compression:** RocksDB supports multiple compression algorithms, including Snappy, Bzip2, LZ4 and Zlib. A different compression algorithm may be chosen for each level. A typical installation might configure no compression for levels L_0-L_2, Snappy compression for the mid levels and Zlib compression for L_{max}. LZ4 and Snappy keep the performance good, while Zlib significantly reduces the data size.

- **Column families:** they provide a way to logically partition the database into separate key spaces. Column families share the WAL but don't share memtables and SST files. By sharing the WAL atomic writes across column families are possible. Each column family is implemented as a separate LSM-tree.

- **Block cache:** RocksDB uses an LRU[50] cache for hot blocks of the SST files to serve reads more efficiently.

After a compaction, compaction output files are added to the list of live SST files, while compaction input files are removed from it. However, input files may not be instantly deleted, as they may be in use by a `Get()` operation or an `Iterator`. The list of SST files in the LSM-tree is kept in a data structure called `version`. Whenever a compaction or a memtable flush happens, a new version is created. At each point in time only one of the preserved versions is the current one. Out of date versions are dropped

[50]Least Recently Used

when the `Get()` operations using them are finished or when `Iterators` are freed. This logic is implemented using reference counts [78].

Improvements over LevelDB

LevelDB is an open-source, fast key-value storage library written by Google. The improvements of RocksDB over LevelDB include support for [79]:

- Multi-threaded compactions. The single-threaded compaction process of LevelDB was insufficient for server workloads and caused latency spikes.
- Multi-threaded memtable inserts
- Bloom filters
- Column families
- Universal style compaction
- Transactions and batching of writes
- Backup and checkpoints
- Many tunable configuration options, etc.

In general, RocksDB achieves better write and read throughput.

Use cases

RocksDB can be used as a storage engine or as an embedded database by applications that need low-latency accesses. Facebook uses RocksDB as the storage engine for its distributed database, ZippyDB. Other users of RocksDB are MySQL, Ceph, LinkedIn, Yahoo, CockroachDB, Airbnb, Pinterest, Netflix, TiKV, etc. [80].

2.4.3 Comparison with BoltDB

Many of the differences between B+ trees and LSM-trees mentioned in subsection 2.4.1 apply here, as the underlying data structure is a determining factor for the behaviour

of a database. The most important observation is that BoltDB is better at handling read-heavy workloads because of the underlying B+ tree and the memory-mapping, while RocksDB is better at handling write-heavy workloads.

According to the developer of BoltDB, B. Johnson, "*if you require a high random write throughput (>10,000 w/sec) or you need to use spinning disks then LevelDB could be a good choice. If your application is read-heavy or does a lot of range scans then Bolt could be a good choice*" [58]. Furthermore, RocksDB makes more efficient use of storage: it applies compression and enables the user to tune it in order to achieve minimal write and space amplification. With compression less storage space will be used, and with less write amplification flash devices will last longer.

In addition, RocksDB supports I/O-bound[51] workloads optimally, while BoltDB struggles with datasets that are larger than the available memory. While the entirety of the dataset is cached, write and read operations are very fast, but when the size of the B+ tree surpasses the available memory, performance drops significantly as it becomes more probable for new requests to require pages that are not present in the cache, resulting in disk reads.

On the other hand, RocksDB is a much more complex system, with a multitude of tuning knobs that require a thorough study of its design in order to be successfully configured. BoltDB's inherent simplicity and its ability to be deployed right out-of-the-box might sometimes be a good enough reason to prefer it. Also, BoltDB's load time is better, especially during recovery from crash, since it does not need to read the log (it does not have one) to reconstruct a memtable and find the last succeeded transaction: it just reads the IDs of two B+ tree roots, and uses the one with the greatest ID.

[51] A workload is I/O-bound when the database size is much larger than memory and there are frequent reads from storage.

2.5 The Go Programming Language

2.5.1 Overview

Go is an open source programming language created by Google and intended to combine efficient compilation, efficient execution and ease of programming. It is statically typed, with a user-friendly, C-like syntax and garbage collection. It easily uses multiple cores, implements concurrency and works in distributed environments [81].

It is believed that the open-source community can benefit greatly from a language such as Go, as it was designed to facilitate the writing, debugging and maintenance of large software systems. Code simplicity and understandability is key for projects that are based on contributions and are therefore developed by multiple programmers [82]. On the other hand, Go has been extensively accused of being overly simplistic and restricting [83], [84]. Its lack of generics, leading to violation of the DRY[52] principle, is one of its most often criticised traits. A common response to these disputes is that Go, like every other programming language is just a tool, and one has to pick the right tool for the particular job they want to do. Infrastructure as a Service, web servers, backend services, cloud orchestration, DevOps and generally concurrent applications are some examples of suitable usecases for Go.

Go is becoming increasingly popular, as can be seen in Figure 2.20. This graph depicts the interest over time[53] in Go and some of the programming languages that it is often compared with. Twitter, BBC, Github, Bitbucket, Canonical, DropBox, DigitalOcean, SoundCloud and CoreOS, are only a few of its production users [89].

Some interesting Go constructs and tools that we utilized in our implementation include:

- **Goroutines:** functions that run in a goroutine are capable of running *concurrently* with other functions. Creating a goroutine is as simple as using the keyword `go` followed by a function invocation [86]. A goroutine is similar to a

[52] "Don't Repeat Yourself (DRY) is a principle of software development aimed at reducing repetition of software patterns, replacing them with abstractions", according to Wikipedia [85]. For example, this can be achieved by allowing the type of parameters of a function to be resolved at compile time, instead of rewriting a function for each type of parameters one needs to use it with.

[53] The value of 100 represents the highest popularity for a search term in Google.

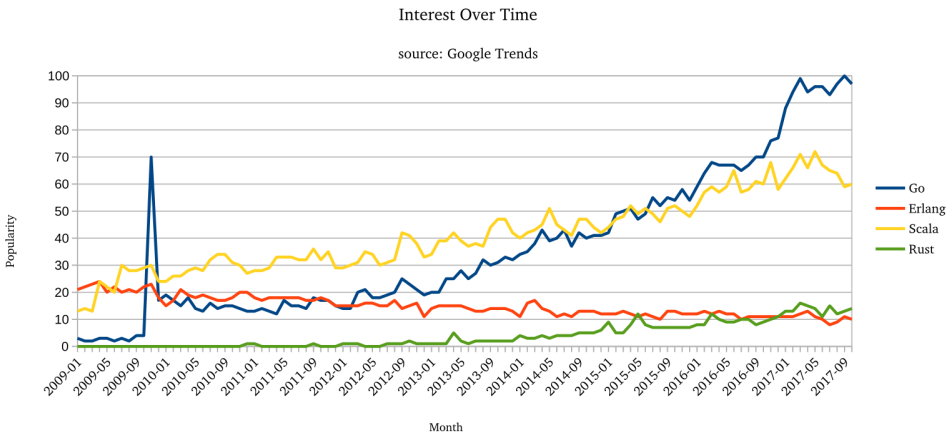


Figure 2.20: *Popularity growth of golang*

thread, with the difference that it is *scheduled by Go* and not the OS. Usually, multiple goroutines run on the same OS thread. Goroutines are lightweight and modern hardware can support millions of them [87].

- **Channels:** they provide a means of communication and synchronization for two or more goroutines. When a writer goroutine sends a message on a channel, it will wait until a reader goroutine is ready to receive the message (i.e., blocking/synchronous communication). However, channels may also be buffered, meaning that they behave in an asynchronous manner; sending or receiving messages will not wait unless the channel is already full [86]. Furthermore, Go has a very useful special statement called `select`. It resembles a `switch` statement but it is used for managing multiple channels. As C. Doxsey explains in his "Introduction to Programming in Go" [86], "`select` picks the first channel that is ready and receives from it (or sends to it)". "If more than one of the channels are ready then it randomly picks which one to receive from. If none of the channels are ready, the statement blocks until one becomes available." This provides a convenient way to implement a timeout.

- **Slices:** Go slices are analogous to the concept of arrays but are more commonly used in practice, as they possess a few extra properties that make them more flexible. Unlike arrays their length is not fixed and can be adjusted dynamically up to a predefined capacity. A slice is implemented in Go as a reference to an underlying array segment. To grow a slice beyond its initial capacity its contents must be copied into a larger slice [90].

- **Defer and Panic:** according to the Go Blog [91], "a `defer` statement pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. `Defer` is commonly used to simplify functions that perform various clean-up actions".

  Also, "`Panic` is a built-in function that stops the ordinary flow of control and begins panicking. When the function F calls `panic`, execution of F stops, any deferred functions in F are executed normally, and then F returns to its caller. To the caller, F then behaves like a call to `panic`. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes".

- **Growable stacks:** Go manages its goroutine stacks in a way that enables each goroutine to take up the least amount of memory. Instead of giving each goroutine a fixed amount of stack memory, like C does with threads, the Go runtime attempts to give goroutines the stack space they need on demand. When a goroutine is created, a section of memory of a few kilobytes[54] is allocated in order to be used as its stack. Each Go function has a prologue at function entry, that checks if the allocated space has been used up and calls the `morestack` function if it has. The `morestack` function allocates a new section of memory with double the size of the previous one and copies the old segment into it. At the bottom of the stack, a stack entry for a function called `lessstack` is inserted, so that we return into it when the function that caused running out of stack space finishes its work [88].

- **Vendoring:** the practice of vendoring allows storing the external dependencies (i.e, third-party packages) in a folder called `vendor`, which is located within a project. This way, problems caused by unstable third-party libraries can be avoided by keeping a local copy of a working version. The Go tools, such as `go build` or `go run` first check if the dependencies are located in the `vendor` folder. Only if they are not found there, they are imported from `$GOPATH/src/`. There exist various dependency management tools, such as `godep` and `govendor`, that facilitate this process.

- **The Go toolchain:** it includes tools such as `pprof`, that provides a heap or CPU

[54] In C the stack is in the order of megabytes.

profile, and race detector. `go test` is another useful tool that implements automated testing using test functions contained in `_test.go` suffixed files within a package, and output PASS/FAIL messages.

2.5.2 Cgo: A Necessary Evil

In plenty of cases it may be needed to use a library written in C from within Go code. There are two ways to do this. The first, is the cgo tool, which is part of the Go distribution. The second is the SWIG program, which is a general tool for interfacing between languages[55] [92]. In this project we have opted to use cgo, as SWIG seemed considerably more complex to configure.

Cgo enables the creation of Go packages that call C code by importing a pseudo-package called `"C"`. The Go code can then refer to types such as `C.size_t` and `C.int`, variables such as `C.stdout`, or functions such as `C.putchar`. When the `go build` tool detects Go files that use the special `import "C"`, it searches for C/C++ files in the directory and invokes the C/C++ compiler to compile them as part of the Go package. As we learn from the documentation of Go [93], "if the import of `"C"` is immediately preceded by a comment, that comment, called the preamble, is used as a header when compiling the C parts of the package". "The preamble may contain any C code, including function and variable declarations and definitions. These may then be referred to from Go code as though they were defined in the package `"C"`." What is more, `CFLAGS`, `CPPFLAGS`, `LDFLAGS`, etc. may be defined with `#cgo` pseudo-directives within the preamble to configure the behaviour of the C/C++ compiler [93].

The above can be clarified by the following code example [94], a Go package comprised of two functions - `Random()` and `Seed()` - that wrap C's `random` and `srandom` functions. The return value of `random()` is of type `C.long` and has to be converted to a Go type in order to be used by Go code outside this package. The `Seed()` function receives a Go `int` as its argument and converts it to the C `unsigned int` type before passing it to `srandom()` [94].

[55] `http://swig.org/`

```
1  package rand
2
3  /*
4  #include <stdlib.h>
5  */
6  import "C"
7
8  func Random() int {
9      return int(C.random())
10 }
11
12 func Seed(i int) {
13     C.srandom(C.uint(i))
14 }
```

Listing 2.1: *Cgo example*

There exist a few special functions that convert between Go and C types by making *copies* of the data. `C.CBytes` is one of those functions and it is used to convert a Go byte slice to a C array. The opposite conversion is performed by the `C.GoBytes` function. Conversion between Go and C strings is done with the `C.CString` and `C.GoString` functions. When a C string created with `C.CString` or a C array created with `C.CBytes` is no longer needed it must be freed by calling `C.free` [94].

It is worth noting that there are restrictions on passing pointers between Go and C. Go's garbage collector needs to know the location of every pointer to Go memory. As a result, Go code may pass a Go pointer to C, provided the Go memory to which it points does not contain any Go pointers[56]. Moreover, C code may not keep a copy of a Go pointer after the call returns [93].

Although cgo is a tremendously useful feature, it does not come without considerable drawbacks. We briefly describe a few of them below [95], [96]:

- **Performance degradation:** C is not familiar with Go's calling convention or

[56] A Go pointer is a pointer to memory allocated by Go (i.e., by using the `&` operator or calling the predefined `new` function).

growable *stacks*[57], so a call to C code has to record the goroutine stack and then *switch*[58] to a C stack.

However, the stack switch, being a relatively cheap operation, is not the main source of delay. The major part of cgo's overhead is owed to the fact that every cgo call is considered blocking and is treated by the Go runtime as a system call. When a goroutine enters a cgo call, it is locked on the thread it was running on and this thread blocks waiting for the call to complete. The thread also frees the `GOMAXPROCS`[59] slot it was occupying[60], so as not to block other goroutines or the garbage collector. After the cgo call returns, the goroutine blocks until there is an available `GOMAXPROCS` slot for it to run on. This requires *coordination with the Go runtime scheduler*, that involves acquiring and releasing a lock, which are atomic memory operations in the order of tens of nanoseconds. In addition, while the thread is blocking, the scheduler may need to create a new OS thread in order to run other ready goroutines [98], [99], [100].

Yet another source of overhead is that, as mentioned before, *copies* are often not avoidable when passing data from Go to C and vice versa.

Thus, the transition is non-trivial and imposes an overhead which, depending on where it exists in the code, may be insignificant or considerable. The *overhead* of a cgo call is reported to be between ten and a hundred times larger than that of a call within Go. For this reason, it is generally advisable to minimize the number of cgo calls. For example, rather than calling a cgo function repeatedly within a loop it is better to move the loop down to C code.

- **Slower build times:** the C compiler has to be invoked for every C file in the package.

- **Complicated builds:** a project using cgo can no longer be built by using the go tool only. A C compiler has to be installed, environment variables may need to

[57] The C code will not know how to grow the stack if it needs more than the few kilobytes that the Go stack offers.

[58] A stack switch involves saving all registers when the C function is called and restoring them when it returns.

[59] `GOMAXPROCS` is an environment variable that determines how many operating system threads can execute user-level Go code simultaneously. Its default value equals the number of available CPU cores. Threads that are blocked in system calls or cgo calls do not count against the `GOMAXPROCS` limit [97].

[60] In the latest versions of Go this only happens if the goroutine spends more than 20 microseconds running C code.

be set, shared objects need to be kept track of and C library dependencies must be satisfied. In addition, producing a single, *static*[61] binary is no longer a simple procedure and requires a lot of tweaking.

- **Cross compilation is no longer supported:** Go's support for cross compilation[62] is thought to be one of the language's strong points. When using cgo one has to give up this feature.
- **No access to Go's toolchain:** the race detector, the `pprof` profiler and other useful Go tools cease working when cgo is added to the mix. As a result, *debugging* becomes harder.
- **Manual memory management:** Go is garbage-collected, but C is not. As a result, special care must be taken when passing data from C to Go and vice versa.
- **Concurrency issues:** while goroutines are lightweight because of the size of their stack, one must keep in mind that when they involve blocking cgo calls each of them occupies a system thread. A thousand goroutines can easily be handled by the go runtime, but when they translate to a thousand threads this might cause significant performance issues. Thus, when a program contains calls to cgo, concurrency has to be appropriately bounded.

All in all, cgo should be used with care and only in cases where it is absolutely necessary. For example, cgo can be very helpful when one needs to avoid reimplementing in Go a large and complex library that already exists in C. What is more, it is generally advisable to use cgo only when the C function called does a substantial amount of work, such that its execution duration causes the overhead of the cgo call to be unnoticeable. Finally, the use of cgo is also justified when the performance of the function implemented in C is notably better than it would be in Go. Common use cases of cgo include calls to physics engines and graphics libraries such as OpenGL.

[61] A static binary has all of the required library code built in. In contrast, a dynamically linked binary uses shared libraries to satisfy its dependencies.

[62] Cross-compilation refers to the process of producing executable code for a platform other than the one on which the compiler is running [101].

3 Design

In this chapter we thoroughly analyse the fundamental design decisions we made prior to the implementation of our solution. At first, we mention the expectations we want our final system to meet and present the architecture of etcd after the integration of its new storage engine. Then, we discuss the reasons that led us to the selection of RocksDB among other storage engines. In addition, we describe the mapping of BoltDB concepts and constructs to their RocksDB counterparts with the preservation of initial semantics in mind.

As stated previously, etcd is designed to reliably store infrequently updated data and be used as a metadata store. Our primary aim is to improve its write throughput, as we believe that it has the potential to be used as a general-purpose key-value store. We plan to do this by replacing its current, read-optimized storage engine, BoltDB, with write-optimized RocksDB. Our expectations from the final system are that:

- A better write performance will be achieved
- Etcd's guarantees of reliability, consistency and high-availability will be honoured
- Read performance will deteriorate slightly, within acceptable levels, as has been made obvious by our analysis of the two storage engines and their underlying data structures in chapter 2

3.1 Proposed Architecture & Design Choices

3.1.1 Integration of RocksDB into etcd

The final step in the path of an incoming request to etcd involves being serviced by the storage engine. More specifically, a new put request first passes through the Raft protocol and then is processed by the `backend` package of etcd, which directs it to the storage engine in order to be persisted.

According to our initial design, we implement a wrapper whose purpose is to *map* the calls that the backend of etcd makes to the BoltDB library to the corresponding calls to the RocksDB library. This approach enables us to minimize interventions in core etcd code. In fact, we have only made changes to etcd code where the abstraction between the backend package and the storage engine key-value API was not good enough and the implementation was strongly coupled to BoltDB.

Our wrapper code is written in Go and is placed inside the `bolt` package that was already vendored in etcd. This way, the change of the storage engine is not *visible* to other etcd packages. We have preserved all BoltDB API functions used by etcd, but changed their code to call the corresponding functions of RocksDB. Finally, wherever the functionality of the two storage engines diverged we added the necessary helper functions to preserve the semantics.

Later, it will be trivial to add support for both storage engines, adopting a pluggable design like the one of MongoDB, which we outlined in subsection 1.3.3. This can be done by simply adding an appropriate compile-time or startup-time configuration option to etcd.

Figure 3.1 is a schematic representation of the architecture of etcd's backend after the integration of RocksDB. The components of the backend are described in the following subsections.

Understanding the steps that an etcd server takes in order to reply to a client request helps get a better idea of the role of the backend and its position in the workflow of etcd:

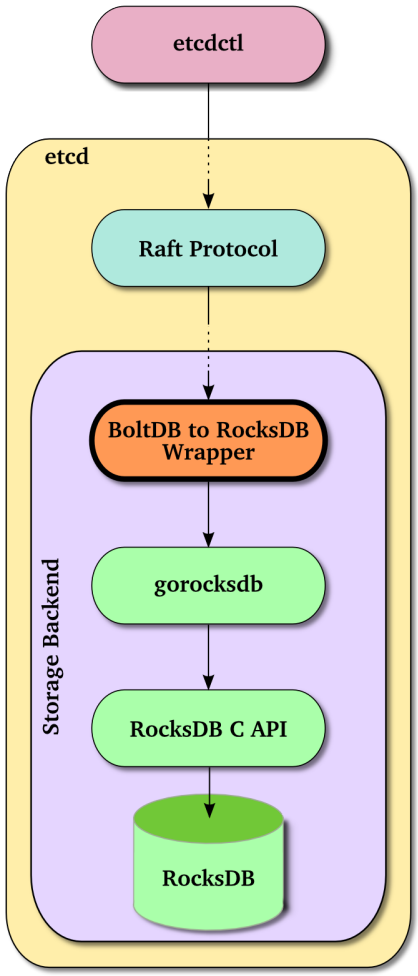


Figure 3.1: *Architecture of the etcd backend with RocksDB as its storage engine*

1. The etcd server receives a client request
2. It registers a new Go channel with a unique id in a map of channels
3. It forwards the request to the Raft protocol
4. It listens on the registered channel for a response, which it will forward to the client
5. The request is written to the WAL and passes through consensus
6. Then, it is applied to the backend
7. Finally, the result of its application is sent to the aforementioned channel

This means that, for example, when writing a key-value pair in etcd, the client will only receive a response when it has been successfully persisted to the backend. Besides, that is when etcd is ready to actually return the written value upon a get request.

3.1.2 Why RocksDB?

The principal reason behind the selection of RocksDB was the write-optimized underlying indexing structure it implements, the LSM-tree. According to Ben Johnson, developer of BoltDB, RocksDB is a better choice in cases where a high random write throughput (>10,000 writes/sec) is required [58]. What is more, RocksDB is a high-performance key-value database, backed by an experienced developer community. It is trusted to be stable and robust as it is already being used in production by many impressive software systems.

Some of the most popular databases that implement an LSM-tree are BigTable, HBase, Cassandra, SQLite[1], MongoDB, LevelDB, and InfluxDB. Out of them, the majority are distributed databases (BigTable, Hbase, Cassandra, MongoDB) with some of them also being proprietary. Others, like SQLite and InfluxDB are relational or have an SQL layer that renders them unsuitable for our scenario. Apparently, among the above only LevelDB and RocksDB fulfill our requirements of an LSM-tree based, open-source, local key-value store in the form of an embedded library. Having mentioned the differences between these two in subsection 2.4.2 there is obviously no reason to prefer LevelDB over RocksDB.

Furthermore, the integration of RocksDB is an idea that the developers of etcd have discussed themselves. In fact, in an etcd community Google Hangout in December 2015, they mentioned the possibility of trying RocksDB as a storage engine for etcd. They concluded that BoltDB is good enough for the current use case of etcd and left this open as an option for future investigation [102].

3.1.3 The gorocksdb Wrapper

As etcd is written in Go and RocksDB is written in C++ an intermediate software component is required for their integration. RocksDB already has a C API (a C wrapper

[1] `https://www.sqlite.org/`

of its C++ API) so what we need is a Go wrapper for this API. Gorocksdb[2] is exactly that: an open-source Go wrapper for RocksDB. There exist a few projects similar to gorocksdb but none is as actively maintained and feature complete.

Each function in the gorocksdb wrapper contains a call to the corresponding function of the RocksDB C API. Cgo is used to enable calling C functions from Go code. A typical gorocksdb function follows this workflow:

1. **Argument conversion:** Go byte slices are converted to `*C.char` pointers by using a function from Go's `unsafe` package to cast them to `Pointer`[3] type first, and then by being cast to `*C.char`, Go `bool` types are converted to `C.uchar`, etc.

2. **Function call:** the C function of the RocksDB C API is called via Cgo.

3. **Return values & Error handling:** the return values of C functions are converted to Go types in a manner analogous to that of argument conversion before being returned to the Go program. Also, error arguments are converted from `*C.char` pointer to Go `error` type.

Moreover, each wrapper struct in gorocksdb contains a field with a C pointer to the corresponding C struct. The above is clarified by the two following examples from gorocksdb code [103]:

```
1 // Put writes data associated with a key to the database.
2 func (db *DB) Put(opts *WriteOptions, key, value []byte) error {
3     var (
4         cErr  *C.char
5         cKey  = byteToChar(key)
6         cValue = byteToChar(value)
7     )
8     C.rocksdb_put(db.c, opts.c, cKey, C.size_t(len(key)), cValue,
          C.size_t(len(value)), &cErr)
```

[2] `https://github.com/tecbot/gorocksdb`

[3] Go's `Pointer` type belongs to the `unsafe` package and represents a pointer to an arbitrary type. A pointer value of any type can be converted to a `Pointer`, and vice versa. Therefore, `Pointer` allows code to override the type system and read and write arbitrary memory [97].

```
    if cErr != nil {
        defer C.free(unsafe.Pointer(cErr))
        return errors.New(C.GoString(cErr))
    }
    return nil
}
```

Listing 3.1: *gorocksdb* `Put()` *function*

```
// WriteOptions represent all of the available options when writing
// to a database.
type WriteOptions struct {
    c *C.rocksdb_writeoptions_t
}
```

Listing 3.2: *gorocksdb* `WriteOptions` *struct*

3.1.4 The RocksDB C API

The C API of RocksDB[4] is implemented as a single C++ file and C header file pair. All wrapper structs and functions are enclosed in an `extern "C"`[5] block.

Wrapper structs only have one field: a pointer to their corresponding C++ struct. Wrapper functions contain a call to their corresponding C++ function. The `SaveError()` helper function is used to convert the `Status` struct returned by C++ functions to a `char` pointer.

```
void rocksdb_put(
    rocksdb_t* db,
    const rocksdb_writeoptions_t* options,
    const char* key, size_t keylen,
    const char* val, size_t vallen,
    char** errptr) {
```

[4] `https://github.com/facebook/rocksdb/blob/master/db/c.cc`

[5] The `extern "C"` attribute on a C++ function turns off name mangling so that the public or external name becomes compatible with the C language. Name mangling is a technique to distinguish functions with the same identifiers by including extra information (e.g., number and type of arguments).

```
7   SaveError(errptr,
8           db->rep->Put(options->rep, Slice(key, keylen), Slice(val,
                vallen)));
9   }
```

Listing 3.3: *RocksDB C API* `rocksdb_put()` *function*

```
1   struct rocksdb_writeoptions_t { WriteOptions rep; };
```

Listing 3.4: *RocksDB C API* `rocksdb_writeoptions` *struct*

3.1.5 Removing the Storage Quota

As mentioned in subsection 2.2.1, etcd imposes a restriction upon the size of its storage backend, implemented as a configurable quota ranging from 2GB to 8GB. Although such a capacity may be reasonable for some of the use cases of etcd as a metadata store, it is certainly not enough for the general case.

Before removing the limitation we decided to carefully investigate the reasons behind its existence. The documentation of etcd mentions that it is imposed in order to prevent *disk space exhaustion* and *performance degradation* [41]. Running out of disk space is not a concern, as most modern computer systems are equipped with storage devices of a capacity much larger than 8GB. Performance degradation most probably refers to the B+ tree's and consequently BoltDB's tendency to not handle well datasets whose size exceeds the available memory. Since in our implementation we will be replacing BoltDB with an LSM-tree based storage engine, there is no reason to worry about this either.

However, there is an environment where the 2GB constraint would make sense: a system with a *32-bit* architecture. While in theory the address space of a 32-bit system is 4GB, the largest available contiguous block of memory is 2GB. As BoltDB uses a file *memory mapped* to a contiguous block of memory it cannot support datasets larger than 2GB on 32-bit systems [105]. By integrating RocksDB, which does not use a memory-mapped file, we can overcome this problem as well.

After discussing the matter with Xiang Li, one of the core developers of etcd, on the project's development mailing list, we gained some new insights. According to Xi-

ang Li, the main reason why the constraint exists is to keep the *mean time to recovery* (MTTR) within acceptable bounds. As we have mentioned again in subsection 2.2.1 the impact of the MTTR becomes evident when a cluster member crashes and has to be replaced by a new one. Having to wait for a very large snapshot to be transferred to the new member would hurt etcd's availability. More specifically, Xiang Li states that in case of a member crash, "simply adding a new member should bring the cluster back to full health within tens of seconds", and today's average hardware is not capable of transferring a very large snapshot in such a short time [104]. This is a problem we cannot easily overcome, but if for a specific usecase being able to accommodate a large dataset is a more important concern than the MTTR we can choose to ignore it.

Thus, we can safely lift the backend size restriction. The quota size is stored in a constant in the `backend` package of etcd, so it is trivial to change it to a larger value or to `math.MaxInt64`.

3.2 Mapping of Concepts & Constructs

In this section we examine the choices we made when mapping BoltDB concepts and constructs to their analogous concepts and constructs of RocksDB, and explain why each option was preferred. All concepts have been described in detail in subsection 2.3.2 and subsection 2.4.2.

3.2.1 DB

Obviously, BoltDB's DB object can be conveniently mapped to a RocksDB DB object. Even though the two are fundamentally different (i.e., the first corresponds to a memory-mapped file and the second one to a directory) they share the same basic functionality (e.g., `Open()`, `Close()` API calls, starting a new transaction on the DB etc.).

3.2.2 Buckets to Prefixes

In BoltDB buckets partition the dataset into separate namespaces. There are two ways to achieve this in RocksDB. The first one is to use Column Families. The second one is

to simply append the name of the bucket to the requested key before servicing queries to the database. Of course, as the strings representing the buckets in etcd vary in length and some of them even start with the same character sequence, we need to use a delimiter, in order to be able to determine where the prefix ends and the key name starts. We chose the "/" character for this. For example, if the key `foo` needs to be stored in bucket `keys`, we store `keys/foo` in RocksDB. Likewise, if the key `bar` has to be retrieved from bucket `meta`, we search for `meta/bar`.

According to the documentation of RocksDB, Column Families are most commonly used when:

1. Different compaction settings, comparators, compression types, or compaction filters are used in different parts of data.
2. Whole groups of data need to be deleted at once. This is achieved by dropping a Column Family.
3. Metadata needs to be stored separately from data. In this case, each is stored in their own Column Family.

Our implementation falls under the third case. However, in the documentation it is also stated that it is only a good idea to use Column Families when each key space is reasonably large, as maintaining multiple Column Families is a troublesome programming experience [44]. In the case of etcd 11 buckets are used:

- **root:** this bucket contains all other buckets.
- **key:** this is where all the key-value pairs stored in etcd are placed.
- **meta:** this bucket stores metadata, like the current consistent index, the next scheduled key space compaction, etc. The consistent index is a global, monotonically-increasing value, incremented for every update. It is also the offset of an entry in the consistent replicated log.
- **alarm:** this bucket stores a key for each activated cluster-wide alarm.
- **test:** this bucket is only used for testing purposes, not during normal operation of etcd.
- **members:** the IDs of active cluster members are stored here.

- **members_removed:** this bucket stores the IDs of members that have been removed from the cluster.
- **cluster:** the version of etcd is stored in this bucket.
- **auth:** authentication related metadata is stored here (e.g., whether authentication is activated or not).
- **authUser:** this bucket contains a key per user. Each user's name, password, and roles are stored in the corresponding value.
- **authRoles:** this bucket contains a key per role. The name and permissions of roles are stored in the value.
- **lease:** this bucket contains a key per active lease, containing information about their expiration times.

Obviously, the only bucket hosting a large key space is the "key" bucket. Therefore, prefixing was preferred in order to avoid the redundant programming complexity of using Column Families.

3.2.3 Get, Put & Delete Operations

Intuitively, a direct mapping can be applied as far as the core key-value store API is concerned. Both storage engines are key-value stores so they expose `Get()`, `Put()` and `Delete()` functions with the same semantics. In both cases, these operations will be applied within a transaction.

3.2.4 Cursor to Iterator

The case of mapping the BoltDB `Cursor` to a RocksDB concept is a simple one. The RocksDB `Iterator` implements the same basic functions required to iterate over a set of key-value pairs (e.g., `First()`, `Seek()`, `Next()`, `Prev()`, etc.).

3.2.5 Transactions

The possible RocksDB alternatives that we could map BoltDB transactions to include Pessimistic Transactions, Optimistic Transactions and a combination of `Snapshot` (to simulate read-only transactions) and `WriteBatch` (to simulate read-write transactions). Our first intuition was to map BoltBD transactions to RocksDB Pessimistic Transactions. In chapter 5 we will see how this approach led to suboptimal results and in chapter 4 we will try more efficient approaches and explain their correctness.

As has been made obvious by the description of the concept of a transaction in each storage engine in subsection 2.3.2 and subsection 2.4.2, the *semantics* differ slightly. Firstly, it is worth mentioning that unlike BoltDB, RocksDB supports running multiple concurrent transactions. BoltDB offers pure ACID transactions that do not fail, while RocksDB's Pessimistic Transactions might fail or deadlock (if no timeout has been set) if two of them simultaneously try to lock a specific key that the other has already acquired a lock for. Furthermore, RocksDB transactions offer a lower level of isolation (i.e., phantom writes are not detected [107]). Atomicity, consistency and durability are ensured by RocksDB transactions by design.

However, etcd maintains *only one* active read-write transaction at any given point. More specifically, in etcd a new transaction is created only once the previous one has been committed. This means that the semantic difference between the two storage engines can be safely disregarded. Failures of transactions or deadlocks will not happen, as there is no contention over keys when only one read-write transaction is running at a given time. In other words, the semantics of BoltDB transactions and RocksDB Pessimistic Transactions are the same in this context.

3.2.6 Snapshot to Checkpoint

As has been discussed in section 2.2, etcd uses snapshots as backups of its keyspace for the purpose of disaster recovery. The client can request the creation of a snapshot by issuing the command `etcdctl snapshot save backup.db`. The snapshot is then sent to the client over a stream. Later, if the cluster for example loses quorum, it can be reinitialized using the most recent snapshot available, with the command `etcdctl snapshot restore backup.db`. This command creates an etcd data directory for

an etcd cluster member from a backend database snapshot. Restoring overwrites some snapshot metadata (e.g., member ID, cluster ID) in order to reflect the new cluster configuration [106]. Those snapshots are also automatically created and sent over the network to new members joining the cluster or to slow followers in order to help them catch up.

In BoltDB a snapshot of the database can be taken with the `Tx.WriteTo()` function which writes a consistent view of the database to a file, as described in subsection 2.3.2. If called from a read-only transaction, `Tx.WriteTo()` performs a hot backup, which means that it does not block other reads and writes on the database [58].

When adopting RocksDB as the storage engine of etcd we have three options for the implementation of the snapshotting functionality, namely `Snapshot`, `Backup` and `Checkpoint`. These have all been analysed in subsection 2.4.2. After carefully evaluating each of them, we conclude that `Checkpoint` is the most suitable. Specifically, a `Snapshot` cannot be used by itself as it is a *logical* concept with no physical representation (e.g., a file) and is not persisted across database restarts. Theoretically, we could use a `Snapshot` to iterate over the keyspace of the database and copy the key-value pairs one by one to a new database, but this seems like an inefficient approach. Both `Backup` and `Checkpoint` satisfy our requirements as they perform a hot backup that is persistent. `Checkpoint` however, is preferred as it provides the extra benefit of *hard linking* the SST files instead of copying them, rendering it a more lightweight operation. Since SST files are immutable hard linking is very convenient. What is more, reference counting makes sure that SST files that belong to a `Checkpoint` will not be deleted when compaction removes them from the database directory. Even if the original database is deleted in its entirety the `Checkpoint` will remain intact. When opening a `Checkpoint` as a new database, and once compactions cause the original SST files to become obsolete, hard links are removed.

Furthermore, it is worth noting that throughout etcd code the snapshot is treated as a single file, matching the BoltDB snapshot concept. However, a RocksDB database and by extension its `Checkpoint` is a directory. To avoid extensive alteration of etcd code, we transform the `Checkpoint` to a *tar archive*[6]. This means that we end up copying the

[6]tar is a computer software utility for collecting many files into one archive file, often referred to as a tarball, for distribution or backup purposes. The name is derived from tape archive, as it was originally developed to write data to sequential I/O devices with no file system of their own. [108]

SST files in the archive. Even this way though, we benefit from using a `Checkpoint`, as in the case of a `Backup` the contents of the database would have to be copied twice.

3.2.7 Defragmentation

The need for regular defragmentation in the case of BoltDB has been explained in subsection 2.3.3. The `Defrag()` function forms part of etcd's backend API and is usually called manually by the user via the command line client (i.e., `etcdctl`), when a space quota alarm is raised, in order to free up some disk space and enable the resumption of normal cluster operation. Obviously, RocksDB as an LSM-tree implementation does not experience fragmentation. However, the most similar RocksDB concept to defragmentation is that of compaction, in the sense that both result in a reduction of the database's size on disk. A call to the function of BoltDB that implements defragmentation could easily be replaced by a manual compaction function call in RocksDB, but we deemed this to be unnecessary since compactions are already performed by RocksDB in a periodic fashion.

3.2.8 Read-Only Mode

BoltDB's `Open()` function accepts an argument that determines whether the database will be opened in read-write mode or in read-only mode. The underlying database is only opened in read-only mode by `etcdctl` when inquiring the status (i.e., hash, revision, number of keys, size) of a previously taken snapshot. When using RocksDB, in case that the aforementioned argument to `Open()` indicates opening in read-only mode, we use the `OpenForReadOnly()` function instead of the `Open()` function. Both databases can be simultaneously opened by multiple processes in read-only mode and by only one process in read-write mode.

The below table summarizes the differences between BoltDB and RocksDB that directly affect the design of our solution.

More details on the issues stemming from semantic discrepancies between BoltDB and RocksDB concepts and the way they were dealt with are presented in chapter 4.

	BoltDB	**RocksDB**
DB format	memory-mapped file	directory
Transactions	ACID	ACI*D (no phantom writes)
Concurrency	one txn at a time	multiple txns
Namespaces	buckets	prefixes / column families
Tuning	not required	multitude of tunable options
Defragmentation	required	not required

Table 3.1: *Key differences between BoltDB and RocksDB*

4 Implementation

In this chapter, we give a detailed description of the storage engine replacement process within the context of etcd and the development of the related BoltDB to RocksDB wrapper. We analyse the obstacles encountered while trying to put our design decisions into practice and the steps we took in order to overcome them. We then go on to present the rationale behind each optimization applied upon our initial approach. In addition, we outline the software patches we wrote in the cases that it was not possible to have a direct mapping because functionality available in BoltDB was missing from RocksDB. Finally, we describe in detail the methods and tools used to ensure the correctness of our code.

Below we have only included a few select segments of code, in order to aid the reader's understanding of our implementation. The complete source code is available on `https://github.com/boolean5/etcd-rocks`

4.1 Wrapping RocksDB in BoltDB

As it was described in subsection 3.1.1, in order to allow etcd code to continue using the same API as before, we kept the declarations of the BoltDB library functions intact and only changed their implementation, so that they call the corresponding gorocksdb functions. We only worked on the *subset* of the BoltDB API that etcd uses. In an analogous manner, the structs declared in the BoltDB package are retained, but they now act as *wrappers* for the corresponding gorocksdb structs, and other fields are added

to or removed from them as needed. Also, whenever the mapping from BoltDB to RocksDB concepts was not trivial we added extra code and auxiliary functions.

In this section we provide a complete list of the functions and structs that were implemented, along with an explanation. This list refers to the final, *optimized* implementation of our wrapper. Later on, in section 4.3, we will present our path towards this final version, starting from our base implementation and reviewing the optimizations applied on it, highlighting the changes made on our functions and structs.

`DB struct`: this struct represents the database. Its main purpose is to act as a wrapper for the underlying gorocksdb `DB struct`, which is a reusable handle to a RocksDB database on disk. Besides that, its fields include information about the path of the database directory, and whether the database has been opened in read-only mode. They also include a mutex responsible for allowing only one writer at a time, a handle to an `Options struct` and a map that contains the buckets that exist in the database[1]. Finally, among the fields of the `DB struct` there is a pointer to a gorocksdb `WriteBatch`, which is used to simulate a BoltDB transaction, and a pointer to a gorocksdb `WriteBatchWithIndex`, which is only used in the special case mentioned in section 4.2. For a description of the `WriteBatch` and `WriteBatchWithIndex` concepts, see subsection 2.4.2.

```
type DB struct {
        readOnly        bool
        db              *gorocksdb.DB
        path            string
        rwlock          sync.Mutex
        wb              *gorocksdb.WriteBatch
        wbwi            *gorocksdb.WriteBatchWithIndex
        options         *Options
        buckets         map[string]bool
}
```

Listing 4.1: *The `DB struct`*

[1]The bucket hierarchy present in BoltDB is preserved, even though RocksDB does not inherently provide such a concept.

`Open(path string, mode os.FileMode, options *Options) (*DB, error)`: this function opens an instance of RocksDB at the given path. If a directory does not already exist at the provided path it will be created automatically. Also, depending on the value of the `mode` argument, which is processed by the `isReadOnly()` function, the database will be opened either in read-only mode by calling the gorocksdb function `OpenDbForReadOnly()` or in normal read-write mode by calling the gorocksdb function `OpenDb()`. Right before that, the appropriate database `Options` are created.

It is worth noting here that while this function is usually invoked with the purpose of creating a new database, it might also be used to open an existing database from a regular database directory (i.e., when an etcd node is restarted) or from a `Checkpoint` (i.e., when recovering a cluster). In the latter case, instead of pointing to a directory as usual, the `path` argument points to a tar archive that needs to be *untarred* before we attempt to open it. To differentiate among these cases, we implemented a check using Go's built-in functions `Stat()` and `IsDir()` from the os package. `Stat()` returns the `FileInfo` associated with a path, which is then used by `IsDir()` to report whether the path points to a directory [97]. In the latter case, we can immediately proceed to the call of gorocksdb that opens the database. On the other hand, if it points to a file, we use our `IsTar()` function to confirm that the file is indeed a tar archive, and then we untar it via our `untar()` function, rename the resulting directory properly and delete the tar archive, before proceeding to open the database.

Before returning, the map of buckets of the `DB struct` is updated, so that it will include the buckets present in the database if the latter was not created by this call to `Open` but it already existed. `Open` returns a `DB struct`, and an error in case any of the functions it calls fail, or if the provided path points to a file that is not a tar archive, or to a tar archive that does not contain a `Checkpoint`.

`(db *DB) Close() error`: this function closes the RocksDB instance and releases all resources. More specifically, it destroys the `WriteBatch`, the `WriteBatchWithIndex` and the `Options` that are associated with the database, by calling the appropriate gorocksdb functions. After that, it calls the gorocksdb function `Close()` on the database. Throughout this process, it holds the writer *lock* of the `DB struct`, to make sure that no other goroutine attempts to use the released resources or write to the database while it is being closed. Even though the initial BoltDB `Close` function re-

turns an error, our implementation does not contain any operations that can fail, so it always returns `nil`.

`(db *DB) Begin(writable bool) (*Tx, error)`: This function creates a RocksDB `WriteBatch` as a means to simulate a BoltDB transaction. Depending on the value of the `writable` argument, one of two functions is called: either `beginRWTx()` to create a read-write transaction or `beginROTx()` to create a read-only transaction.

While `Begin` is most often called to start a new transaction, sometimes it is invoked by etcd in order to provide a snapshot of the database. In the context of BoltDB a snapshot can be perceived as a consistent, read-only view of the database, which is the definition of a BoltDB transaction. In our case however, we need to differentiate between these two circumstances, so we implement a check that determines who the caller of the `Begin` function is. If the caller is identified as the `Snapshot` function from the `backend` package of etcd, a call to `createCheckpoint()` follows. To implement this check we utilize the `Caller()` and `CallersFrames()` functions from Go's `runtime` package. The former returns information about function invocations on the calling goroutine's stack, including a program counter, while the latter, based on this program counter value, returns a stack frame that contains the name and file of a function [97].

The `Begin` function returns a pointer to a `Tx struct` and an error which is only non-nil in case one of the called functions fails. If a checkpoint has been created, its path is stored in the appropriate field of the `Tx struct`. Whenever an error is returned the `Tx` pointer is `nil`. When `createCheckpoint()` fails, we make sure to roll back the transaction that has already been created.

```
1  func (db *DB) Begin(writable bool) (*Tx, error) {
2        var (
3              err error
4              tx *Tx
5        )
6        if writable {
7              tx, err = db.beginRWTx()
8        } else {
9              tx, err = db.beginROTx()
10       }
```

```
11        if err != nil {
12              return nil, err
13        }
14
15        pc, _, _, ok := runtime.Caller(1)
16        if !ok {
17              _ = tx.Rollback()
18              return nil, ErrRuntimeCaller
19        }
20        pcs := make([]uintptr, 1)
21        pcs[0] = pc
22        frames := runtime.CallersFrames(pcs)
23        frame, _ := frames.Next()
24        if pathpkg.Base(frame.Function) ==
              "backend.(*backend).Snapshot" {
25              tx.checkpoint, err = tx.createCheckpoint()
26              if err != nil {
27                    _ = tx.Rollback()
28                    return nil, err
29              }
30        }
31        return tx, nil
32  }
```

Listing 4.2: *The `Begin` function*

`(db *DB) beginRWTx() (*Tx, error)`: this function first obtains the writer lock associated with the `DB struct` to enforce the existence of only one writer transaction (i.e., `WriteBatch`) at a time. This lock is released when the transaction is committed or rolled back. Then, `beginRWTx` allocates a new `Tx struct` and initializes its fields with the appropriate values, including a pointer to the `WriteBatch` that will be used to simulate the read-write transaction. In order to boost performance, we only allocate a new `WriteBatch` the first time `beginRWTx` is called, via calling the gorocksdb function `NewWriteBatch()`. In subsequent calls to `beginRWTx` we can reuse the same `WriteBatch` which we have made sure to *clear* upon the commitment or rolling back

of its associated read-write transaction. `beginRWTx` returns a pointer to a `Tx` struct and an error. The error is only non-nil if the database has been opened in read-only mode.

`(db *DB) beginROTx() (*Tx, error)`: this function allocates a new `Tx struct` and initializes its fields with the appropriate values. We simulate a BoltDB read-only transaction by using a RocksDB `Snapshot` which provides a consistent, read-only view of the database and thus has the same semantics. The new `Snapshot` is created by calling the gorocksdb function `NewSnapshot()`, and with the `SetSnapshot()` gorocksdb function it is associated with the gorocksdb `ReadOptions` struct, freshly created by the `createReadOptions()` function. A pointer to this gorocksdb `ReadOptions struct` is kept in one of the `Tx struct` fields. `beginROTx` returns a pointer to a `Tx` struct and an error, which is always `nil`.

`(db *DB) View(fn func(*Tx) error) error`: this function provides a way for the function that it receives as an argument to be executed in the context of a read-only transaction. It is used by etcd to iterate through the buckets and keys of its backend with the purpose of printing a list of them, or computing a hash value. In the context of BoltDB, the `View` function offers a managed way to work with read-only transactions, as opposed to the manual way that involves direct calls to `Begin()`, `Commit()` and `Rollback()` functions. First, a new transaction is started by a call to `Begin()`. Subsequently, the function-argument is called. This function must receive a pointer to a `Tx struct` as its unique argument and return an error. Finally, `Rollback()` is called[2]. By using Go's `defer` statement we make sure that `Rollback()` will be called on the transaction even in the event of a panic in the function-argument. `View` returns an error in case one of the functions it calls returns an error.

`Tx struct`: this struct represents an active transaction. Its main purpose is to act as a wrapper for the underlying `RocksDBWriteBatch interface`, which serves to simulate a BoltDB transaction. Besides that, its fields include information about whether the transaction is read-write or read-only, whether the transaction is simulated by a `WriteBatch` or a `WriteBatchWithIndex`[3], and the path of the `Checkpoint` tar file, if the transaction has been created as a result of a call to the `Snapshot()` function of the `backend` package of etcd. Among its fields, there is also a pointer to the `DB`

[2] `Commit()` is only called to close read-write transactions.
[3] For an explanation of these two cases see section 4.2.

`struct` of the database, a pointer to the gorocksdb `Snapshot` that is used to simulate the transaction if it is read-only and serves as a handle for release when the transaction is rolled back, a pointer to the gorocksdb `ReadOptions` that contain the aforementioned `Snapshot` and a `Bucket struct` that serves as the `root` bucket. Finally, one of the struct's fields contains a slice that holds all the gorocksdb `Iterators` allocated during the transaction's lifetime, so that they can be released when the transaction is committed or rolled back.

```
type Tx struct {
        writable      bool
        db            *DB
        index         bool
        wb            RocksDBWriteBatch
        snapshot      *gorocksdb.Snapshot
        ro            *gorocksdb.ReadOptions
        root          Bucket
        checkpoint    string
        iterators     []*gorocksdb.Iterator
}
```

Listing 4.3: *The `Tx struct`*

`(tx *Tx) Switch() error`: this function switches between a `WriteBatch` and a `WriteBatchWithIndex`. Even though in the final, optimized version of our implementation we simulate read-write transactions using the `WriteBatch` concept, seeing considerable performance benefits over the version that uses `WriteBatchWithIndex`[4], there exists a corner case where `WriteBatchWithIndex` is still necessary. This corner case is explained in detail in section 4.2 and `Switch` is called from etcd code whenever it is encountered.

The switch is implemented by iterating over the `WriteBatch`'s records and replaying the `Put` and `Delete` actions upon a new `WriteBatchWithIndex`. This `WriteBatchWithIndex` is only allocated the first time it is needed and is later reused, making sure it is cleared every time its associated transaction is committed or rolled back. After the

[4]For more details on this see subsection 4.3.5

"copying" is complete, the `WriteBatchWithIndex` replaces the `WriteBatch` in the `Tx struct`'s `wb` field.

`Switch()` returns an error when the transaction has already been closed or when the `WriteBatch Iterator` returns an error.

`(tx *Tx) Size() int64`: if the transaction has been created as a result to a call to the `Snapshot()` function of the `backend` package and has an associated `Checkpoint`, this function returns the size of this `Checkpoint` in tar archive form, as computed by the function `tarSize()`. If the transaction has no associated `Checkpoint` then the return value represents the size of the database directory on disk, as computed by the function `rocksdbSize()`. `Size` is called by etcd whenever a transaction is created and committed, to update the relevant field of its `backend struct`. It is also called to compute the size of an etcd `Snapshot` when the client issues a `etcdctl snapshot status backup.db` command.

`(tx *Tx) Cursor() *Cursor`: this function returns a `Cursor struct` that can be used to iterate over the prefixes that exist in the database. In the context of BoltDB, this function returns a `Cursor` associated with the `root Bucket`. In the context of our implementation it returns a `Cursor` able to iterate over the keys that are prefixed by "root". For details on the mapping of buckets to prefixes see subsection 3.2.2. The implementation of this function consists only of a call to the `(b *Bucket) Cursor() *Cursor` function and the return of its result.

`(tx *Tx) Rollback() error`: this function closes the ongoing transaction and discards all previous updates. Read-only transactions must be rolled back and not committed. If the transaction is a read-write one, we implement `Rollback` by clearing the underlying `WriteBatch` via a call to the gorocksdb `Clear()` function on it. In addition, upon rolling-back a read-write transaction, the writer lock of the `DB struct` is released. If the transaction is read-only, to implement `Rollback` it is enough to release the underlying `Snapshot` and destroy the `ReadOptions` that contain it. Moreover, if the transaction has an associated `Checkpoint`, the related tar archive is removed using Go's `Remove()` function from the `os` package. We also make sure to remove the `Checkpoint` directory if the tar archive has been untarred. To that end, we use Go's functions `filepath.Abs()` and `os.RemoveAll()`. Finally, in both cases we close all `Iterators` used during the lifetime of the transaction. `Rollback` returns an error

if the transaction has already been closed, or if any of the functions it calls return an error.

`(tx *Tx) Commit() error`: this function writes all transaction changes to disk. It is implemented by calling the gorocksdb function `Write()` or `WriteWithIndex()` to persist changes made on the underlying `WriteBatch` or `WriteBatchWithIndex` respectively. After that, the `WriteBatch` or `WriteBatchWithIndex` is cleared and the writer lock of the `DB struct` is released. Finally, all `Iterators` used during the lifetime of the transaction are closed. `Commit` returns an error if the transaction has already been closed, if it is called on a read-only transaction or if `Write()` or `WriteWithIndex()` fails.

```
1  func (tx *Tx) Commit() error {
2        if tx.db == nil {
3              return ErrTxClosed
4        } else if !tx.writable {
5              return ErrTxNotWritable
6        }
7        // rocksdb commit
8        if !tx.index {
9              wb, _ := tx.wb.(*gorocksdb.WriteBatch)
10             if err := tx.db.db.Write(tx.db.options.writeOptions,
                   wb); err != nil {
11                 return err
12             }
13       } else {
14             wb, _ := tx.wb.(*gorocksdb.WriteBatchWithIndex)
15             if err :=
                   tx.db.db.WriteWithIndex(tx.db.options.writeOptions,
                   wb); err != nil {
16                 return err
17             }
18       }
19       tx.wb.Clear()
20       // release writer lock
21       tx.db.rwlock.Unlock()
```

```
22
23          tx.db = nil
24          tx.wb = nil
25          tx.ro = nil
26          tx.snapshot = nil
27          tx.checkpoint = ""
28          tx.root = Bucket{tx: tx}
29
30          // close iterators
31          for _, iter := range tx.iterators {
32                iter.Close()
33          }
34          tx.iterators = nil
35
36          return nil
37    }
```

Listing 4.4: *The `Commit` function*

`(tx *Tx) Bucket(name []byte) *Bucket`: in the context of BoltDB this function retrieves and returns a `Bucket struct` by name. In the context of our implementation, we simply check the map of buckets of the `DB struct` to confirm the the bucket has previously been created and we return a freshly allocated `Bucket struct` with the requested name. If the `Bucket` is not found in the map `nil` is returned. The initial, non-optimized version of this function involved a call to function `Get()`, applied on the `root Bucket`, to retrieve the requested `Bucket` from the database. For the rationale supporting this change see subsection 4.3.2.

`(tx *Tx) ForEach(fn func(name []byte, b *Bucket) error) error`: it executes the function that it receives as its argument for every `Bucket` in the `root`. If at any point the provided function returns an error, the iteration is stopped and the error is returned to the caller. It is implemented as a call to `(b *Bucket) ForEach(fn func(k, v []byte) error) error`, using as its argument a function that executes `fn` on the current `Bucket` and checks if the error returned from `fn` is non-nil.

`(tx *Tx) CreateBucket(name []byte) (*Bucket, error)`: this function creates a new `Bucket` in the database. In our implementation the `Put()` function is called to write the new `Bucket` to the database as a key-value pair with key equal to the concatenation of "root" with `name` and value "-". Then, the new bucket is added to the map of existing buckets of the `DB struct`. Finally, a newly allocated `Bucket struct` with the requested name is returned. `CreateBucket` returns an error if the `Bucket` already exists, if the bucket name is blank, if the transaction has been closed or if the transaction is read-only. In addition, an error is returned if the `Put()` function fails.

`(tx *Tx) createCheckpoint() (string, error)`: this function creates a gorocksdb `Checkpoint` by calling the gorocksdb function `NewCheckpointObject()` and consequently calling the gorocksdb function `Create()` on this newly created object. In RocksDB a `Checkpoint` object needs to be created for a database before checkpoints are created. `createCheckpoint()` is called by `Begin()` when it has been invoked by the `Snapshot()` function from the `backend` package. The path of the `Checkpoint` is constructed from the absolute path of the database directory by replacing its last component (i.e., the name of the database directory) with "Checkpoint" and appending a timestamp. This ensures that there will not be any path conflicts among different checkpoints. Subsequently, the `Checkpoint` directory is transformed into a tar archive by a call to `tarit()` function and a `.tar` extension is added to the path. Finally, the `Checkpoint` object is destroyed. An alternative would be to retain the object for the creation of future checkpoints. However, even if `createCheckpoint()` happens to be called multiple times during the lifetime of an etcd node, in the general case these calls will be seldom enough to render the overhead of recreating the `Checkpoint` object every time negligible[5]. This function returns a `string` that represents the path of the `Checkpoint` tar archive (e.g., `/tmp/test/member/snap/Checkpoint2017-06-28 12:21:34.34229393 +0300 EEST`) and an error which is non-nil if one of the called function fails. `createCheckpoint` is not part of the BoltDB API, it is a function we implemented to avoid excessively complicating the code of `Begin()` with the addition of the `Checkpoint` creation operations.

`(tx *Tx) WriteTo(w io.Writer) (n int64, err error)`: this is the function that BoltDB uses to write a consistent view of the database to a writer. An `io.Writer` is an interface that includes anything that implements the `Write()` method (e.g., a

[5]For the usage of checkpoints in etcd see the paragraph on Disaster Recovery in subsection 2.2.1

file, a pipe, a buffer, a `stdout`). In our case the `Writer` is an `io.PipeWriter`, which is the write end of a synchronous in-memory pipe. This kind of pipe is often used in Go to connect code expecting an `io.Reader` with code expecting an `io.Writer`.

In order to better understand the role of the `WriteTo` function it is important to outline the general process followed to send a snapshot to the client. First, the client requests a snapshot by issuing the command `etcdctl snapshot save backup.db`. On the client's side, a file is created at the given path (i.e., the last argument of the command) and a snapshot request is sent to the server over a gRPC stream. In addition, a pipe is created. The initial goroutine reads from the pipe's `io.PipeReader` end (pr) and writes to the file, while a separate goroutine reads the servers response from the gRPC stream and writes it to the pipe's `io.PipeWriter` end (pw).

On the server's side, when the snapshot request is received over the gRPC stream, a snapshot of the backend is created (i.e., a `Checkpoint` in our implementation). Subsequently, a pipe is created. A new goroutine is started and it calls `WriteTo()` with the pipe's `io.PipeWriter` end (pw) as its argument, while the initial goroutine reads from the pipe's `io.PipeReader` end (pr) and sends the contents to the client over the gRPC stream.

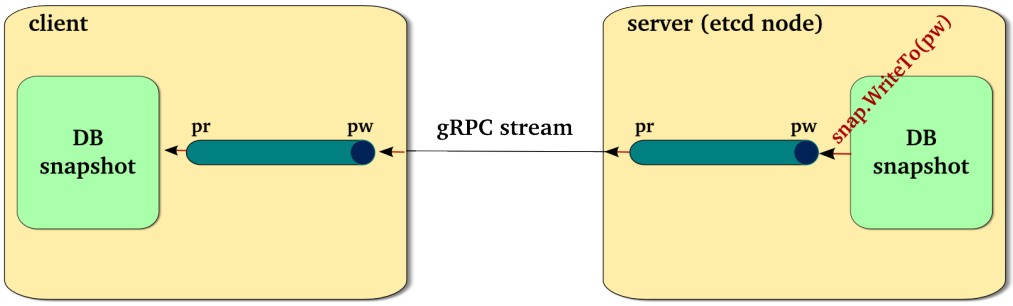


Figure 4.1: *The role of the `WriteTo()` function*

In our implementation of the `WriteTo` function we open the `Checkpoint` file (i.e., the tar archive) associated with the transaction and inside a for loop we gradually read its contents into a buffer and then copy them using the `io.Writer`'s `Write()` method until end-of-file is reached[6]. As etcd always calls `WriteTo` from a read-only transaction, it performs a hot backup and does not block other database reads and writes. The

[6] An improvement over this would be to direct the output of the *tarring* procedure straight to the pipe upon creation of the `Checkpoint`, thus avoiding the intermediate copy. This would require modifications in etcd code.

return values are the number of bytes written and an error. If the error is equal to nil then exactly `tx.Size()` bytes will be written to the writer. The error is non-nil when the function is called on a transaction that does not have an associated `Checkpoint` or when one of the called functions fails.

```
func (tx *Tx) WriteTo(w io.Writer) (n int64, err error) {
	if tx.checkpoint == "" {
		return 0, ErrNoCheckpoint
	}
	var num int64
	fi, err := os.Open(tx.checkpoint)
	if err != nil {
		return 0, err
	}
	defer func() {
		if err = fi.Close(); err != nil {
			panic(err)
		}
	}()
	buf := make([]byte, 1024)
	for {
		n, err := fi.Read(buf)
		if err != nil && err != io.EOF {
			return num, err
		}
		if n == 0 { break }
		n2, err := w.Write(buf[:n])
		num = num + int64(n2)
		if err != nil {
			return num, err
		}
	}
	return num, nil
}
```

Listing 4.5: *The `WriteTo` function*

`Bucket struct`: this struct represents a bucket in the database. As we have seen in subsection 3.2.2 the concept of buckets does not exist in RocksDB, so in order to ensure the required namespace separation among keys we use prefixes. Thus, the first field of this struct contains the prefix of the specific bucket, which is no other than the bucket's name in the form of a byte slice. The second field of the `Bucket struct` contains a pointer to the associated `Tx struct`. A bucket is always accessed in the context of a transaction and in a multitude of cases we need a handle to this transaction (e.g., in the `Put()` function, where we need access to the transaction's associated `WriteBatch`). Finally, the last field of the `Bucket struct` is not used at all by our implementation and is only retained for compatibility reasons, so as to avoid changing etcd code that uses it. More specifically, this is a `float` value that in BoltDB determines the fill percentage of a bucket. The fill percentage is a concept specific to BoltDB and B+ trees that sets the threshold for filling nodes before they split[7] and is not relevant in RocksDB.

`(b *Bucket) Cursor() *Cursor`: this function creates and returns a `Cursor struct` associated with the bucket it is called on. In our implementation a gorocksdb `Iterator` is created via a call to the gorocksdb `NewIterator()` function, with the `ReadOptions` of the associated transaction. This means that if the transaction is read-only the `Iterator` will operate on its underlying RocksDB `Snapshot`. However, in case that the transaction uses an underlying `WriteBatchWithIndex` instead of a `WriteBatch`, the creation of the `Iterator` involves two steps. First, we create the base iterator, which is just a normal `Iterator` over the database. Then, we call the gorocksdb function `NewIteratorWithBase()`, which uses the aforementioned base iterator to construct an `Iterator` that merges the results of RocksDB with those of the `WriteBatchWithIndex` 's buffer. Finally, in both cases the newly created `Iterator` is appended to the transaction's list of iterators so that it can later be closed. This function will panic in case the bucket's associated transaction has already been closed, as its declaration in BoltDB does not allow us to return an error value.

`(b *Bucket) ForEach(fn func(k, v []byte) error) error`: this function executes the function that it receives as its argument for each key-value pair in a bucket. If at any point the provided function returns an error, the iteration is stopped and the error is returned to the caller. This function also returns an error if the associated

[7] It can be useful to increase the fill percentage if it is known that workloads are mostly append-only [58]. Etcd code sets this value to 0.9 when performing sequential writes.

transaction has been closed. The implementation of `ForEach` consists of creating a new `Cursor` and using it to iterate over the key-value pairs of the bucket it has been called on and call the function-parameter on each of those pairs, with the appropriate error checking. For this iteration the functions `First()` and `Next()` are used. Interestingly, as this function's initial implementation was made up only of calls to the BoltDB API we did not have to change it at all.

`(b *Bucket) Put(key []byte, value []byte) error`: this function sets the value for a key in a specific bucket. It basically works as a wrapper for the underlying gorocksdb `Put()` function which is called on the associated transaction's `WriteBatch` or `WriteBatchWithIndex`. The key that is actually put in the database results from the concatenation of the `Bucket`'s prefix, the "/" separator and the key that was passed as an argument to this function, which is performed by the `concatenate()` function. `Put` returns an error if the transaction has been closed, if it is read-only, if the key is blank, if the key is too large or if the value is too large. The maximum recommended key size in RocksDB is 8MB, while the maximum recommended value size is 3GB.

```
1  func (b *Bucket) Put(key []byte, value []byte) error {
2        if b.tx.db == nil {
3              return ErrTxClosed
4        } else if !b.tx.writable {
5              return ErrTxNotWritable
6        } else if len(key) == 0 {
7              return ErrKeyRequired
8        } else if len(key) > maxKeySize {
9              return ErrKeyTooLarge
10       } else if len(value) > maxValueSize {
11             return ErrValueTooLarge
12       }
13       s := make([][]byte, 3)
14       s[0] = b.prefix; s[1] = sep; s[2] = key
15       b.tx.wb.Put(concatenate(s), value)
16       return nil
17 }
```

Listing 4.6: *The `Put()` function*

`(b *Bucket) Get(key []byte) []byte`: this function retrieves the value for a key in a specific bucket. If the key does not exist, `nil` is returned. It basically works as a wrapper for the underlying gorocksdb `GetBytes()` function, passing as an argument to it the `ReadOptions` of the associated transaction. This means that if the transaction is read-only `GetBytes()` will operate on its underying RocksDB `Snapshot`. However, in case that the transaction uses an underlying `WriteBatchWithIndex` instead of a `WriteBatch` (i.e., `Get` has been called in the context of a read-write transaction) we call `GetBytesFromBatchAndDB()` instead of plain `GetBytes()`. The former looks for the key in the `WriteBatchWithIndex`'s buffer before checking the database. The key that we look for is produced from the concatenation of the `Bucket`'s prefix, the "/" separator and the key that was passed as an argument to this function. This function will panic in case the bucket's associated transaction has already been closed or in case its equivalent underlying gorocksdb function fails, as its declaration in BoltDB does not allow us to return an error value.

```
1  func (b *Bucket) Get(key []byte) []byte {
2        if b.tx.db == nil {
3              panic(fmt.Sprintf("Get error (%s)", ErrTxClosed))
4        }
5        var (
6              err error
7              v []byte
8        )
9        s := make([][]byte, 3)
10       s[0] = b.prefix; s[1] = sep; s[2] = key
11       if b.tx.db.readOnly || !b.tx.writable || !b.tx.index {
12             v, err = b.tx.db.db.GetBytes(b.tx.ro, concatenate(s))
13       } else {
14             wb, _ := b.tx.wb.(*gorocksdb.WriteBatchWithIndex)
15             v, err = wb.GetBytesFromBatchAndDB(b.tx.db.db,
                   b.tx.db.options.readOptions, concatenate(s))
16       }
17       if err != nil {
18              panic(fmt.Sprintf("Get error: %v", err))
19       }
```

```
20          return v
21  }
```

Listing 4.7: *The `Get()` function*

`(b *Bucket) Delete(key []byte) error`: this function removes a key from a specific bucket. If the key does not exist then nothing is done and a nil error is returned. An error is returned if the transaction has been closed or if the bucket was created from a read-only transaction. This function basically works as a wrapper for the underlying gorocksdb `Delete()` function, which is called on the associated transaction's `WriteBatch` or `WriteBatchWithIndex`. The key that is actually deleted from the database results from the concatenation of the `Bucket`'s prefix, the "/" delimiter and the key that was passed as an argument to this function.

`Cursor struct`: this struct acts as a wrapper for the underlying gorocksdb `Iterator`. It also has a field that contains a pointer to a `Bucket struct`, the bucket that the `Cursor` is associated with.

`(c *Cursor) First() (key []byte, value []byte)`: this function moves the `Cursor` to the first key of its associated `Bucket` and returns that key-value pair. In our implementation this is the first key in the database with a specific prefix (i.e., the prefix of the `Bucket` the `Cursor` operates on). If the bucket is empty then a nil key and value are returned. The prefix is constructed by appending the "/" delimiter to the `Bucket`'s name. Then, we use the gorocksdb `Seek()` function on the `Cursor`'s underlying gorocksdb `Iterator` to look for the first occurrence of this prefix in the database. Right after that, we use the gorocksdb function `ValidForPrefix()` to check whether the `Iterator` has stopped at a key that has the requested prefix, in other words, if there exists at least one key with this prefix in the database. If yes, we obtain the current key-value pair of the `Iterator` via calls to the gorocksdb `Key()` and `Value()` functions. As those two functions return gorocksdb structs (i.e., `Slice structs`) that contain pointers to the key and value in memory allocated by C, we need to copy the key and value in Go byte slices and subsequently free the C data. Finally, the `Err()` function of the gorocksdb `Iterator` is used to check if any errors occurred during the usage of the `Iterator`.

Before returning the appropriate key-value pair, the prefix and delimiter are removed

from the key's name with the aid of Go's function `SplitN()` from the `bytes` package. `SplitN()` splits the byte slice that it receives as its first argument into as many subslices as its third argument determines, separated by the delimiter defined from its second argument [97]. `First` will panic in case the associated transaction has already been closed or in case the `Iterator` returns an error, as its declaration in BoltDB does not allow us to return an error value.

```
1  func (c *Cursor) First() (key []byte, value []byte) {
2        if c.bucket.tx.db == nil {
3              panic(fmt.Sprintf("Cursor error (%s)", ErrTxClosed))
4        }
5        // move the cursor to the first element of its corresponding
             bucket
6        s := make([][]byte, 2)
7        s[0] = c.bucket.prefix; s[1] = sep
8        prefix := concatenate(s)
9        c.rocksdbIterator.Seek(prefix)
10       if !c.rocksdbIterator.ValidForPrefix(prefix) {
11             return nil, nil
12       }
13       k := c.rocksdbIterator.Key()
14       key = make([]byte, k.Size())
15       copy(key, k.Data())
16       k.Free()
17
18       v := c.rocksdbIterator.Value()
19       value = make([]byte, v.Size())
20       copy(value, v.Data())
21       v.Free()
22
23       if err := c.rocksdbIterator.Err(); err != nil {
24             panic(fmt.Sprintf("Iterator error: %v", err))
25       }
26
27       // remove prefix
```

```
28          s = bytes.SplitN(key, sep, 2)
29
30          return s[1], value
31  }
```

Listing 4.8: *The `First()` function*

`(c *Cursor) Last() (key []byte, value []byte)`: this function moves the `Cursor` to the last key of its associated `Bucket` and returns that key-value pair. This function is generally analogous to the `First()` function. In this case however, we iterate to the last key with the prefix of the given `Bucket`. To do that we first seek the first key after the keys that have the requested prefix, with a `for` loop based on the gorocksdb functions `Seek()`, `ValidForPrefix()` and `Next()`. Once we reach that key, we call the gorocksdb function `Prev()` and once again use `ValidForPrefix()` to eliminate the case of not having any keys with the requested prefix in the database.

`(c *Cursor) Next() (key []byte, value []byte)`: this function moves the `Cursor` to the next item in the `Bucket` and returns its key and value. If the `Cursor` is at the end of the bucket then a nil key and value are returned. This function acts as a wrapper for the underlying gorocksdb `Next()` function which is applied on the gorocksdb `Iterator` associated with the `Cursor`. To check if we have iterated past the last key in the `Bucket` or the last key in the database we call the gorocksdb `ValidForPrefix()` function, where the prefix is constructed by appending the "/" delimiter to the `Bucket`'s name. The rest of the implementation is identical to that of function `First()`.

`(c *Cursor) Prev() (key []byte, value []byte)`: this function moves the `Cursor` to the previous item in the `Bucket` and returns its key and value. If the `Cursor` is at the beginning of the bucket then a nil key and value are returned. `Prev`'s implementation is completely analogous to that of `Next()`, the only difference being that it is a wrapper for the gorocksdb `Prev()` function applied on the gorocksdb `Iterator` associated with the `Cursor`.

`(c *Cursor) Seek(seek []byte) (key []byte, value []byte)`: this function moves the `Cursor` to the key passed as its argument and returns the key and its value. If the key does not exist then the next key is used. If no keys follow in the `Bucket`,

a nil key and value is returned. `Seek` works as a wrapper for the gorocksdb `Seek()` function, applied on the underlying gorocksdb `Iterator` of the `Cursor`. The key passed as an argument to the gorocksdb `Seek()` function is constructed by appending the "/" delimiter and the actual name of the key to the `Bucket`'s name in that order. Other than that, its implementation is analogous to that of function `First()`.

`Options struct`: this struct contains all the kinds of options used by RocksDB, plus two fields accessed by etcd code, which have only been included in order to avoid changing code outside the `bolt` package. The latter are two `int` fields, `MmapFlags` and `InitialMmapSize`, that contain parameters related to the memory-mapped file of BoltDB and are not relevant in the context of RocksDB. The different kinds of `Options` required by RocksDB include gorocksdb `Options`, `WriteOptions` and `ReadOptions`.

`(opts *Options) createAllOptions() *Options`: this function creates all the kinds of options used by RocksDB and is called before opening a database. More specifically, it contains calls to the functions `createOptions()`, `createWriteOptions()` and `createReadOptions()` and it stores pointers to the created RocksDB options in the appropriate fields of the `Options struct` that it returns. `createAllOptions()`, as well as the following option-related functions, do not form part of the BoltDB API. BoltDB, which is far less tunable than RocksDB, is started with the default options and whenever something needs to be changed it is done by directly accessing the appropriate field of its `Options struct`.

`(opts *Options) destroyAllOptions()`: this function destroys all the options associated with a RocksDB database and it is called right before closing it. More specifically, it destroys the gorocksdb `Options`, `BlockBasedTableOptions`, `Env`[8], `WriteOptions` and `ReadOptions` by calling the gorocksdb `Destroy()` function for each of them and assigning `nil` to the `Options struct`'s fields.

`createOptions() *gorocksdb.Options`: this function creates and returns the necessary options to open a RocksDB instance. The implementation starts with a call to the gorocksdb function `NewDefaultOptions()` which returns a new gorocksdb `Options struct`. Then, the gorocksdb function `SetCreateIfMissing(true)` is applied on those newly created options to make sure that gorocksdb `OpenDb()` will

[8]For details on `BlockBasedTableOptions` and `Env` see subsection 4.3.6.

create a new database directory if one does not already exist at the given path. The largest part of this function implements RocksDB parameter *tuning*. For an analysis of our tuning methodology and a detailed description of each parameter's role see subsection 4.3.6. Listing 4.9 contains the source code of this function.

`createWriteOptions() *gorocksdb.WriteOptions`: this function calls the gorocksdb `NewDefaultWriteOptions()` function and returns the created gorocksdb `WriteOptions`. `WriteOptions` are passed as an argument to the gorocksdb functions `Write()` and `WriteWithIndex()`. The most important `WriteOptions` determine whether a write will be flushed from the operating system buffer cache before it is considered complete or whether a particular write should first be written to the WAL of RocksDB. Our implementation always uses the default `WriteOptions`. This means that writes are synchronously written to the WAL before being considered complete[9].

`createReadOptions() *gorocksdb.ReadOptions`: this function calls the gorocksdb `NewDefaultReadOptions()` function and returns the created gorocksdb `ReadOptions`. `ReadOptions` are passed as an argument to the gorocksdb functions `GetBytes()`, `GetBytesFromBatchAndDB()` and `NewIterator()`. The `ReadOptions` may contain a RocksDB `Snapshot`. In our implementation the default `ReadOptions` are always used, except for the case of read-only transactions, where we use the gorocksdb `SetSnapshot()` function on the `ReadOptions` to set the `Snapshot` that should be used for reads.

`concatenate(slices [][]byte) []byte`: this function is usually used to perform the concatenation of a `Bucket`'s prefix, the "/" delimiter and the key that was passed as an argument to functions like `Put()`, `Get()` and `Seek()`. First, it iterates through the slice of byte slices passed as its argument, computing the sum of lengths of the byte slices. Subsequently, it allocates a new slice with length equal to the computed sum, and once again iterates through the slice of byte slices, copying one after the other into the newly allocated byte slice. Finally, it returns this byte slice. `concatenate`, as well as the following three functions, are auxiliary functions implemented by us and are not part of the BoltDB API.

`tarSize(path string) (int64, error)`: this function computes and returns the size of the tar archive at the given `path`. It uses Go's `Stat()` function from the `os` pack-

[9]For more information on this choice see subsection 4.3.6

age and then applies the `Size()` function on the returned `FileInfo` [97]. `tarSize` is called by `tx.Size()` to report the size of the transaction's associated `Checkpoint`.

`rocksdbSize(database *DB) (int64, error)`: this function returns an *estimation* of the on-disk size of the RocksDB database. It is called by `tx.Size()`, which in turn is called whenever a transaction is created and committed to update the relevant field of the `backend struct`. Then, `backend.size` is used by etcd code to trigger the backend quota alarm. Since in our implementation we have decided to remove the storage restriction (see subsection 3.1.5), the accuracy of `rocksdbSize`'s result is not crucial.

Theoretically, the size of a RocksDB database on disk can be computed as the sum of the sizes of the SST files, the WAL and the rest of the files present in the database directory, namely `LOG`, `MANIFEST`, `OPTIONS`, `CURRENT`, `IDENTITY` and `LOCK`[10].

Our first approach to the implementation of this function was based on calls to Go's `Walk()` function from the `filepath` package. According to Go's documentation, `Walk(root string, walkFn WalkFunc) error` "walks the file tree rooted at `root`, calling `walkFn` for each file or directory in the tree, including `root`" [97]. In our case, `walkFn` would check if the current "node" was a directory or a file with a call to `IsDir()`, and in the latter case call `Size()` from package `os` and add the returned size to the sum. However, when we tested this approach, `rocksdbSize` caused etcd to crash several times, as ongoing RocksDB compactions constantly add and remove SST files in the database directory[11]. Another idea to make this work would be to close the RocksDB instance, compute the directory's size and then reopen it. However, since accuracy is not that crucial for the usecase of `rocksdbSize` we decided to avoid a solution that would incur such a big overhead.

This forced us to turn to a second approach, involving direct calls to RocksDB methods that would give us information regarding the current size of the database. Even though

[10] `MANIFEST` maintains a list of SST files at each level and their corresponding key ranges, among other metadata. It is used to identify the SST files that may contain a given key. It functions as a log to which the latest changes on SST files are appended. `CURRENT` is a special file that identifies the latest manifest log file. `OPTIONS` stores the options used in the database. `LOCK` is used to ensure that only one process at a time can open RocksDB on a single directory. `LOG` is where statistics are dumped and changes in options are recorded. Finally, `IDENTITY` contains a serial number, unique to the database.

[11] `Walk()`'s internal implementation makes a list of all the names in the directory and calls Go's `Lstat()` function from the `os` package for each of them, which returns a `FileInfo`. A "disappearing" file will cause `Lstat()` to return `nil`. Also, a file created after the creation of the name list will not be taken into account.

this approach seems more natural, we encountered a problem: RocksDB does not provide a straightforward and completely accurate way to return its total size to the user. Therefore, we approximate the on-disk size of the database by adding the size of the SST files to the size of the memtables, which are returned by the gorocksdb function calls `db.GetProperty("rocksdb.total-sst-files-size")` and `db.GetProperty("rocksdb.cur-size-all-mem-tables")` respectively. The LSM-tree's structure is such that the size of the memtable is approximately the same as the size of the WAL. This is because the WAL is automatically purged whenever memtables are flushed to disk.

The rest of the files in the database directory can be safely ignored as the sum of their sizes is in the order of a few tens of kilobytes and would not change the result significantly. The majority of those files retain the same size throughout the lifetime of a RocksDB instance, while the ones that grow (i.e., `LOG` and `MANIFEST`) do so at a very slow rate[12].

`isReadOnly(mode os.FileMode) (bool, error)`: this function receives a `FileMode` and extracts permission information for the database. It returns `true` for read-only and `false` for read-write. An error is returned if the `FileMode` does not provide read permission. `isReadOnly` is called by `Open()` to determine whether the database should be opened with a call to gorocksdb function `OpenDb()` or `OpenDbForReadOnly()`.

`RocksDBWriteBatch interface`: this is an interface with the following set of functions: `Clear()`, `Put(key, value []byte)`, `Delete(key []byte)`, `NewIterator() *gorocksdb.WriteBatchIterator` and `Destroy()`. Both `WriteBatch` and `WriteBatchWithIndex` implement this interface. As we have mentioned previously, even though in the final, optimized version of our implementation we simulate read-write transactions using the `WriteBatch` concept, there exists a corner case where `WriteBatchWithIndex` is still necessary. This corner case is explained in detail in section 4.2. This means that before applying an operation on a transaction we need to know if it is simulated by an underlying `WriteBatch` or a `WriteBatchWithIndex`.

The `RocksDBWriteBatch interface` was introduced in order to simplify our code. Before its existence, we had two different fields in `Tx struct` for `WriteBatch` and

[12]They add up to a few hundreds of kilobytes when the database size is ~1GB.

`WriteBatchWithIndex`. Every time an operation needed to be applied on the transaction's underlying concept, we used an `if` clause to decide which of the two fields it should be applied on. Now, with the addition of the `RocksDBWriteBatch` interface we can have one common field of type `RocksDBWriteBatch` for both and there is no need for all those `if` clauses, which makes the code more readable.

However, there are still some cases where we need to distinguish between a `WriteBatch` and a `WriteBatchWithIndex`. For example, when we need to use functions that are unique to `WriteBatchWithIndex`, such as those that enable combining the contents of of its buffer with those of the database (i.e., `GetBytesFromBatchAndDB()`, `NewIteratorWithBase()`). In those cases we use the `index` boolean field of the `Tx struct` and a *type assertion*. The type assertion converts the `RocksDBWriteBatch interface` type to one of the types that implement it. It uses a dot and the required type in parentheses and looks like this:

```
wb, ok := tx.wb.(*gorocksdb.WriteBatchWithIndex)
```

This statement asserts that the interface value `tx.wb` holds the concrete type `*gorocksdb.WriteBatchWithIndex` and assigns the underlying `*gorocksdb.WriteBatchWithIndex` value to the variable `wb`. If `tx.wb` does not hold a `*gorocksdb.WriteBatchWithIndex`, the boolean value `ok` will be `false`, reporting the failure of the assertion [109].

`tarit(source, target string) error`: this function produces a tar archive from the directory at `source` and saves it at the path defined by `target`. The source code for this function was found on Svetlin Ralchev's blog [110]. `tarit` uses functions and structs from Go's `archive/tar` package and `Walk()` from `filepath` package. `Walk()` traverses the directory tree starting at `source` and for every "node" a `tar.Writer` writes to the tar archive a header that encodes metadata information, followed by the "node's" contents if it is a file. `tarit` returns an error if any of the functions it calls fails.

`untar(tarball, target string) error`: this function produces a directory at the path determined by `target` by *untarring* the tar archive located at `tarball`. The source code for this function was found on Svetlin Ralchev's blog [110]. It uses functions and structs from Go's `archive/tar` package. A `tar.Reader` is used to read all headers and use the metadata encoded in them to recreate the files and directories.

`untar` returns an error if any of the functions it calls fails.

`IsTar(path string) (bool, error)`: this function receives a `path` and returns `true` if it points to a tar archive and false if it does not. `Open()` uses it to confirm that the file its `path` argument points to is indeed a tar archive before proceeding to untar it. `IsTar` opens the file at `path` and stores its 262 first bytes in a buffer which it then feeds to the auxiliary function `isTar()`. An error is returned if one of the called functions fails.

`isTar(buf []byte) bool`: this function receives the first 262 bytes of a file in the form of a byte slice and returns `true` if the file is a tar archive. It determines this by inspecting the file's *magic numbers* signature. Magic numbers are specific constants in specific positions among the first bytes of a file that provide a way to distinguish between file formats. For example, a tar archive is expected to have the hexadecimal values `75`, `73`, `74`, `61` and `72` at its 258th, 259th, 260th, 261st and 262nd bytes respectively.

Next, we provide a brief description of all the error types defined in the `bolt` package. Some of them already existed in the original implementation of BoltDB, while others where added by us to cover extra failure cases. Each of these errors belongs to Go's `error` type and encloses a descriptive message, indicating the cause of failure. They are declared as can be seen below:

```
Err = errors.New("error message")
```

`ErrDatabaseReadOnly`: this error is returned when a read-write transaction is started on a read-only database.

`ErrTxClosed`: this error is returned when attempting to commit or roll back a transaction that has already been committed or rolled back.

`ErrTxNotWritable`: this error is returned when attempting to perform a write operation on a read-only transaction.

`ErrBucketExists`: this error is returned when attempting to create a bucket that already exists.

`ErrBucketNameRequired`: this error is returned when attempting to create a bucket with a blank name.

ErrKeyRequired: this error is returned when attempting to insert a zero-length key.

ErrKeyTooLarge: this error is returned when attempting to insert a key that is larger than the maximum recommended key size in RocksDB.

ErrValueTooLarge: this error is returned when attempting to insert a value that is larger than the maximum recommended value size in RocksDB.

ErrRuntimeCaller: this error is returned when `runtime.Caller` or `runtime.FuncForPC`, called by `Begin()` fail to return the caller function's name.

ErrInadequatePermissions: this error is returned when `Open()` is called with permissions that do not provide read access.

ErrFileTypeNotSupported: this error is returned when `Open()` is called with a path that points to a file which is not a tar archive, or to a tar archive that does not contain a `Checkpoint`.

ErrNoCheckpoint: this error is returned when `WriteTo()` is called on a transaction that has no `Checkpoint` associated with it.

4.2 Modifications in etcd Code

Even though we tried to contain our changes in the `bolt` package and avoid alterations of core etcd code, there were times when they were inevitable. We provide a detailed description of those cases below.

Defragmentation

As we have explained in subsection 3.2.7, now that BoltDB has been replaced by RocksDB, defragmentation is no longer necessary. For that reason we have commented out the contents of the function `Defrag()` in the `backend` package and removed the auxiliary functions `defrag()` and `defragdb()`. Any defragmentation command issued by the client will now result in a dummy function call. `TestBackendDefrag()` test function was also removed as it obviously is not needed anymore. In a future improved version of our implementation, `etcdctl` will return an appropriate error when a defragmentation is attempted while etcd has been configured to use RocksDB as its

storage engine, and will apply the defragmentation as requested when etcd has been configured to use BoltDB.

Removal of the database directory

In a lot of places throughout etcd code, mainly in test functions, the backend database needs to be removed from the file system. Go's `Remove()` function from the `os` package was used for this purpose. `Remove()` though is only able to remove single files and cannot recursively remove whole directories [97]. This approach was appropriate when etcd used BoltDB as its storage engine, as the database was a single file. However, in our implementation, where the backend is a RocksDB database whose on-disk representation is a directory, this is not enough. Therefore, we had to replace all calls to `Remove()` with calls to `RemoveAll()` from the same package.

Application of the snapshot

The function `applySnapshot()` from the `etcdserver` package is responsible for replacing an etcd node's backend with a given backend snapshot. It does this by starting a new backend based on the given snapshot, then closing the old backend and finally assigning the new backend to the etcd node. The obstacle we ran into, once more stems from the difference of the database format in BoltDB and RocksDB. Go's `Rename()` function from package `os` does not work if the target name points to an existing directory[13] [97]. `applySnapshot()` uses `Rename()` to rename the untarred snapshot directory to the old backend's name, replacing the latter. We overcame this by first removing the old backend's directory and then renaming the snapshot's directory.

What is more, closing the old backend might block until all the transactions on it are finished. To avoid waiting, in the original implementation of `applySnapshot()` a goroutine is invoked to call the `Close()` function of the `backend` package. This caused our implementation to fail some tests, as the `LOCK` file of RocksDB prevented us from opening a second instance of RocksDB at the same path. Our workaround for this was to make sure the old backend has been closed before attempting to open the new one, which was done by simply placing the call to `Close()` in the initial goroutine, before

[13] `Remove()` works as expected if the the target name points to an existing file or if there is no existing file or directory at the location it points to.

the command that opens the new backend. This change is slightly detrimental to the performance of snapshot application in etcd.

Getting the snapshot status

When the command `etcdctl snapshot status backup.db` is issued by the client, a backend is opened based on the given snapshot in order to get the required statistics. The call to `newBackend()` function of the `backend` package involves a call to `Open()` function of the `bolt` package, which as we have seen untars the snapshot tar archive. The problem is that the `snapshotRestoreCommandFunc()` function which is called when the client issues the command `etcdctl snapshot restore backup.db` expects to see a file and not a directory[14]. When a `etcdctl snapshot status backup.db` command comes before a `etcdctl snapshot restore backup.db` command, that will cause an error. To eliminate this case we had to alter the `dbStatus()` auxiliary function called by `snapshotStatusCommandFunc()` to tar the snapshot again after untarring it and getting the desired metrics.

Switching between WriteBatch and WriteBatchWithIndex

As we have mentioned in section 4.1, sometimes we need to change the underlying representation of a transaction from a `WriteBatch` to a `WriteBatchWithIndex`. This change is performed by the `Switch()` function. Here we explain the exact reason why this switch is necessary and describe the modifications we made to etcd code outside the `bolt` package to make it happen.

An intermediate version of our implementation used `WriteBatchWithIndex` to simulate BoltDB transactions. Later, we discovered that using `WriteBatch` instead of `WriteBatchWithIndex` would offer a significant performance improvement, as it will be further analysed in subsection 4.3.5. For a description of the properties of those two RocksDB structures see subsection 2.4.2.

Actually, replacing `WriteBatchWithIndex` with `WriteBatch` was made possible because of a recent change in the backend of etcd. Before this change, `WriteBatchWithIndex` was indispensable for our implementation. The focus of this change in

[14] `snapshotRestoreCommandFunc()` copies the snapshot file to the appropriate directory before starting a backend from it.

etcd was decoupling reads from writes so that they do not have to contend for the same lock (i.e., the lock of the `batchTx struct`). Reads are issued on a `readTx` and writes are issued on a `batchTx`, while before the change both reads and writes were issued on a `batchTx`. `batchTx` and `readTx` are structs defined in the `backend` package and are basically wrappers for a `bolt Tx struct`. In the case of `batchTx` this is a read-write transaction, while in the case of `readTx` it is a read-only transaction. In etcd only one of each is active at a given time[15]. Every access to the storage engine passes through a `batchTx` or a `readTx`.

This decoupling practically means that etcd read requests will be mapped to the `readTx` and etcd write requests to the `batchTx`. Reads must be able to *see* the updates contained in the `batchTx` even before they get committed. For this reason, there exists a buffer associated with the `readTx` that contains all of the `batchTx`'s updates (i.e., key-value pairs that are pending commit). Reads first check this buffer and then the storage engine. The buffer is basically a shared in-memory map in front of the `batchTx` and `readTx`.

A problem arises from this approach, as an iteration will inadvertently return duplicates if some keys in its range happen to have been recently updated and reside both in the buffer and the storage engine. This is only a problem for buckets other than the `key` bucket. Because of MVCC, keys are never updated in-place in the `key` bucket[16]. This makes it impossible for duplicates of the keys existing in the storage engine to exist in the `readTx` buffer. What is more, uncommitted deletes will be ignored: the deleted keys are found in the storage engine and returned as if they had never been deleted.

To avoid returning duplicates and deleted keys in the cases that an iteration is directed to buckets other than the `key` bucket, the read request must circumvent the buffer and be directly applied on the `batchTx`, which has the inherent ability to correctly combine its uncommitted updates with the contents of the storage engine and produce the appropriate result. A BoltDB read-write transaction has this ability and so does a RocksDB `WriteBatchWithIndex`. A `WriteBatch` however, does not.

[15]The first `batchTx` begins when etcd starts and the backend is created. The batch interval, which is the maximum time before committing a `batchTx`, is equal to $100ms$. The batch limit, which is the maximum number of updates before committing the `batchTx`, is set to 10000. A dedicated goroutine commits the `batchTx` when one of these limits is exceeded and begins a new one.

[16]For a description of the data model of etcd, see subsection 2.2.1.

Knowing that etcd already implements *read-your-own-writes* with a shared buffer and that no reads will be applied to the `batchTx` in the general case permits us to implement it with an underlying `WriteBatch`, but we must account for the aforementioned case.

A solution is to generally use `WriteBatch` but copy it into a `WriteBatchWithIndex` whenever a read request is applied on the `batchTx` instead of the `readTx`. We do this by calling the `Switch()` function before applying a read request to a `batchTx`. `Switch()` iterates over the `WriteBatch`'s records and replays the Put and Delete actions upon a new `WriteBatchWithIndex`, with which it replaces the `WriteBatch`.

To do this we have a added a call to `Switch()` in functions `(t *batchTx) UnsafeRange(bucketName, key, endKey []byte, limit int64) ([][]byte, [][]byte)` and `(t *batchTx) UnsafeForEach(bucketName []byte, visitor func(k, v []byte) error) error` of the `backend` package. `UnsafeForEach()` is called on a `readTx` only in two cases:

- to read the activated alarms from the `alarm` bucket and store them in an in-memory map
- to delete the stored members of the old cluster from the `members` bucket when starting an etcd node from a snapshot

`UnsafeRange()` is called on a `readTx` in the following cases:

- to retrieve lease information from the `lease` bucket when starting an etcd node
- to read the consistent index from the `meta` bucket when the client issues a command to migrate the keys of the v2 store to the v3 store, or when a snapshot is applied
- to retrieve information from the `meta` bucket when an MVCC history compaction is applied
- to retrieve information from the `auth` bucket upon client request

None of these calls happens very often compared to the average lifetime of a transaction, which is shorter than $100ms$. Thus, we decided that there is no reason to switch

back to a `WriteBatch` after having switched to a `WriteBatchWithIndex`. When a new transaction is created, it always starts with a `WriteBatch`.

Removal of the storage quota

For the reasons explained in subsection 3.1.5 we have decided to lift the restriction placed by etcd on the size of its storage backend. The default quota size and the maximum configurable quota size are stored in the constants `DefaultQuotaBytes` and `MaxQuotaBytes` in the `etcdserver` package. Thus, removing the quota is as trivial as setting those constants equal to `math.MaxInt64`. We can also consider setting the constants to reasonably large values, such as 30GB for the `DefaultQuotaBytes` and 50GB for the `MaxQuotaBytes`. The user is still able to impose a restriction on the backend size if they wish to do so, through the start-up time `--quota-backend-bytes` flag.

Scripts

The `build` script of etcd sets the `CGO_ENABLED` environment variable to 0 before compiling etcd. We set this variable to 1 in order to enable the use of cgo. As a side effect, this renders the compilation no longer static. We also set the environment variables `CGO_CFLAGS` and `CGO_LDFLAGS` appropriately in order to be able to use RocksDB as a *shared* library as can be seen below. These environment variables contain compiler and linker *flags*; the first one allows the C compiler to "see" the RocksDB header files (i.e., the `rocksdb/c.h` header, which is imported in the cgo preambles of various go-rocksdb files) and the second one allows our pre-compiled RocksDB library and its dependencies to be used by the linker.

```
export CGO_CFLAGS="-I ${GOPATH}/src/${ORG_PATH}/rocksdb/include"
export CGO_LDFLAGS="-L ${GOPATH}/src/${ORG_PATH}/rocksdb -lrocksdb
    -lstdc++ -lm -lz -lbz2 -lsnappy -llz4"
```

Those two changes are also applied in the `build` script of the functional tester, located at `etcd/tools/functional-tester/build`, with the difference that `CGO_ENABLED` only needs to be set to 1 for the `etcd-tester`. What is more, we set the

variable `ORG_PATH` to point to `"github.com/boolean5"`. Finally, in the function `etcd_build()` we add the following line to make the script build the `benchmark` tool along with `etcd` and `etcdctl`:

```
CGO_ENABLED=1 go build $GO_BUILD_FLAGS -installsuffix cgo -ldflags
   "$GO_LDFLAGS" -o ${out}/benchmark ${REPO_PATH}/cmd/tools/benchmark
        || return
```

4.3 Optimizations

In this section we review the most impactful of the optimizations we applied over our base implementation and provide the rationale behind each of them. In chapter 5 we will present their beneficial effect on the performance of etcd.

4.3.1 Base Implementation

Our first approach to the integration of RocksDB into etcd resulted from applying the design decisions described in section 3.2. Among other things, BoltDB transactions were naively mapped to RocksDB transactions, a design which was semantically correct but lead to the development of overly complex code and imposed an unnecessary overhead. Furthermore, just like it happened with BoltDB before the change, every put/get/delete request to etcd resulted in two calls to the `bolt` package, one to fetch the appropriate bucket and a second one to get/put/delete the key-value pair.

4.3.2 Bucket Access

As the reader can confirm by taking a look at chapter 5, the performance of the base implementation was rather unsatisfactory. Specifically, although the write performance should theoretically have improved, it was a little worse than before. At that point we used Go's *profiler* to identify potential bottlenecks. It allowed us to get a 30-second CPU profile, while loading etcd with a million key-value pairs, using the built-in `benchmark` tool. Etcd already supports profiling[17], so all we had to do was

[17] If it did not we would have to import `runtime/pprof` or `net/http/pprof` if the application runs an http server and add a few lines of code.

run it with the appropriate flag: `./etcd --enable-pprof`. Then, to interpret the profile we used the following command:

```
go tool pprof http://localhost:2379/debug/pprof/profile
```

When in interactive mode, typing `top 10` showed the 10 most CPU-consuming samples. The output of this command can be inspected below.

```
Showing top 10 nodes out of 225 (cum >= 680ms)
      flat  flat%   sum%        cum   cum%
   10780ms 17.61% 17.61%    10980ms 17.94%  runtime.cgocall
    8600ms 14.05% 31.66%     9010ms 14.72%  syscall.Syscall
    1800ms  2.94% 34.60%     4950ms  8.09%  runtime.mallocgc
    1510ms  2.47% 37.07%     1510ms  2.47%  runtime.readvarint
    1330ms  2.17% 39.24%     2890ms  4.72%  runtime.selectgoImpl
    1330ms  2.17% 41.41%     2840ms  4.64%  runtime.step
    1170ms  1.91% 43.33%     4350ms  7.11%  runtime.pcvalue
     850ms  1.39% 44.71%      850ms  1.39%  runtime.heapBitsSetType
     780ms  1.27% 45.99%     1270ms  2.07%  runtime.deferreturn
     680ms  1.11% 47.10%      680ms  1.11%  runtime.adjustpointers
```

We also used the `svg` command, which constructs a graph of the profile data in SVG[18] format. Each box in the graph corresponds to a single function, and the boxes are sized according to the time consumed in the function. An edge from box x to box y indicates that x calls y; the number along the edge is the number of seconds spent in a function [111]. A glance was enough to reveal that the most time-consuming operations in etcd were the *cgo calls*. For a discussion on the overhead imposed by cgo, see subsection 2.5.2. To get a more comprehensible graph with less noise that would allow us to trace the source of the problem we ran the experiment again, this time using the command `go tool pprof --nodefraction=0.1 http://localhost:2379/debug/pprof/profile`, that ignores nodes that don't account for at least 10% of the total duration of the profiling, and the command `svg cgocall` that narrows the graph down to the functions that result in cgo calls. Part of the output graph appears in Figure 4.2.

[18]Scalable Vector Graphics

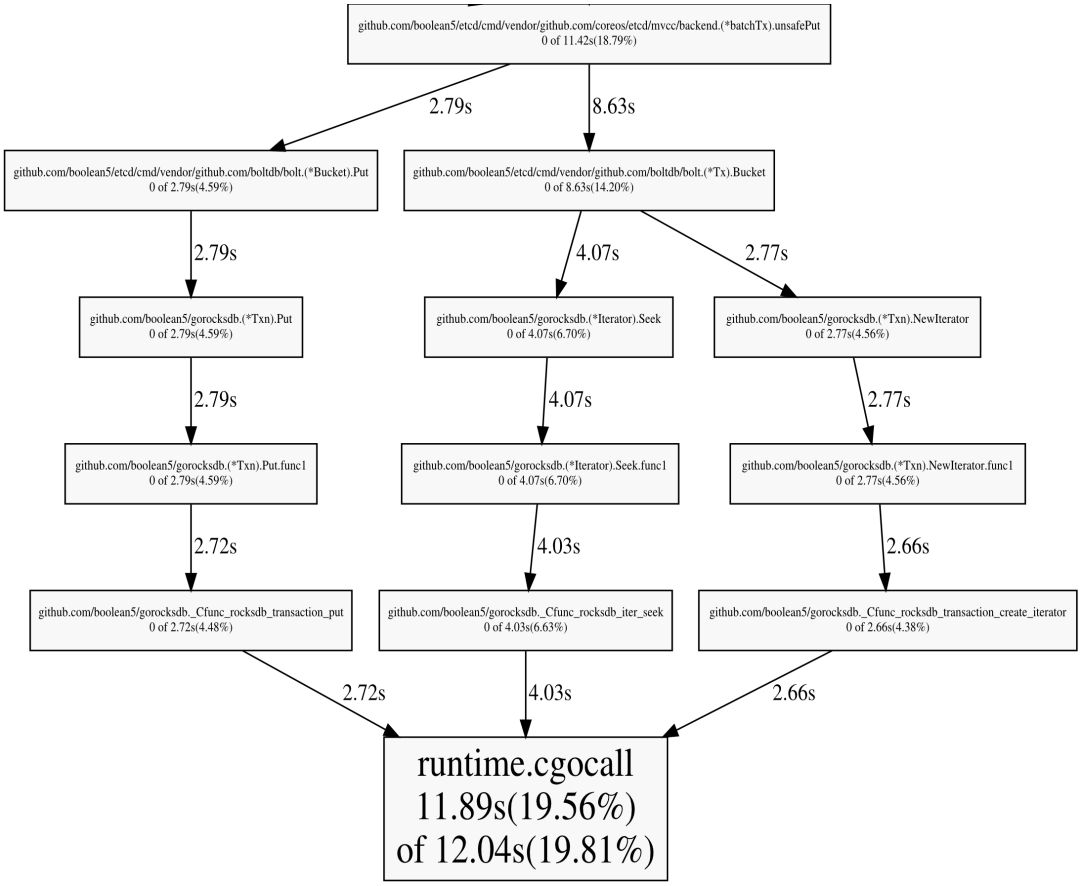


Figure 4.2: *CPU profile of the base implementation*

From this graph we can deduce that the most frequent caller of cgo is the `Bucket()` function of the `bolt` package. When a put request is issued to the backend of etcd, it results in two calls to the `bolt` package, one to fetch the appropriate bucket and a second one to put the key-value pair in that bucket. What is more, our base implementation of the `Bucket()` function involved two calls to the storage engine (i.e., two cgo calls): one to create a RocksDB iterator and a second one to seek the specific bucket, using this iterator.

An obvious improvement over this naive design that copied the implementation of the original `Bucket()` function of BoltDB, would be to replace those two cgo calls with one cgo call to the `Get()` function of RocksDB. However, we can achieve an even better result by eliminating both calls to cgo and instead maintaining an *in-memory* set of existing buckets, implemented as a Go map. This way the `Bucket()` function will not need to query the storage engine. The map of buckets is associated with the `DB struct`, it is created in function `Open()` of the `bolt` package and filled with any already existing

buckets stored in the database. Finally, it is updated in `CreateBucket()`. The map's usefulness in the context of the `Bucket()` function, lies in confirming that a certain bucket exists and not in retrieving a value that will be returned to the caller. Those changes can be seen in the corresponding code segments of section 4.1. The map of buckets would normally need to be lock protected, but since within etcd the buckets are only created when a node is started and before any other operation can be applied, this is not necessary. It is worth noting here that buckets are an internal concept, employed for the separation of keyspaces; the etcd client cannot access or modify them.

We can confirm that after this change `runtime.cgocall` no longer takes up such a significant part of the run time, by running the profiler again and issuing a `top 10` command, whose output can be seen below. We have now managed to reduce the cgo calls per put request to the backend from three to just one[19]. In chapter 5 we will see how this change in the way buckets are accessed led to a significant performance improvement.

```
Showing top 10 nodes out of 220 (cum >= 940ms)
      flat  flat%   sum%        cum   cum%
    7570ms 12.80% 12.80%     8000ms 13.52%  syscall.Syscall
    5060ms  8.55% 21.35%     5110ms  8.64%  runtime.cgocall
    2110ms  3.57% 24.92%     6170ms 10.43%  runtime.mallocgc
    1910ms  3.23% 28.14%     5900ms  9.97%  runtime.pcvalue
    1870ms  3.16% 31.30%     3570ms  6.03%  runtime.step
    1700ms  2.87% 34.18%     1700ms  2.87%  runtime.readvarint
    1430ms  2.42% 36.60%     3430ms  5.80%  runtime.selectgoImpl
    1130ms  1.91% 38.51%     1130ms  1.91%  runtime.heapBitsSetType
    1050ms  1.77% 40.28%     9060ms 15.31%  runtime.gentraceback
     940ms  1.59% 41.87%      940ms  1.59%  runtime.adjustpointers
```

[19]At the level of a put request to etcd, we have reduced the number of cgo calls from 6 to 2, as each of those requests is translated into two put requests to the backend: one to store the key-value pair and a second one to update the `consistentIndex`.

4.3.3 Optimistic Transactions

As it has been mentioned before, pessimistic transactions are better for workloads with heavy concurrency, while optimistic transactions are more suitable for workloads that do not expect high contention between multiple transactions.

In the context of our current implementation, only one read-write transaction is active at a time. This fact is enforced both by the `DB struct`'s lock in the `bolt` package and by etcd's internal design, which channels all changes to the storage engine through a single `batchTx` object that is reused after being committed. Therefore, since write-conflicts are practically non-existent, we replaced pessimistic transactions with optimistic ones and saw a significant improvement in write performance due to avoiding the locking overhead of the former.

For a thorough analysis of the semantic and performance differences between pessimistic and optimistic concurrency control see subsection 2.1.7 and for a comparison of the two types of transactions in RocksDB see subsection 2.4.2.

Putting this optimization into practice was rather trivial: it was enough to substitute the gorocksdb `TxnDB` associated with the `DB struct` with an `OptimisticTxnDB` and replace some gorocksdb functions with their equivalent version for optimistic transactions.

4.3.4 WriteBatchWithIndex & Snapshot

Taking the conclusions of subsection 4.3.3 a little further, we can deduce that if the design of etcd and that of the `bolt` package ensure that there are no concurrent transactions, there is no need for concurrency control on the side of RocksDB. In other words, providing atomicity and durability is enough. This means that we can simulate a read-write BoltDB transaction using a RocksDB `WriteBatchWithIndex`, which, unlike `WriteBatch`, allows us to perform reads that have access to the uncommitted writes of the batch. Moreover, we can simulate a read-only BoltDB transaction using a RocksDB `Snapshot` which from a semantic point of view is essentially the same thing: a consistent, read-only view of the database. For a description of the concepts of `WriteBatchWithIndex` and `Snapshot` in the context of RocksDB see subsection 2.4.2.

A `WriteBatchWithIndex` is much more lightweight than a RocksDB transaction, since transactions are implemented with an underlying `WriteBatchWithIndex` and a concurrency control mechanism. In fact, a simple benchmark revealed that writing 1 million key-value pairs to the database with a `WriteBatchWithIndex` is 2.5 times as fast as with an optimistic transaction.

To implement this optimization we had to:

- Replace the underlying gorocksdb `OptimisticTxnDB` attached to the `DB struct` with a plain gorocksdb `DB`
- Replace the underlying gorocksdb `Txn` attached to the `Tx struct` with a `WriteBatchWithIndex`
- Make `beginRWTx()` create a `WriteBatchWithIndex` instead of beginning an optimistic transaction
- Make `beginROTx()` create a new `Snapshot` instead of beginning an optimistic transaction
- In `Rollback()`, `Clear()` the `WriteBatchWithIndex` or `Release()` the `Snapshot` instead of rolling back a RocksDB transaction
- In `Commit()`, use the gorocksdb `WriteWithIndex()` function to write the `WriteBatchWithIndex` to the database, instead of committing a RocksDB transaction
- In `Put()`, put the key-value pair in the `WriteBatchWithIndex` instead of an optimistic transaction
- In `Get()`, use the function `GetBytesFromBatchAndDB()` on the `WriteBatchWithIndex`
- In `Delete()`, apply the delete operation on the `WriteBatchWithIndex` instead of an optimistic transaction
- In `Cursor()`, use `NewIteratorWithBase()`
- Remove all code that handles `OptimisticTxnOptions`

After the application of this optimization the code also became less complex. Before, there existed several `if` clauses, to distinguish between an `OptimisticTxnDB` and a plain `DB`, which was used when the database was opened in read-only mode. Now, it was possible to eliminate those `if` clauses, as we use a plain `DB` both when working in read-only mode and read-write mode.

`WriteBatch` and `WriteBatchWithIndex` are not thread safe, in the sense that multiple operations cannot be issued to the same batch concurrently, and require external synchronization. Because this is also the case for the `batchTx struct` of the `backend` package of etcd, adequate locking protection is already implemented[20].

Initially, every read-write transaction had an attached `Snapshot`, created in `beginRWTx()`, to ensure repeatable reads. After careful inspection, we realized that this was not necessary. Since only one read-write transaction is active at any given time, it will already possess a consistent view of the database. On the other hand, read-only transactions still need a `Snapshot`, as the database may be changed by a read-write transaction during their lifetime. The removal of `Snapshots` from read-write transactions gave a small extra performance benefit.

4.3.5 WriteBatch

A simple benchmark revealed that writing 1 million key-value pairs to the database with a `WriteBatch` is 3.5 times as fast as with an optimistic transaction and 1.4 times faster than with a `WriteBatchWithIndex`. `WriteBatchWithIndex` has a little more overhead compared to `WriteBatch`, because it maintains an internal buffer in the form of a searchable index.

As we have mentioned before, a recent change in the `backend` package of etcd, enabled `readTx` to access the uncommitted entries of `batchTx`. This allows us to replace `WriteBatchWithIndex` with `WriteBatch` in our implementation. For a description of the concept of the `WriteBatch` see subsection 2.4.2. Also, the semantic correctness of this change is thoroughly explained in section 4.2, as well as the reason why `WriteBatchWithIndex` is still needed in some cases.

To implement this optimization we had to replace some gorocksdb functions related

[20] In etcd, functions whose name has “Unsafe” as a prefix must be called while holding the lock of the associated `batchTx`.

to `WriteBatchWithIndex` with their equivalent version for `WriteBatch`. We also had to add some `if` clauses to distinguish between the case of using a `WriteBatch` and the case of using a `WriteBatchWithIndex`, and write a function to switch from one to the other when necessary. The resulting code has been described and explained in section 4.1.

4.3.6 Tuning RocksDB

Tuning RocksDB is a complex task involving more than 120 configurable parameters with different degrees of interdependencies. The default values do not match up to the actual potential of the system, thereby delivering low performance. The following segment from the documentation of RocksDB [64] gives a good idea of the complexity of the tuning process:

> "Unfortunately, configuring RocksDB optimally is not trivial. Even we as RocksDB developers don't fully understand the effect of each configuration change. If you want to fully optimize RocksDB for your workload, we recommend experiments and benchmarking, while keeping an eye on the three amplification factors."

Besides depending on the expected workload, tuning is hardware dependent[21], so it should be readjusted for different machines. It can be viewed as a trade-off between the three amplification factors: write amplification, read amplification and space amplification[22]. In our case, we try to favour write amplification, as our ultimate goal is to improve the write performance of etcd and we are willing to sacrifice read performance and space consumption within reasonable bounds. This can be translated into following a strategy of less aggressive compactions[23] and generally tuning the options in the direction of faster writes.

We will now present a few documented existing approaches to tuning RocksDB. Developers at Samsung's memory solutions laboratory used a greedy algorithmic approach

[21]The storage technology (i.e., SSD vs HDD), available cores and memory capacity should be taken into account.

[22]For a definition of read, write and space amplification see subsection 2.4.1.

[23]Performing compactions often, increases write amplification and reduces space and read amplification.

to identify optimal values for RocksDB configuration parameters. They identified ~25 parameters that significantly affect performance, as well as ~5 possible values for each of them. Then, they performed a benchmark with each of those different values for an isolated parameter, while keeping the rest fixed to reasonable values, and selected the value that gave the highest operations/second rate. This process was repeated for all of the selected parameters. It is worth noting that as this process is not exhaustive, it only finds local optimal values and might miss the global optimal ones. However, the developers at Samsung report a 77% increase in performance over the baseline configuration [112].

Furthermore, the developers of Redis-on-Flash[24] have ran a series of experiments to tune RocksDB and have published a paper where they report in detail both the parameters that significantly improved performance, as well as configurations that seemed promising but had a negative effect. They managed to improve overall performance by more than 11 times over the baseline performance [113]

M. Callaghan, one of the developers of RocksDB, has created a script[25] that receives some of the configuration options of RocksDB as parameters and runs a series of benchmarks, to facilitate tuning. Finally, the Apache Flink framework[26] has established 4 predefined option configurations[27], empirically determined to be beneficial for performance under different hardware and workload settings, based on experiments by the Flink community and guidelines from RocksDB documentation.

Our approach was very similar to that of Samsung's memory solutions laboratory. We selected a subset of parameters worth experimenting with, based on tuning tips found in the documentation of RocksDB [64], in the RocksDB developers forum[28] and in the reports of the approaches mentioned above. Then, starting from the parameters that we had identified as the most influential based on previous tests, we ran some benchmarks for several values of each one, stabilized it choosing the value that maximized write throughput and moved on to the next.

[24] Redis-on-Flash is an implementation of the Redis in-memory key-value store that uses SSDs as an extension to RAM in order to increase the capacity per-node. Hot data are retained in RAM, while RocksDB is utilized for storing cold data on SSD drives [113].

[25] `https://github.com/mdcallag/mytools/blob/master/bench/rocksdb.db_bench/all.sh`

[26] `https://flink.apache.org/`

[27] `https://ci.apache.org/projects/flink/flink-docs-master/api/java/org/apache/flink/contrib/streaming/state/PredefinedOptions.html`

[28] `https://www.facebook.com/groups/rocksdb.dev/`

Below we provide a theoretical description of each parameter we have included in our configuration and an explanation of the related trade-offs and side effects if any. The parameters we experimented with were a lot more than those mentioned below, but they did not appear to have a significant performance impact for our workload and hardware configuration.

Bloom filters: In order to reduce read amplification we enable Bloom filters by calling the function `blockOpts.SetFilterPolicy(gorocksdb.NewBloomFilter(10))`. This filter will keep 10 bits of data in memory per key, reducing the number of unnecessary disk reads needed for get operations by a factor of approximately 100 and yielding a ~1% false positive rate [64]. Increasing the bits per key will lead to a larger reduction at the cost of more memory usage. However, Bloom filters are only useful for point lookups and not for range scans. For the definition of Bloom filters see subsection 2.4.1.

Parallelism options: In the LSM-tree architecture, there are two background processes: flushing memtables and compaction. We can enable concurrent execution for both of them in order to take advantage of storage technology concurrency. The HIGH priority thread pool contains threads used for flushing, since flushing is in the critical code path, while the LOW priority pool contains threads used for compaction. To set the number of threads accessible by RocksDB we call `opts.IncreaseParallelism(total_threads)`. To adjust the number of threads in each pool we call: `env.SetBackgroundThreads(total_threads - 1)` and `env.SetHighPriorityBackgroundThreads(1)`. To compute the number of available threads in the system we use the Go function `runtime.NumCPU()` [64].

To benefit from more threads we call the following functions to change the maximum number of concurrent compactions and flushes: `opts.SetMaxBackgroundCompactions(total_threads - 1)` and `opts.SetMaxBackgroundFlushes(1)`. The default maximum number of concurrent compactions is 1, but to fully utilize our CPU and storage it is recommended to increase this to approximately the number of cores in the system. Two compactions operating at different levels or at different key ranges are independent and may be executed concurrently. More parallelism on the compaction threads shortens the compaction cycles and provides a higher write throughput [113]. As far as the maximum number of concurrent flush operations is

concerned, it is usually good enough to set it to 1[29] [64].

Moreover, we use `opts.SetMaxSubcompactions()` to set the maximum number of threads that will concurrently perform a compaction job by breaking it into multiple, smaller ones that are run simultaneously. With `opts.SetAllowConcurrentMemtableWrites(true)` we allow concurrent memtable updates. According to the documentation, when doing this it is strongly recommended to also call `opts.SetEnableWriteThreadAdaptiveYield(true)`.

Flushing options: There are numerous options that control the flushing behaviour. To determine the size of a single memtable in bytes we use the function `opts.SetWriteBufferSize()`. Once a memtable exceeds this size, it is marked as immutable and a new one is created. To set the maximum number of memtables, both active and immutable, we call `opts.SetMaxWriteBufferNumber()`. If the active memtable fills up and the total number of memtables is larger than this, we stall further writes. This may happen if the flush process is slower than the write rate. In addition, with `opts.SetMinWriteBufferNumberToMerge()` we can set the minimum number of memtables to be merged before flushing to storage. If for example this option is set to 2, immutable memtables are only flushed when there are two of them. When multiple memtables are merged together, less data may be written to storage since different updates of the same key are merged. On the other hand, every get operation must traverse all immutable memtables linearly to check if the requested key is there [64].

An issue we encountered while experimenting with the flushing options was that when the total capacity of the memtables was large, RocksDB was quite slow to restart after being closed. This happens because, in order to recover its previous state, RocksDB has to read its WAL and reload everything that was in memory before being closed. This trade-off between write performance and recovery time should be taken into account.

Compaction options: According to the documentation and the outcome of our experiments, Universal Compaction is more suitable than Level Compaction for write-heavy workloads. On the other hand, it increases read and space amplification. To select this style of compaction we call `opts.SetCompactionStyle(gorocksdb.UniversalCompactionStyle)` and `opts.OptimizeUniversalStyleCompaction()`

[29] Another reasonable configuration is to assign the 2/3 of the available cores to compactions and the remaining 1/3 to flushes.

for further optimization. More information on compaction styles is available in subsubsection 2.4.1.

Compression options: Each block is individually compressed before being written to persistent storage. The default compression method, Snappy is very fast. We tried disabling compression entirely by calling `opts.SetCompression(gorocksdb.NoCompression)`, but this did not yield a performance improvement.

Other options: There exists an option, accessed through the `DisableWAL()` function, that disables the use of the WAL in RocksDB, offering a significant performance benefit. Putting a key-value pair in RocksDB with the WAL disabled involves only an in-memory operation. Since etcd keeps a WAL, one might think that a second one at the level of the storage engine is redundant. In the event of a crash all data stored in the memtable of RocksDB would be lost. Then, theoretically, when the node is restarted, etcd could check the latest revision stored in the storage engine and replay its WAL from that revision onwards.

In practice however, disabling the WAL of RocksDB might render the cluster state inconsistent. This is because in etcd v3, the on-disk storage engine database serves as an incremental snapshot, as we have seen in subsection 2.2.2. In other words, if any updates *committed*[30] to the storage engine reside exclusively in memory, etcd will not be able to recover its state correctly after a crash. The WAL of etcd (i.e., the raft log) and the snapshot (i.e., the storage engine database) are complementary. A snapshot record is appended to the WAL every 100000 entries to the backend, and previous log entries are discarded. The snapshot record is a marshalled struct with an index and a term. Right before this happens a commit is also issued, that commits outstanding transactions to the underlying backend.

RocksDB also offers the option to make all write operations (i.e., calls to the `Put()` and `Delete()` functions) synchronous. We need not configure this option since all our updates are done through the `Write()` function which is synchronous by default.

Another frequently tuned option is the block size. When reading a key-value pair from an SST file, an entire block is loaded into memory. The default block size is 4KB. Increasing this value makes memory consumption decrease, since the blocks per SST

[30]The uncommitted updates of the current transaction are sure to be kept in the Raft log so it is safe for them to exist only in the in-memory component of the storage engine.

file become fewer and the in-memory indices that list the offsets for all blocks contain fewer entries [64].

Before running the benchmarks of chapter 5, we enabled parallelism in all versions of our implementation. However, only the final version is actually tuned. An example tuning configuration on a machine with an SSD, 4 cores and 4GB of RAM is available in the code segment below.

```
func createOptions() *gorocksdb.Options {
	opts := gorocksdb.NewDefaultOptions()
	opts.SetCreateIfMissing(true)

	// ******************** Tuning RocksDB ********************
	// Bloom Filters
	blockOpts := gorocksdb.NewDefaultBlockBasedTableOptions()
	blockOpts.SetFilterPolicy(gorocksdb.NewBloomFilter(10))
	opts.SetBlockBasedTableFactory(blockOpts)

	// Parallelism Options
	total_threads := runtime.NumCPU()
	opts.IncreaseParallelism(total_threads)
	env := gorocksdb.NewDefaultEnv()
	env.SetBackgroundThreads(total_threads - 1)
	env.SetHighPriorityBackgroundThreads(1)
	opts.SetEnv(env)

	opts.SetMaxBackgroundCompactions(total_threads - 1)
	opts.SetMaxBackgroundFlushes(1)
	opts.SetMaxSubcompactions(2)
	opts.SetAllowConcurrentMemtableWrites(true)
	opts.SetEnableWriteThreadAdaptiveYield(true)

	// Flushing Options
	opts.SetWriteBufferSize(512 * 1024 * 1024)
	opts.SetMaxWriteBufferNumber(3)
```

```
28
29          // Compaction Options
30          opts.SetCompactionStyle(gorocksdb.UniversalCompactionStyle)
31          opts.OptimizeUniversalStyleCompaction(2048 * 1024 * 1024)
32
33          return opts
34  }
```

Listing 4.9: *Example tuning configuration for RocksDB*

4.4 External Contributions

In numerous occasions during the course of our implementation we found ourselves in need of functionality that was not available in the RocksDB C API or gorocksdb, which are both relatively incomplete libraries. As a result, we worked on certain patches, some of which were merged in the original upstream projects. This section provides a complete list of our contributions. The full source code can be found in our repositories, forked from RocksDB and gorocksdb: `https://github.com/boolean5/rocksdb` and `https://github.com/boolean5/gorocksdb`. In this section we include only a few code segments to give the reader a general idea of the content of the patches. For a brief description of how the C API and the gorocksdb wrapper work, see subsection 3.1.4 and subsection 3.1.3.

Transactions

At the time of our implementation neither the C API of RocksDB nor gorocksdb supported Transactions or Optimistic Transactions. Therefore, we added to both of them more than 40 structs and functions to handle a transaction/optimistic transaction database (e.g., `rocksdb_transactiondb_open()`), the transaction/optimistic transaction options (e.g., `rocksdb_transaction_options_create()`, `rocksdb_transaction_options_set_set_snapshot()`), and the transactions or optimistic transactions themselves (e.g., `rocksdb_transaction_begin()`, `rocksdb_transaction_get()`, `rocksdb_transaction_commit()`). Some indicative code examples follow.

```
struct rocksdb_transaction_t {
  Transaction* rep;
};
```

Listing 4.10: *The `rocksdb_transaction_t struct` in the RocksDB C API*

```
char* rocksdb_transaction_get(rocksdb_transaction_t* txn,
                              const rocksdb_readoptions_t* options,
                              const char* key, size_t klen, size_t* vlen,
                              char** errptr) {
  char* result = nullptr;
  std::string tmp;
  Status s = txn->rep->Get(options->rep, Slice(key, klen), &tmp);
  if (s.ok()) {
    *vlen = tmp.size();
    result = CopyString(tmp);
  } else {
    *vlen = 0;
    if (!s.IsNotFound()) {
      SaveError(errptr, s);
    }
  }
  return result;
}
```

Listing 4.11: *The `rocksdb_transaction_get()` function in the RocksDB C API*

```
type Txn struct {
      c      *C.rocksdb_transaction_t
      opts   *WriteOptions
      txnOpts *TxnOptions
}
```

Listing 4.12: *The `Txn struct` in gorocksdb*

```
func (txn *Txn) Get(opts *ReadOptions, key []byte) (*Slice, error) {
	var (
		cErr   *C.char
		cValLen C.size_t
		cKey   = byteToChar(key)
	)
	cValue := C.rocksdb_transaction_get(txn.c, opts.c, cKey,
	    C.size_t(len(key)), &cValLen, &cErr)
	if cErr != nil {
		defer C.free(unsafe.Pointer(cErr))
		return nil, errors.New(C.GoString(cErr))
	}
	return NewSlice(cValue, cValLen), nil
}
```

Listing 4.13: *The transaction `Get()` function in gorocksdb*

WriteBatchWithIndex

Gorocksdb was also lacking support for `WriteBatchWithIndex`, leading us to develop a patch with the required functions for the creation and destruction of the batch, for writing, reading and deleting values in the batch, clearing it, iterating over its records, etc. We also had to implement a function to persist the batch to the database, equivalent to `Write()`, but for a `WriteBatchWithIndex`. Because function overloading is not supported in Go, this function was named `WriteWithIndex()`.

```
type WriteBatchWithIndex struct {
	c *C.rocksdb_writebatch_wi_t
}
```

Listing 4.14: *The `WriteBatchWithIndex struct` in gorocksdb*

```
func (wb *WriteBatchWithIndex) Put(key, value []byte) {
	cKey := byteToChar(key)
	cValue := byteToChar(value)
```

```
4       C.rocksdb_writebatch_wi_put(wb.c, cKey, C.size_t(len(key)),
            cValue, C.size_t(len(value)))
5   }
```

Listing 4.15: *The `WriteBatchWithIndex Put()` function in gorocksdb*

Checkpoint

Yet another thing that was missing both from the RocksDB C API and the gorocksdb wrapper was support for `Checkpoint`. We extended both of them with the structs and functions required to create and destroy a `Checkpoint` object and an actual checkpoint of the database. Some representative code examples follow.

```
1   void rocksdb_checkpoint_create(rocksdb_checkpoint_t* checkpoint,
2                                  const char* checkpoint_dir,
3                                  uint64_t log_size_for_flush, char**
                                       errptr) {
4    SaveError(errptr, checkpoint->rep->CreateCheckpoint(
5                      std::string(checkpoint_dir),
                           log_size_for_flush));
6   }
```

Listing 4.16: *The `rocksdb_checkpoint_create()` function in the RocksDB C API*

```
1   func (c *Checkpoint) Create(checkpointDir string, logSizeForFlush
        uint64) error {
2       var (
3             cErr *C.char
4             cDir = C.CString(checkpointDir)
5       )
6       defer C.free(unsafe.Pointer(cDir))
7       C.rocksdb_checkpoint_create(c.c, cDir,
            C.uint64_t(logSizeForFlush), &cErr)
8       if cErr != nil {
9             defer C.free(unsafe.Pointer(cErr))
```

```
10                return errors.New(C.GoString(cErr))
11        }
12        return nil
13  }
```

Listing 4.17: *The checkpoint `Create()` function in gorocksdb*

Tuning options

In the process of fine tuning RocksDB, which is described in subsection 4.3.6, we had to set the values of multiple configuration options, some of which had not yet been exported to the C API of RocksDB or to the gorocksdb wrapper. To make those options accessible from Go code we added the function `rocksdb_options_set_max_subcompactions()` to the C API and the following functions to gorocksdb: `SetCompactionReadaheadSize()`, `SetEnableWriteThreadAdaptiveYield()`, `SetMaxSubcompactions()`, `GetBlockBasedTableOptions()` and `GetEnv()`.

Database properties

The function `GetProperty()` of RocksDB allows accessing database properties, such as the number of SST files on a specific level of the LSM-tree, the total size of the memtables, the total size of the SST files, database statistics and other useful information. This function was only implemented in the C API and gorocksdb for a plain database and not a transaction database, so we added this extra version to both (i.e., `rocksdb_transactiondb_property_value()` and `GetProperty()` respectively).

4.5 Installation & Configuration

In this section, we provide detailed instructions for the installation of etcd with RocksDB as its storage engine, as well as for the process of setting up an etcd cluster of 3 nodes. Our tests in section 4.6 and benchmarks in chapter 5 have been run with the following versions of software:

- Go 1.8.3
- etcd 3.2.0, modified[31] as mentioned in section 4.2
- RocksDB 5.5.1, extended[32] by our contributions mentioned in section 4.4
- gorocksdb extended[33] by our contributions mentioned in section 4.4

1. Go installation

First of all, to build etcd from source a working installation of Go is required[34].

```
root@etcd1:~# wget https://storage.googleapis.com/golang/
go1.8.3.linux-amd64.tar.gz
root@etcd1:~# tar -C /usr/local -xzf go1.8.3.linux-amd64.tar.gz
root@etcd1:~# export PATH=$PATH:/usr/local/go/bin
root@etcd1:~# mkdir $HOME/go/src/github.com/boolean5
```

Listing 4.18: *Installation of Go*

2. RocksDB installation

The next step is the installation of RocksDB and its dependencies[35][36]. We use RocksDB as a shared library, using the flags mentioned in section 4.2.

```
root@etcd1:~# apt-get update
root@etcd1:~# apt-get install zlib1g-dev
root@etcd1:~# apt-get install libbz2-dev
root@etcd1:~# apt-get install libsnappy-dev
root@etcd1:~# apt-get install liblz4-dev
```

Listing 4.19: *Installation of RocksDB dependencies*

[31] `https://github.com/boolean5/etcd-rocks`
[32] `https://github.com/boolean5/rocksdb/tree/v.5.5.1-extended`
[33] `https://github.com/boolean5/gorocksdb/tree/temp-extended`
[34] `https://golang.org/doc/install`
[35] Complete list of RocksDB dependencies: `https://github.com/facebook/rocksdb/blob/master/INSTALL.md#supported-platforms`
[36] If the machine has just been created it will also be necessary to run `apt install make` and `apt-get install build-essential` before being able to install RocksDB.

```
1 root@etcd1:~# cd $HOME/go/src/github.com/boolean5
2 root@etcd1:~/go/src/github.com/boolean5# wget
     https://github.com/boolean5/rocksdb/archive/v.5.5.1-extended.zip
3 root@etcd1:~/go/src/github.com/boolean5# unzip
     v.5.5.1-extended.zip && mv rocksdb-v.5.5.1-extended rocksdb
4 root@etcd1:~/go/src/github.com/boolean5# cd rocksdb && make
     shared_lib
5 root@etcd1:~/go/src/github.com/boolean5/rocksdb# cp
     librocksdb.so.5.5 /usr/lib
```

Listing 4.20: *Installation of RocksDB*

3. Gorocksdb download

```
1 root@etcd1:~/go/src/github.com/boolean5# wget
     https://github.com/boolean5/gorocksdb/archive/temp-extended.zip
2 root@etcd1:~/go/src/github.com/boolean5# unzip temp-extended.zip
     && mv gorocksdb-temp-extended gorocksdb
```

Listing 4.21: *Downloading gorocksdb*

4. etcd installation

```
1 root@etcd1:~/go/src/github.com/boolean5# wget
     https://github.com/boolean5/etcd-rocks/archive/master.zip
2 root@etcd1:~/go/src/github.com/boolean5# unzip master.zip && mv
     master etcd
3 root@etcd1:~/go/src/github.com/boolean5# cd etcd && ./build
```

Listing 4.22: *Installation of etcd*

To be able to try all the versions of our implementation, which reside in different branches, and not just the final version which is in the master branch, it would be better to use the `git clone https://github.com/boolean5/etcd-rocks` command.

5. Setting up a cluster

To start etcd on each node, from within `/go/src/github.com/boolean5/etcd/bin` we run the following script, in which the correct IP addresses must be filled.

```
#!/bin/bash
 #This script is used to bootstrap an etcd cluster

 #This section is the same for all members
 TOKEN=token-01
 CLUSTER_STATE=new
 NAME_1=etcd-1
 NAME_2=etcd-2
 NAME_3=etcd-3
 HOST_1=192.168.10.11
 HOST_2=192.168.10.12
 HOST_3=192.168.10.13
 CLUSTER=${NAME_1}=http://${HOST_1}:2380, \
${NAME_2}=http://${HOST_2}:2380, \
${NAME_3}=http://${HOST_3}:2380

 #This section is member-specific
 #For machine 1
 THIS_NAME=${NAME_1}
 THIS_IP=${HOST_1}
 ./etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 \
    --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 \
    --listen-client-urls http://${THIS_IP}:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} \
    --initial-cluster-token ${TOKEN}
```

Listing 4.23: *etcd cluster setup script*

The above 5 steps are followed for each of our etcd servers. To set up an etcd client we only have to follow steps 1 and 4, but before running the `build` script, comment out the line responsible for the compilation of etcd in function `etcd_build()`, leaving only the line that compiles `etcdctl`. To apply the configuration of `etcdctl` we issue the command `source etcdctl-config.sh`. The contents of this script are displayed below.

```
1 #!/bin/bash
2  #etcdctl configuration
3  export ETCDCTL_API=3
4  export HOST_1=192.168.10.11
5  export HOST_2=192.168.10.12
6  export HOST_3=192.168.10.13
7  export ENDPOINTS=${HOST_1}:2379,${HOST_2}:2379,${HOST_3}:2379
```

Listing 4.24: *etcd client setup script*

We can confirm that everything is functioning properly with the command `./etcdctl --endpoints=$ENDPOINTS -w table endpoint status`.

4.6 Testing

In this section, we present the testing methods we used to verify that etcd still functions as expected after the replacement of its storage engine with RocksDB. After several testing and bug fixing iterations we managed to make our implementation robust and reliable. Because of the critical role etcd plays in a distributed system, its developers have put a lot of effort in the development of an adequate testing framework. In fact, over half of its code base is dedicated to tests.

Before using the official testing framework of etcd, we ran some rounds of manual testing in order to spot and fix the most obvious bugs. This involved trying the commands exposed by the API one by one and confirming that their output is as expected, both on a standalone etcd node and a local 3-node cluster.

4.6.1 Unit Tests

Unit tests are used to test individual components within a package, such as a function. Etcd provides a `test` script that uses the `go test` tool, mentioned in subsection 2.5.1, to run all the unit tests in the code base. Examples of what these unit tests check include reading and writing key-value pairs to the cluster, adding and removing members from it, taking a snapshot and starting a fresh etcd instance from it, etc.

The `test` script also performs integration tests and end-to-end tests, among other things. Integration tests check client and server interactions by starting an etcd server, sending client requests to it and checking its responses. End-to-end tests configure a local 3-node cluster and simulate real-world operations to verify that the command line interface of etcd is working correctly [114]. After fixing several bugs pointed out by this `test` script we managed to get every unit test to run successfully.

4.6.2 Functional Test Suite

Functional tests are used to test a piece of functionality in a system. The functional test suite of etcd is designed to make sure that etcd fulfills its reliability and robustness guarantees. This is its main workflow, based on the description given by Y.Qin [115]:

1. It sets up a new etcd cluster and makes continuous write requests to it in order to simulate heavy load.
2. It injects a failure into the cluster. Various types of system and network failures are modelled:
    - kill random node: a single machine needs to be upgraded or maintained
    - kill leader
    - kill majority: part of the data center experiences an outage, and the etcd cluster loses quorum
    - kill all nodes: the whole data center experiences an outage and the etcd cluster in the data center is killed
    - kill node for a long time to trigger snapshot when it comes back: a single machine is down due to hardware failure, and requires manual repair

- network partition: the network interface on a single machine or the router or switch in the data center is broken
- slow network

3. It repairs the failure and expects the etcd cluster to recover within a short amount of time (i.e., one minute).
4. It waits until the etcd cluster is fully consistent and making progress.
5. It starts the next round of failure injection.

If the cluster cannot recover from a failure, the functional tester archives the cluster state so that it can be inspected later [115].

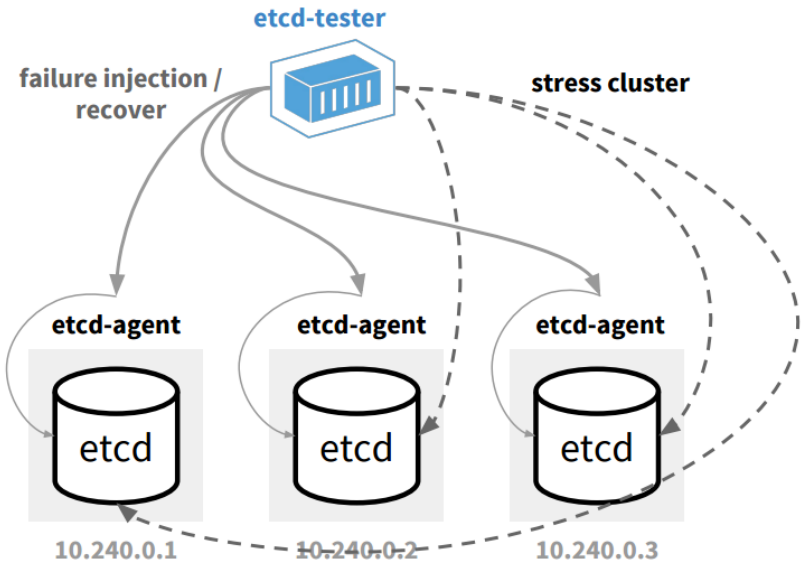


Figure 4.3: *The functional test suite of etcd [114]*

The functional test suite of etcd has two components: the `etcd-agent` and the `etcd-tester`. An instance of the `etcd-agent` daemon runs on every node of the cluster and controls the state of etcd: it starts it, stops it, restarts it, manipulates network configurations etc. `etcd-tester` runs on a single separate machine and injects failures (e.g., by triggering the `etcd-agent` via RPC to stop/start etcd) and verifies the correct operation of etcd. `etcd-agent` stops etcd by sending a `kill` signal to its process, simulates network partitions by manipulating the `iptables`[37], and slow network with the

[37] `https://linux.die.net/man/8/iptables`

help of the `tc` command[38]. It also performs crash tests, such as power loss and I/O error, by triggering failpoints[39] carefully placed in various critical positions in etcd code [114]. Figure 4.3 presents the architecture of the test suite.

To run the functional tests on a single machine using goreman[40], we build with the provided build script and run with the provided Procfile [1]. We can also redirect the output of the tester to a file so that we can inspect it later.

```
./tools/functional-tester/build
goreman -f tools/functional-tester/Procfile start |& tee func-out
```

To run the functional tester on an actual etcd cluster, after building it on each machine we start the `etcd-agent` on every cluster node and the `etcd-tester` on a separate machine with the following commands:

```
./etcd-agent --etcd-path=$ETCD_BIN_PATH
./etcd-tester -agent-endpoints="$MACHINE1_IP:9027,$MACHINE2_IP:9027,
    $MACHINE3_IP:9027" -limit=3 -stress-key-count=250000
        -stress-key-size=100
```

Each time we ran the functional tester we checked the saved output for panics, and the agent nodes for failure archives. With the aid of this information we tracked down numerous bugs and fixed them. Finally, we let the functional test suite run for 10 consecutive rounds with no reported failures of our cluster. In each round all the mentioned types of failures are injected.

[38] `https://linux.die.net/man/8/tc`

[39] `https://github.com/coreos/gofail`

[40] goreman is a clone of foreman written in Go. Foreman is a tool that manages Procfile-based applications. It starts the executables defined in the Procfile as separate processes.

5 Experimental Evaluation

In this chapter, we run a series of carefully selected benchmarks, in order to evaluate the read and write performance of our implementation and compare it to that of etcd with BoltDB as its storage engine. We present the results in graphic form and draw conclusions based on them. More specifically, we will examine and compare the following points between our implementation and the original version of etcd:

- Write and read[1] performance (throughput and latency)
- Scalability with the number of clients
- Effect of the value size on performance
- Disk space consumption
- Memory usage

We will also conduct a performance comparison between the intermediate versions of our implementation and the final version, so as to quantify the contribution of each optimization. Finally, we will attempt to measure the overhead introduced by cgo and assess its impact on the overall performance.

[1] Both for point lookups and range queries.

5.1 Tools, Methodology & Environment

We make exclusive use of the built-in CLI `benchmark` tool of etcd, which communicates with the etcd servers via gRPC. To install it, we use the `build` script of etcd, modified as described in section 4.2 to include the compilation of the `benchmark` tool. What is more, we made a minor modification in the code of the `benchmark` tool: we disabled the printing of its progress bar in order to make the output more readable. Sample `benchmark` commands are:

```
./benchmark --endpoints=${ENDPOINTS} --conns=100 --clients=100 put
    --key-size=8 --sequential-keys=false --key-space-size=100000
        --total=100000 --val-size=256

./benchmark --endpoints=${ENDPOINTS} --conns=100 --clients=100
    range a --consistency=l --total=100000
```

The first one sets up 100 clients that over 100 gRPC connections concurrently perform 100000 random[2] writes of key-value pairs with key size 8 bytes and value size 256 bytes. The `key-space-size` parameter defines the maximum possible number of different keys. The second command sets up 100 clients that over 100 gRPC connections concurrently perform 100000 linearizable[3] point lookups for the key "a". Clients are implemented as separate goroutines. Each client waits for a reply to its current request before sending the next one.

A complete list of the benchmarks we ran can be found in our benchmarking scripts on `https://github.com/boolean5/ntua-thesis`. Each measurement is performed three times and the average values are used. Before moving on from one write benchmark to the next, the data directory of etcd is removed on all nodes. When performing read benchmarks however, we make sure to populate the database beforehand by running a write benchmark that puts 1000000 key-value pairs in etcd.

[2]The `--sequential-keys` parameter does not actually make a big difference in the results. As we have seen in subsection 2.2.1, in the data model of etcd all writes to the storage engine are sequential and sorted by the revision number. However, writes to the in-memory B-tree index are still affected by this parameter.

[3]For an explanation of the concept of linearizable reads, see subsection 2.2.1.

The exact versions of all software used, as well as a guide for the installation process, can be found in section 4.5. The experimental evaluation set up consisted of 4 Amazon Elastic Compute Cloud (EC2) virtual machines of type `m3.xlarge`. Three of them formed the etcd cluster and the remaining one was used as the client. Each had 4 cores, 15GB of memory, 2 SSDs of 40GB and the Ubuntu OS installed.

In the next section, we will use the following naming convention, which is also used in the branches of our repository, to distinguish between the different versions of our implementation:

- `original:` this is an unmodified fork of etcd, with BoltDB as its storage engine, upon which the rest of the versions were based.
- `base:` this is our baseline implementation, with no optimizations applied. For a brief description see subsection 4.3.1.
- `bucket:` this version contains the optimization related to the retrieval of buckets from the backend, described in subsection 4.3.2.
- `optimistic:` in this version, pessimistic transactions were replaced by optimistic ones, an optimization described in subsection 4.3.3.
- `wbwi:` in this version, transactions were replaced by a combination of `WriteBatchWithIndex` and `Snapshot`, an optimization discussed in detail in subsection 4.3.4.
- `wb:` in this version, `WriteBatchWithIndex` was replaced by the more lightweight `WriteBatch`, an optimization introduced in subsection 4.3.5.
- `final:` this is our final, optimal version, obtained after tuning RocksDB as explained in subsection 4.3.6.

5.2 Results

In this section, we describe each of our experiments, present the results in graphic form and attempt to interpret them based on our previous analysis of etcd, BoltDB and RocksDB.

Write performance

The write throughput achieved by each version of our implementation, as well as its improvement as a percentage of the `original` version's throughput, can be studied in Figure 5.1. In this experiment, we ran the `benchmark` tool with the following parameters: `--conns=100 --clients=1000 put --key-size=8 --sequential-keys=false --key-space-size=1000000 --total=1000000 --val-size=256`.

As expected, every optimization we applied yielded a more or less noticeable performance improvement over the previous version of our implementation. More specifically, the `bucket` version provided a significant increase over the rather disappointing throughput of `base` version, by eliminating unnecessary accesses to the storage engine. Versions `optimistic` and `wbwi` offered a small boost over their corresponding previous ones, by switching to more lightweight structures, with `wb` being the first version to surpass the throughput of the `original`. Finally, tuning RocksDB in the `final` version gave us a small extra lead over the `original`.

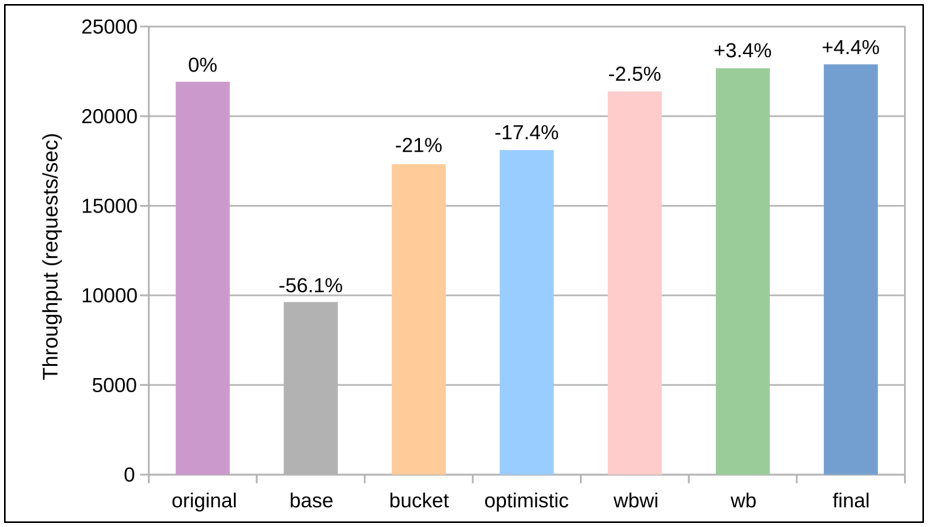


Figure 5.1: *Write throughput of etcd in all versions*

We can also observe the average latency[4] for the same experiment in Figure 5.2. We later ran this experiment with `--clients=100 --total=100000` and `--clients=1000 --total=100000` and the general tendency was the same.

[4]The values depicted in the average latency graphs refer to the latency from the point of view of a single simulated client. To obtain the latency from the point of view of the whole benchmark we just need to divide these values with the number of clients used in the given benchmark.

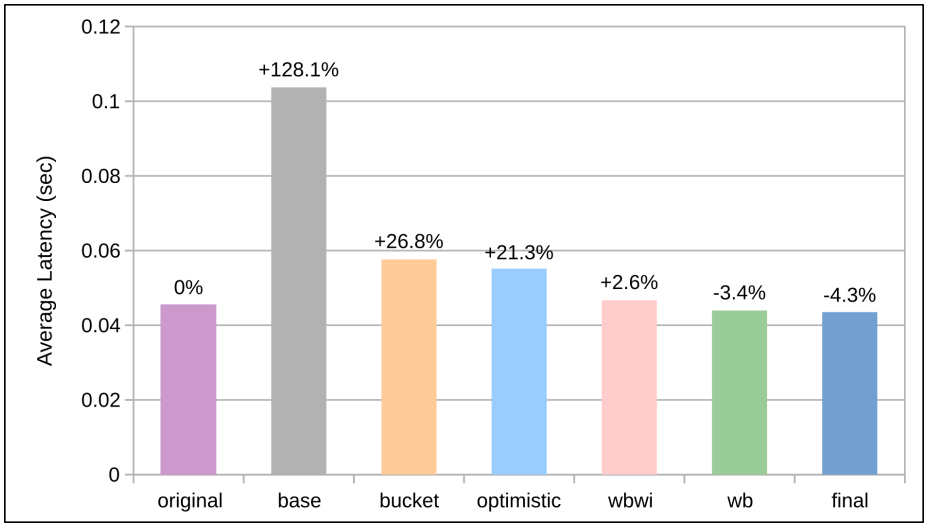


Figure 5.2: *Average write latency of etcd in all versions*

In order to interpret these results with the correct perspective, one should bear in mind that the latency imposed by the storage engine of etcd is only a small contributing factor to the latency depicted in these graphs. As it has been made clear in subsection 2.2.5, the performance of etcd is dominated by disk I/O (i.e., WAL `fsync` operations) and network latency (i.e., message exchange between members for the establishment of consensus). Therefore, the suitability of the LSM-tree and by extension RocksDB for this particular workload should not mislead us into expecting to see a groundbreaking improvement. Nevertheless, as we will see later on, under some circumstances we managed to get a throughput increase as high as 13.3%.

Based on the runtime metrics of etcd[5], the average latency of a put operation as we time it later on and the round trip time as reported by the `ping` command when run from one EC2 instance to another, we can deduce that the contribution of each factor in the duration of a write benchmark's execution and by extension in the average write latency of etcd in our particular environment, follows the rough approximation depicted in Figure 5.3. Besides the WAL, the network and the storage engine, other factors contributing to the latency are the compactions of the MVCC store running in the background and the small overhead of the gRPC API.

It should be noted here that as all of our EC2 instances belonged in the same region and availability zone the round trip time between them was in the order of a few mi-

[5]The metrics `etcd_disk_wal_fsync_duration_seconds_sum` and `etcd_disk_backend_commit_duration_seconds_sum` in particular. All metrics can be accessed with the command `curl -L $ENDPOINTS:2379/metrics > metrics.txt`.

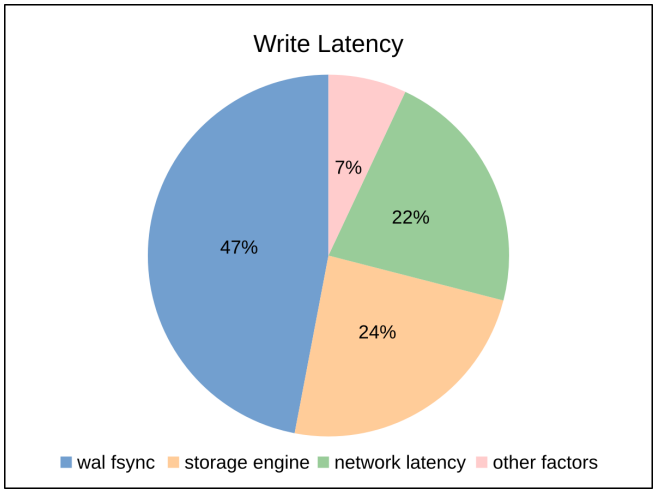


Figure 5.3: *Contributing factors in the write latency of etcd*

croseconds. If the instances were located in different datacenters though, the network latency which could be in the order of tens of milliseconds would constitute the dominant factor in the write latency of etcd.

Read performance

Figure 5.4 shows the *point lookup* throughput of every version of our implementation and Figure 5.5 shows the average point lookup latency. We obtained these results by running the `benchmark` tool with the parameters `--conns=100 --clients=1000 range a --consistency=l --total=1000000`, after having loaded the store with 1000000 key-value pairs and added the key "a".

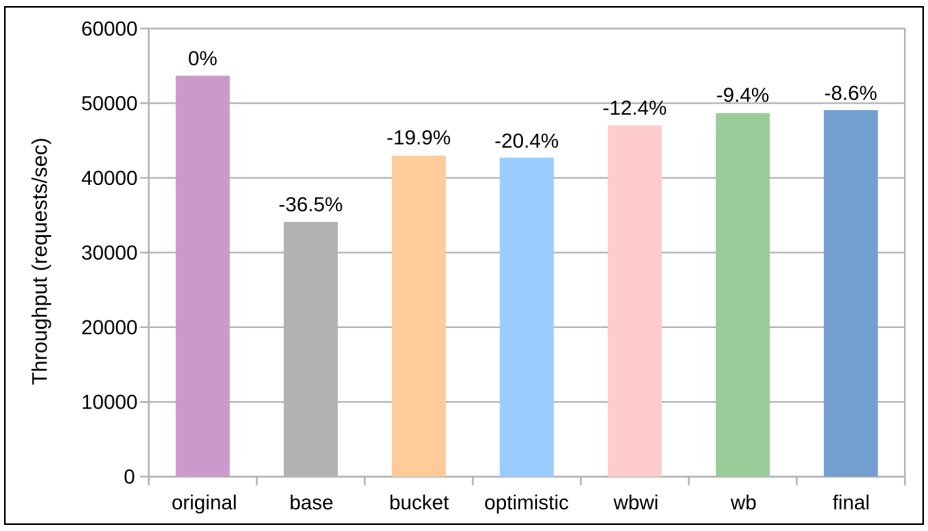


Figure 5.4: *Point lookup throughput of etcd in all versions*

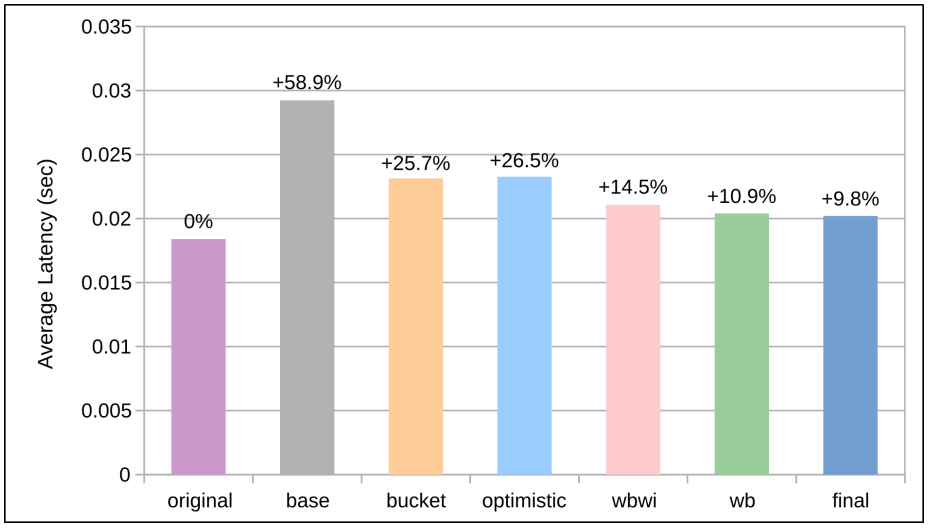


Figure 5.5: *Average point lookup latency of etcd in all versions*

As expected, read performance deteriorates when we switch from BoltDB to RocksDB. However, even though our optimizations were devised with the write performance in mind, they helped bring read performance back to acceptable levels. In the `final` version we note a slight improvement over `wb` due to tuning, which involved the enlargement of the memtables and the addition of Bloom filters.

The throughput and average latency of *range queries* can be observed in Figure 5.6 and Figure 5.7 respectively. These results were obtained by running the `benchmark` tool with the parameters `--conns=100 --clients=100 range a z --consistency=l --total=100000`, after having loaded the store with 1000000 key-value pairs and added the keys "a" to "z".

Here, the performance deterioration is much worse than in the case of point lookups. In general, range queries on LSM-trees are slow because the keys in the range may be scattered in multiple levels of the tree, thus requiring multiple file accesses. To put it in different words, since the keys that are to be returned are not known a priori, every level of the LSM-tree *must* be checked. Furthermore, the benefits of the Bloom filters on point lookups do not extend to range queries, again because the keys that are to be returned are not known a priori. The on-disk structure of BoltDB on the other hand, gives it a clear advantage for this type of workload, as we have seen in subsection 2.3.1.

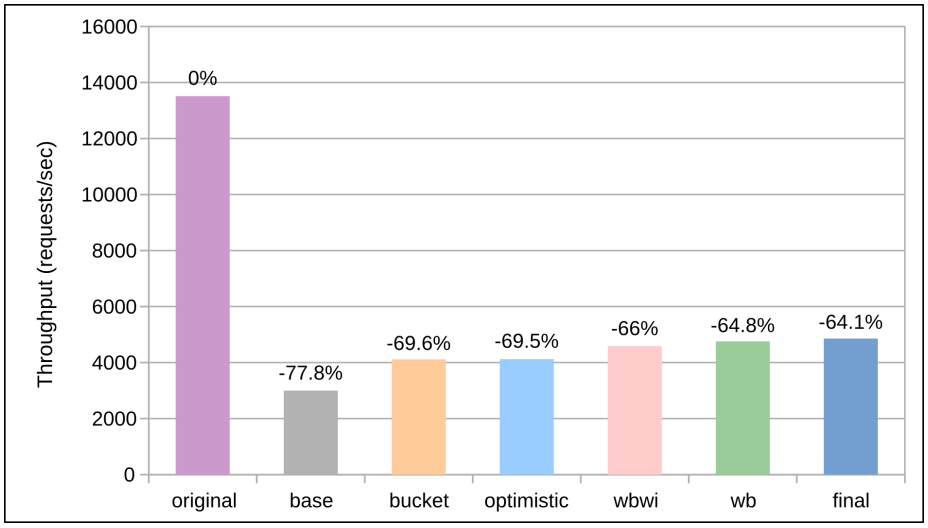


Figure 5.6: *Range query throughput of etcd in all versions*

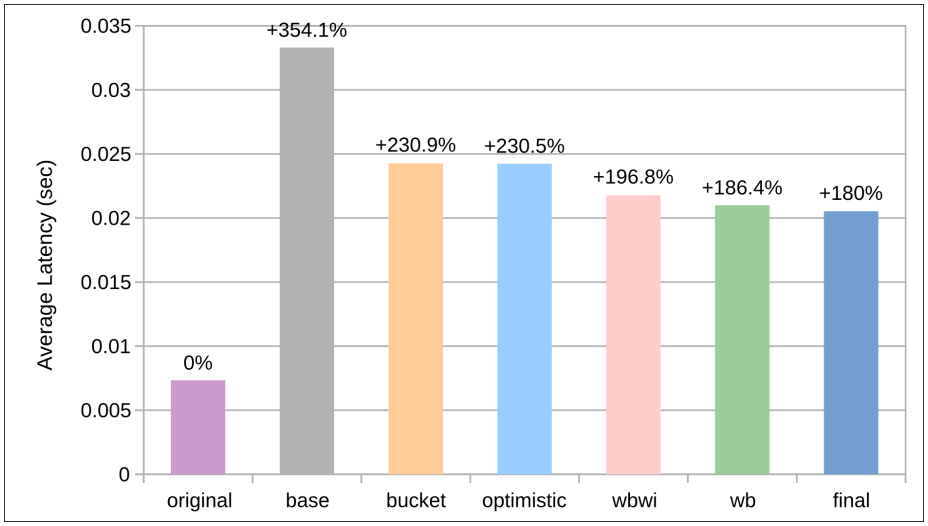


Figure 5.7: *Average range query latency of etcd in all versions*

Cgo overhead

Based on what we have explained in subsection 2.5.2 and on the profiler findings presented in subsection 4.3.2, we have reasons to suspect that the use of cgo has introduced an overhead that does not let our implementation profit from the full potential of RocksDB in terms of performance. The experiment[6] we are about to describe shed some light on the extent of this phenomenon. During the execution of a benchmark that loaded etcd with 1000000 key-value pairs using 1000 clients, we timed[7] put opera-

[6]This experiment, as well as the following one that measures the total duration of commit operations, was not conducted on the EC2 cluster but on our local machine with 4 cores, 4GB of memory, SSD and Ubuntu OS.

[7]Timing in C code was done using the `sys/time.h` library and in Go code using the `time` package.

tions in three different places. Two of them were in the `final` version: the `rocksdb_writebatch_put()` function of the RocksDB C API and the `WriteBatch Put()` function of the gorocksdb wrapper, which uses cgo to call `rocksdb_writebatch_put()`. The third one was in the `original` version: the `Bucket Put()` function of the `bolt` package. Figure 5.8 shows the average duration of those three put operations.

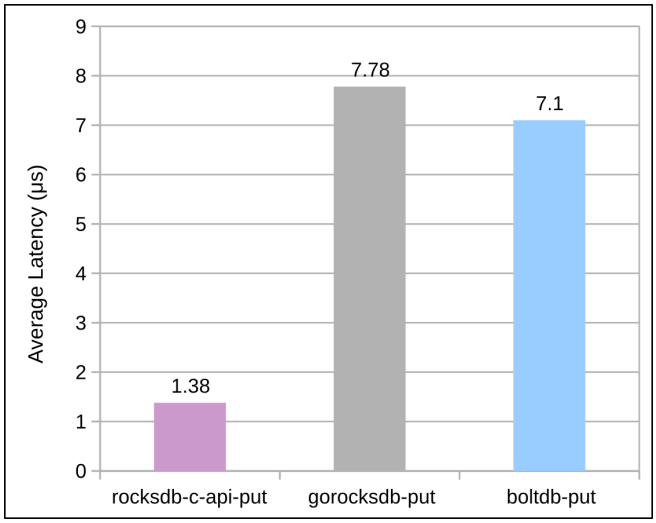


Figure 5.8: *The impact of cgo overhead on put latency*

Apparently, RocksDB demonstrates a clear advantage over BoltDB in terms of put operation performance. However, this advantage is overshadowed by the ~$6.4\mu s$ average overhead of cgo. As a result, a put operation in the `final` version ends up being slightly more costly that its counterpart in the `original` version.

The `rocksdb_writebatch_put()` function is a good example of a case where the use of cgo is discouraged, as its duration is comparable to the overhead of cgo (~4.6 times smaller in this case). Additionally, `rocksdb_writebatch_put()` is called very often in the context of our targeted workload, a fact that makes the small additional latency of a gorocksdb put operation over a BoltDB put operation significant for the overall performance of etcd. The `rocksdb_write()` function on the other hand, used to implement the commit operation, constitutes a much more suitable usecase for cgo. It is called less often and does a substantial amount of work before returning; work that dwarfs the cost of cgo. What is more, the commit operation is where we expect the on-disk structure of RocksDB to be leveraged, as this is where the key-value pairs stored in the `WriteBatch` buffer by put operations are actually written to the LSM-tree. The above, justifies what we see in Figure 5.9.

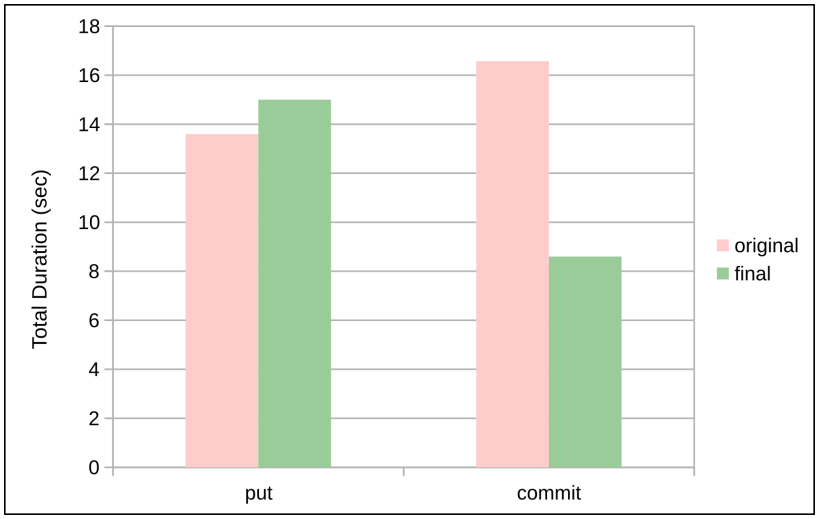


Figure 5.9: *Total duration of put and commit operations*

Figure 5.9 shows the total time spent on commit and put operations both in the `final` and the `original` version of etcd after having run a benchmark that loaded the database with 1000000 key-value pairs using 1000 clients. This data was obtained by accessing the runtime metric `etcd_disk_backend_commit_duration_seconds_sum` and by summing the put operation duration values obtained in the previous experiment.

At this point, it might be useful to review what put and commit actually do in each version. In the `original` version, a put operation traverses the B+ tree to the appropriate page and position and stores the key-value pair there. It is an in-memory operation unless the required pages must be read from disk (e.g., when the dataset does not fit in memory). A commit operation writes the dirty pages of the B+ tree to disk [58]. In the `final` version, a put operation appends a key-value pair to the `WriteBatch`. This operation is always in-memory. A commit operation is translated to a call to `Write()`, which writes the `WriteBatch` to the memtable of the LSM-tree and synchronously updates the WAL.

The effect of the client number & value size on performance

In the next experiment, we examine our implementation's scalability with the number of clients concurrently sending write requests to etcd and compare it to the scalability of the `original` version. We used the `benchmark` tool with the following parameters: `--conns=100 put --key-size=8 --sequential-keys=false --key-space-`

`size=100000 --total=100000 --val-size=256`. The `--clients` parameter was set each time to a different value from the set of 1, 10, 100, and 1000. The result can be observed in Figure 5.10.

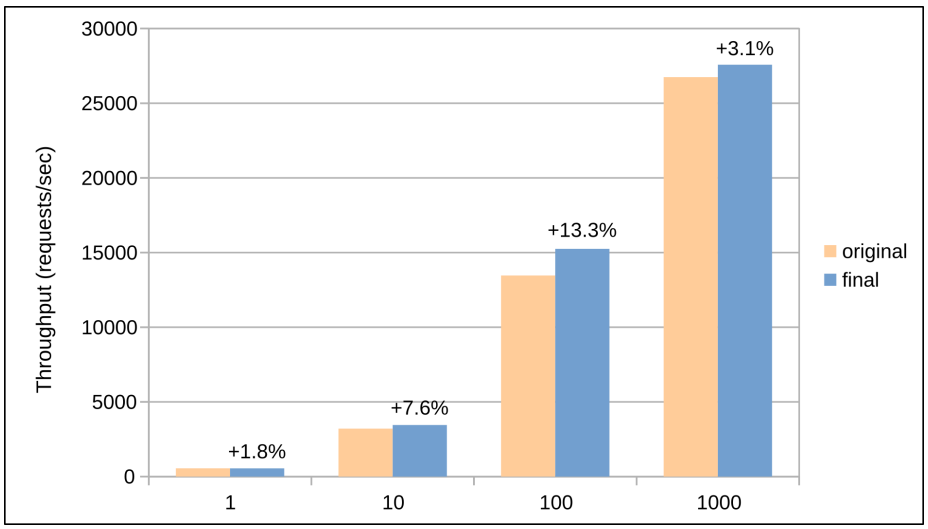


Figure 5.10: *Write throughput of etcd with number of clients*

Regardless of the number of clients, the `final` version of our implementation outperforms the `original`. However, the largest improvement percentages are obtained in the cases of 10 and 100 clients. To interpret this tendency we have to think about the request *batching* policy of etcd, described in detail in section 4.2. When more clients participate in the benchmark the batching becomes more efficient. The number of total put operations is the same in all cases but the number of commit operations changes. With 1000 clients we have ~1000 puts per commit, which means ~100 commits. Similarly, with 100 clients we have ~1000 commits, with 10 clients ~10000 commits and with 1 client 100000 commits. The more the commits the more we benefit from our implementation's superiority of commit performance, demonstrated in Figure 5.9. Naturally, read benchmarks do not exhibit this behaviour.

Nevertheless, one may note that based on this interpretation we should expect the throughput to reach its maximum improvement percentage in the case of 1 client, which is not happening. Indeed, the `etcd_disk_backend_commit_duration_seconds_sum` metric shows that the difference of the total time spent on commit operations between the `original` and the `final` version consistently grows larger when the number of clients decreases. However, this difference represents a smaller percentage of the total benchmark completion time in the case of 1 and 10 clients than

it does in the case of 100 because of the smaller degree of concurrency[8].

In the next experiment, we examine the effect of the value size on the performance of the `final` version of our implementation compared to the `original`. We use the `benchmark` tool with the following parameters: `--conns=100 --clients=100 put --key-size=8 --sequential-keys=false`. The `--val-size` and `--total` parameters were set each time to a different pair from the set of 256 and 1000000, 4000 and 64000, 64000 and 4000, 512000 and 500, 1000000 and 256. The `--key-space-size` parameter follows `--total`. The results are presented in Figure 5.11.

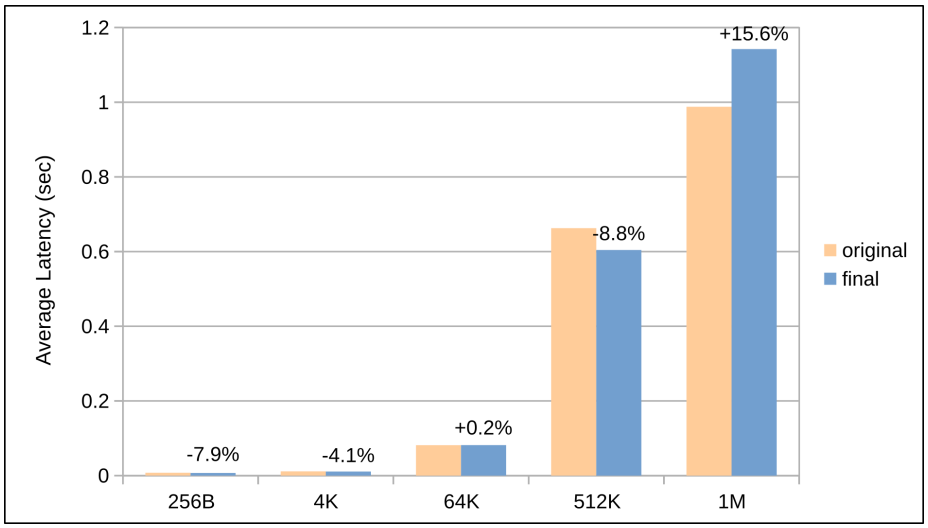


Figure 5.11: *Average write latency of etcd with value size*

Apparently, the `final` version outperforms the `original` in most cases. When the values are as large as 1MB however, BoltDB handles them considerably better. In LSM-trees the insertion of large values might trigger repeated compactions which will incur extra delays. In this particular case, 1MB is the value size that is closest to the total size of the memtables, repeatedly causing them to be flushed to disk.

Disk space consumption

Next, we compare the on-disk size of the backend database between the `final` version and the `original`, using the `du -sh db` command, after having run a benchmark that loads etcd with 4000000 key-value pairs, where the key size was 8 bytes and the value size is 256 bytes. The result may be inspected in Figure 5.12.

[8]etcd also applies batching at the level of the Raft log, so as to amortize the cost of `fsync` over

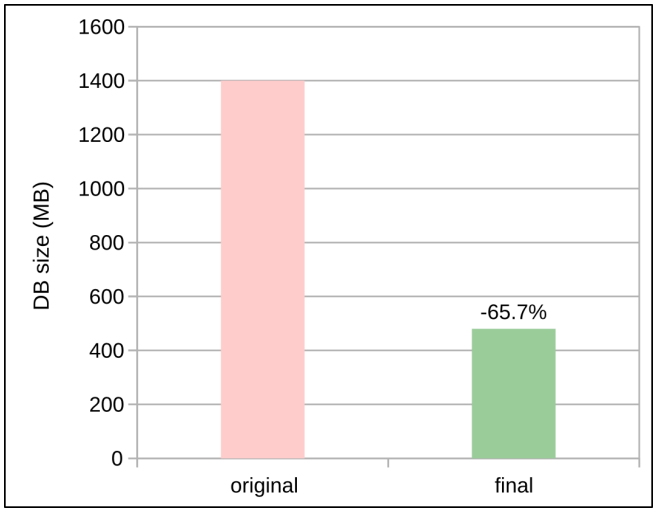


Figure 5.12: *On-disk size of the backend database of etcd*

As expected, since we have configured RocksDB to use Snappy compression and BoltDB does not apply any kind of compression, the disk space consumption is 65.7% smaller in the `final` version. Moreover, as we have discussed in subsection 2.3.3, BoltDB exhibits a kind of internal fragmentation, reserving more disk space than it is actually using at a given time, in order to avoid constant allocations. This is why in the case of the `original` version the size of the database is larger than the size of the inserted data (i.e., 1056MB).

Memory usage

In order to monitor the memory consumption of etcd in the `final` and `original` version we started etcd with the `/usr/bin/time -v ./etcd` command and used the `benchmark` tool to load it with 1000000 key-value pairs, where the key size was 8 bytes and the value size 256 bytes. After the termination of etcd, this command returns the maximum *resident* set size (i.e., the portion of virtual memory occupied by the etcd process that is held in physical memory). Figure 5.13 displays the obtained results.

Obviously, the `original` version uses a lot less memory than the `final`. This result was not unexpected, as we have configured RocksDB to use multiple large memtables. For a use case that demands a smaller memory footprint, the size and number of memtables can be easily adjusted, trading off write performance.

---

multiple requests.

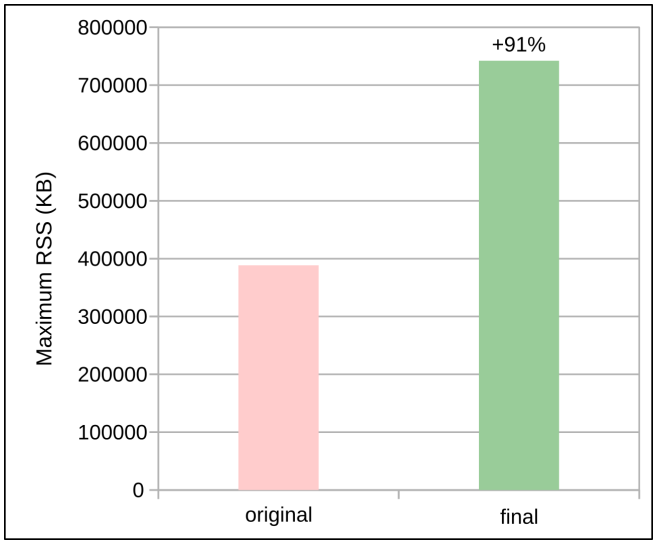


Figure 5.13: *Memory consumption of etcd*

As we have analysed in subsection 2.3.3, most of the memory consumption of BoltDB stems from the fact that the OS will cache as much of the memory-mapped file as it can in memory. In the case of RocksDB, besides the memtables, other factors that contribute to memory usage are the block cache (i.e., where RocksDB caches uncompressed data blocks), page indices for each SST file, and the bloom filters, which in this particular case occupy just 1.25MB as we have configured them with 10 bits per key.

6 Conclusion

In this final chapter, we make an overall assessment of our implementation and outline a few directions for further improvement that we deem worthy of investigation.

6.1 Concluding Remarks

All in all, we have managed to meet the expectations formulated in chapter 3. According to the experimental evaluation results, our implementation achieves a better write performance and disk space consumption, trading off read performance and memory consumption.

In more detail, read performance, as far as point lookups are concerned, has remained within acceptable levels. Range queries however exhibit significant deterioration. The write performance improvement varies between 1.8% and 13.3%, depending on the number of clients concurrently issuing requests. Disk space consumption has dropped by more than 50%, while memory usage has almost doubled. Furthermore, the testing framework has validated that after the changes we applied, etcd still honours its guarantees of reliability, consistency and high-availability. Positive side effects of our implementation include the elimination of the need for a periodic manually initiated defragmentation of the backend and the removal of the storage quota.

This work has demonstrated that an LSM-tree-based approach can make a big difference in the suitability of etcd for write-heavy workloads. However, the overhead of cgo has proven to be far from negligible and has taken its toll on our system's performance.

6.2 Future Work

A very promising alternative implementation, would replace RocksDB with an LSM-tree-based storage engine written purely in Go. Badger [116], a new[1] and quite popular embedded LSM-tree-based key-value store, has emerged precisely from the necessity of a performant RocksDB replacement for Go projects. The main motivation of its developers was the need to avoid the cost and complexity of cgo. Its highly SSD-optimized design is based on a paper published in 2016 by L. Lu et al. [117], which proposes the separation of keys from values. More specifically, according to this design, values are stored in a write-ahead log, called the value log, and keys are stored in the LSM-tree, alongside a pointer to their respective value in the value log. This way, read and write amplification are minimized. Since keys tend to be smaller than values, the generated LSM-tree is also much smaller. As a result, compaction costs are reduced. Also, with fewer levels in the LSM-tree, the number of file accesses required to retrieve a key in the worst case is reduced [118]. Badger's API can be directly mapped to that of BoltDB, making the transition trivial.

Furthermore, it would be interesting to conduct an experiment in which the dataset would be larger than the available memory, with the intent of comparing the behaviour of the two storage engines under I/O-bound workloads. BoltDB would be forced to constantly read pages from disk in order to satisfy incoming requests, as we have described in subsection 2.4.3. Another proposal, in the direction of getting the most out of our current implementation, would be to automate the process of optimally tuning RocksDB with an appropriate script. Storing the contents of the "meta" bucket in a separate RocksDB column family, might also offer a performance benefit, as this would render updating the consistent index of etcd more lightweight. What is more, in order to mitigate the cost of cgo, we could add a layer in Go that would buffer the updates directed to the `WriteBatch` and apply them all together at commit time, thus reducing thousands of cgo calls to one.

An additional idea, is to apply sharding on the key space at the level of RocksDB, as proposed in its tuning guide [64], so as to allow compactions to fully utilize storage concurrency. Taking this one step further, by storing the shards on different disks, we

[1] Badger was announced in May 2017.

could spread write operations across them, increasing the throughput of etcd significantly. Finally, we could modify etcd to support multiple leaders, each responsible for a different shard.

Bibliography

[1] etcd project, `https://github.com/coreos/etcd`, accessed August 14th, 2017.

[2] N. Christenson, *Sendmail Performance Tuning*, Addison-Wesley Professional, 2003.

[3] T. Critchley, *High-Performance IT Services*, CRC Press, 2016.

[4] Z. Cao et al., *LogKV: Exploiting Key-Value Stores for Event Log Processing*, Proceedings of the 6th Conference on Innovative Data Systems Research, CIDR 2013, California, USA, January 2013.

[5] *Exploring Performance of etcd, Zookeeper and Consul Consistent Key-value Datastores*, G. Lee, CoreOS blog, February 2017, `https://coreos.com/blog/performance-of-etcd.html`, accessed August 27th, 2017.

[6] R. Sears and R. Ramakrishnan *bLSM: A General Purpose Log Structured Merge Tree*, Proceedings of the 2012 ACM International Conference on Management of Data, SIGMOD '12, Arizona, USA, May 2012.

[7] *Log Structured Merge Trees*, B. Stopford, February 2015, `http://www.benstopford.com/2015/02/14/log-structured-merge-trees/`, accessed August 31st, 2017.

[8] CockroachDB project, `https://github.com/cockroachdb/cockroach/`, accessed August 16th, 2017.

[9] *Hello World: Meet CockroachDB, the Resilient SQL Database*, J. Edwards, October 2015, https://thenewstack.io/cockroachdb-unkillable-distributed-sql-database/, accessed August 23rd, 2017.

[10] *etcd versus other key-value stores*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/learning/why.md, accessed August 17th, 2017.

[11] TiKV project, https://github.com/pingcap/tikv, accessed August 17th, 2017.

[12] TiDB Documentation, https://github.com/pingcap/docs, accessed August 17th, 2017.

[13] MongoDB architecture, https://www.mongodb.com/mongodb-architecture, accessed August 18th, 2017.

[14] MongoDB storage integration layer for the Rocks storage engine, https://github.com/mongodb-partners/mongo-rocks, accessed August 18th, 2017.

[15] S. Nakamura and K. Shudo, *MyCassandra: A Cloud Storage Supporting both Read Heavy and Write Heavy Workloads*, Proceedings of the 5th Annual International Systems and Storage Conference, SYSTOR 2012, Haifa, Israel, June 2012.

[16] *Distributed data store*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Distributed_data_store&oldid=777883924, accessed August 19th, 2017.

[17] *What is a distributed storage system and why is it important?*, StorPool Distributed Storage, https://storpool.com/blog/what-is-distributed-storage-system, accessed August 19th, 2017.

[18] *Scalability*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Scalability&oldid=789065570, accessed August 20th, 2017.

[19] *What is a Key-Value Store?*, Aerospike, http://www.aerospike.com/what-is-a-key-value-store/, accessed August 21st, 2017.

[20] *Top 5 Considerations When EvaluatingNoSQL Databases*, A MongoDB White Paper, 2016. https://www.mongodb.com/nosql-explained, accessed August 21st, 2017.

[21] *NoSQL*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=NoSQL&oldid=794350817, accessed August 21st, 2017.

[22] *Exploring the Different Types of NoSQL Databases Part ii*, G. Kumar, https://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases, accessed August 21st, 2017.

[23] *DB-Engines Ranking of Key-value Stores*, DB-Engines, Knowledge Base of Relational and NoSQL Database Management Systems, https://db-engines.com/en/ranking/key-value+store, accessed August 21st, 2017.

[24] S. Gilbert and N. Lynch, *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*, ACM SIGACT News, New York, USA, June 2002.

[25] *CAP Theorem*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=CAP_theorem&oldid=791473737, accessed August 21st, 2017.

[26] *CAP Theorem*, T. Viraj, February 2016, https://blingtechs.blogspot.gr/2016/02/cap-theorem.html, accessed August 21st, 2017.

[27] *Please stop calling databases CP or AP*, M. Kleppmann, May 2015, https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html, accessed August 22nd, 2017.

[28] G. Kohad et al., *Concepts and Techniques of Transaction Processing of Distributed Database Management Systems*, International Journal of Computer Architecture and Mobility, Volume 1, Issue 8, June 2013.

[29] L. Lamport, *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, Issue: 9, September 1979.

[30] A. S. Tanenbaum and M. Van Steen, *Distributed Systems - Principles and Paradigms*, Prentice Hall, 2007.

[31] *Isolation Levels in the Database Engine*, Microsoft Technet, https://technet.microsoft.com/en-us/library/ms189122(v=SQL.105).aspx, accessed August 23rd, 2017.

[32] *Write-ahead logging*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Write-ahead_logging&oldid=709989245, accessed August 25th, 2017.

[33] P. A. Bernstein and N. Goodman, *Multiversion Concurrency Control - Theory and Algorithms*, ACM Transactions on Database Systems (TODS), Volume 8, Issue 4, New York, USA, December 1983.

[34] *Multiversion Concurrency Control*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Multiversion_concurrency_control&oldid=796567230, accessed August 25th, 2017.

[35] D. Ongaro and J. Ousterhout, *In Search of an Understandable Consensus Algorithm*, Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference, Philadelphia, USA, June 2014.

[36] *Heartbeat*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Heartbeat_(computing)&oldid=786649097, accessed August 24th, 2017.

[37] L. Lamport, *The Part-Time Parliament*, ACM Transactions on Computer Systems, Volume 16, Issue 2, SYSTOR 2012, New York, USA, May 1998.

[38] *List of Raft Implementations*, The Raft Consensus Algorithm (Official Website), https://raft.github.io/#implementations, accessed August 24th, 2017.

[39] *KV API guarantees*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/learning/api_guarantees.md, accessed August 25th, 2017.

[40] *Data model*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/learning/data_model.md, accessed August 25th, 2017.

[41] *Maintenance*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/op-guide/maintenance.md, accessed August 26th, 2017.

[42] *etcd API Reference*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/dev-guide/api_reference_v3.md, accessed August 26th, 2017.

[43] *Disaster recovery*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/op-guide/recovery.md, accessed August 26th, 2017.

[44] *Frequently Asked Questions (FAQ)*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/faq.md, accessed August 26th, 2017.

[45] *CoreOS Delivers etcd v2.3.0 with Increased Stability and v3 API Preview*, X. Li, CoreOS Blog, March 2016, https://coreos.com/blog/etcd-v230.html, accessed August 26th, 2017.

[46] *The Docker Ecosystem: Service Discovery and Distributed Configuration Stores* , J. Ellingwood, February 2015, https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-service-discovery-and-distributed-configuration-stores, accessed August 27th, 2017.

[47] *Production users*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/production-users.md, accessed August 27th, 2017.

[48] ZooKeeper Documentation, https://zookeeper.apache.org/doc/trunk/zookeeperOver.html, accessed August 27th, 2017.

[49] *Consul vs. ZooKeeper, doozerd, etcd*, Consul (Official Website), https://www.consul.io/intro/vs/zookeeper.html, accessed August 27th, 2017.

[50] *Performance*, etcd Documentation, https://github.com/coreos/etcd/blob/master/Documentation/op-guide/performance.md, accessed August 27th, 2017.

[51] D. Comer, *The Ubiquitous B-Tree*, ACM Computing Surveys (CSUR), Volume 11, Issue 2, New York, USA, June 1979.

[52] D. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching* , Addison-Wesley, 2nd edition, 1998.

[53] *B-tree*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=B-tree&oldid=794880526, accessed August 28th, 2017.

[54] *B+ tree*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=B%2B_tree&oldid=790327312, accessed August 29th, 2017.

[55] *How the append-only btree works*, http://www.bzero.se/ldapd/btree.html, accessed September 1st, 2017.

[56] *Memory-mapped file*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Memory-mapped_file&oldid=784434520, accessed August 29th, 2017.

[57] *Bolt — an embedded key/value database for Go*, Progville, January 2015, https://www.progville.com/go/bolt-embedded-db-golang/, accessed August 29th, 2017.

[58] BoltDB project, https://github.com/boltdb/bolt, accessed August 29th, 2017.

[59] *WTF Dial: Data storage with BoltDB*, B. Johnson, September 2016, https://medium.com/wtf-dial/wtf-dial-boltdb-a62af02b8955, accessed August 29th, 2017.

[60] *database file size not updating?*, BoltDB, Issue#308, https://github.com/boltdb/bolt/issues/308, accessed August 29th, 2017.

[61] P. O'Neil et al., *The log-structured merge-tree (LSM-tree)*, Acta Informatica, Volume 33, Issue 4, June 1996.

[62] *SSTable and Log Structured Storage: LevelDB*, I. Grigorik, February 2012, `https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/`, accessed August 30th, 2017.

[63] *Research Scientists Put RocksDB on Steroids*, Yahoo!, April 2015, `https://research.yahoo.com/news/research-scientists-put-rocksdb-steroids`, accessed August 31st, 2017.

[64] *RocksDB Tuning Guide*, RocksDB Wiki, `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`, accessed August 31st, 2017.

[65] *Bloom filter*, In Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=822632093`, accessed September 1st, 2017.

[66] *Bloom Filters: Is element x in set S?*, A. Tiwari, April 2017, `https://abhishek-tiwari.com/bloom-filters-is-element-x-in-set-s/`, accessed September 1st, 2017.

[67] *Universal Compaction*, RocksDB Wiki, `https://github.com/facebook/rocksdb/wiki/Universal-Compaction`, accessed August 31st, 2017.

[68] *Leveled Compaction*, RocksDB Wiki, `https://github.com/facebook/rocksdb/wiki/Leveled-Compaction`, accessed September 3rd, 2017.

[69] *RocksDB Basics*, RocksDB Wiki, `https://github.com/facebook/rocksdb/wiki/RocksDB-Basics`, accessed September 2nd, 2017.

[70] B. C. Kuszmaul, *A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees*, A Tokutek White Paper, April 2014.

[71] RocksDB IAmA page - Reddit, `https://www.reddit.com/r/IAmA/comments/3de3cv/we_are_rocksdb_engineering_team_ask_us_anything/`, accessed September 4th, 2017.

[72] *TokuMX Fractal Tree(R) indexes, what are they?*, Z. Kasheff, July 2013, https://www.percona.com/blog/2013/07/02/tokumx-fractal-treer-indexes-what-are-they/, accessed September 1st, 2017.

[73] *Optimizing RocksDB for Open-Channel SSDs*, J. Gonzalez, October 2015, https://www.slideshare.net/JavierGonzlez49/optimizing-rocksdb-for-openchannel-ssds, accessed September 2nd, 2017.

[74] *WriteBatchWithIndex: Utility for Implementing Read-Your-Own-Writes*, S. Dong, RocksDB Blog, February 2015, http://rocksdb.org/blog/2015/02/27/write-batch-with-index.html, accessed September 3rd, 2017.

[75] *Hard Link Definition*, The Linux Information Project, http://www.linfo.org/hard_link.html, accessed September 3rd, 2017.

[76] *RocksDB FAQ*, RocksDB Wiki, https://github.com/facebook/rocksdb/wiki/RocksDB-FAQ, accessed September 3rd, 2017.

[77] *Transactions*, RocksDB Wiki, https://github.com/facebook/rocksdb/wiki/Transactions, accessed September 3rd, 2017.

[78] *How we keep track of live SST files*, RocksDB Wiki, https://github.com/facebook/rocksdb/wiki/How-we-keep-track-of-live-SST-files, accessed September 3rd, 2017.

[79] *Features Not in LevelDB*, RocksDB Wiki, https://github.com/facebook/rocksdb/wiki/Features-Not-in-LevelDB, accessed September 2nd, 2017.

[80] RocksDB Project, https://github.com/facebook/rocksdb/, accessed September 4th, 2017.

[81] *Google's Go (golang) Programming Language*, C. Brown, https://medium.com/@charliebrown9/googles-go-golang-programming-language-c7953e826b2e, accessed October 11th, 2017.

[82] *Why Golang is doomed to succeed*, Texlution, June 2015, https://texlution.com/post/why-go-is-doomed-to-succeed/, accessed October 11th, 2017.

[83] *Go Is Unapologetically Flawed, Here's Why We Use It*, T. Treat, May 2015, `http://bravenewgeek.com/go-is-unapologetically-flawed-heres-why-we-use-it/`, accessed October 17th, 2017.

[84] *Why Go's design is a disservice to intelligent programmers*, G. Willoughby, March 2015, `http://nomad.so/2015/03/why-gos-design-is-a-disservice-to-intelligent-programmers/`, accessed October 12th, 2017.

[85] *Don't repeat yourself*, In Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Don%27t_repeat_yourself&oldid=804674085`, accessed October 17th, 2017.

[86] C. Doxsey, *An Introduction to Programming in Go*, O'Reilly, 2016.

[87] K. Seguin, *The Little Go Book*, `https://github.com/karlseguin/the-little-go-book`, accessed April 27th, 2016.

[88] *How Stacks are Handled in Go*, D. Morsing, `https://blog.cloudflare.com/how-stacks-are-handled-in-go/`, accessed November 3rd, 2017.

[89] *Companies Currently Using Go Throughout the World*, Go Documentation, `https://github.com/golang/go/wiki/GoUsers`, accessed October 27th, 2017.

[90] *Go Slices: usage and internals*, The Go Blog, `https://blog.golang.org/go-slices-usage-and-internals`, accessed December 27th, 2017.

[91] *Defer, Panic, and Recover*, The Go Blog, `https://blog.golang.org/defer-panic-and-recover`, accessed December 27th, 2017.

[92] *Command go*, Go Documentation, `https://golang.org/cmd/go/#hdr-Calling_between_Go_and_C`, accessed October 30th, 2017.

[93] *Command cgo*, Go Documentation, `https://golang.org/cmd/cgo/`, accessed October 30th, 2017.

[94] *C? Go? Cgo!*, The Go Blog, `https://blog.golang.org/c-go-cgo`, accessed October 31st, 2017.

[95] *cgo is not Go*, D. Cheney, https://dave.cheney.net/2016/01/18/cgo-is-not-go, accessed November 2nd, 2017.

[96] *The Cost and Complexity of Cgo*, T. Schottdorf, https://www.cockroachlabs.com/blog/the-cost-and-complexity-of-cgo/, accessed November 6th, 2017.

[97] *Packages*, Go Documentation, https://golang.org/pkg/, accessed November 7th, 2017.

[98] *cgocall.go*, Go runtime source code, https://golang.org/src/runtime/cgocall.go, accessed November 7th, 2017.

[99] *Foreign Function Interface in Go and Assembly*, golang-nuts mailing list, https://groups.google.com/forum/#!msg/golang-nuts/NNaluSgkLSU/0bq1kXZueCwJ, accessed November 7th, 2017.

[100] *What is the overhead of calling a C function from Go?*, golang-nuts mailing list, https://groups.google.com/forum/#!topic/golang-nuts/RTtMsgZi88Q, accessed November 7th, 2017.

[101] *Cross compiler*, In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Cross_compiler&oldid=807526997, accessed November 7th, 2017.

[102] *etcd community hangout 2015-12-01*, December 2015, https://www.youtube.com/watch?v=J5AioGtEPeQ&feature=youtu.be&t=2170, accessed September 4th, 2017.

[103] gorocksdb Project, https://github.com/tecbot/gorocksdb, accessed September 4th, 2017.

[104] *etcd as a general-purpose key-value store*, etcd-dev mailing list, December 2016, https://groups.google.com/forum/#!topic/etcd-dev/vCeSLBKC_M8, accessed September 5th, 2017.

[105] *Maximum database size on 32bit architectures*, BoltDB, Issue#280, https://github.com/boltdb/bolt/issues/280, accessed September 5th, 2017.

[106] *etcdctl*, etcd Documentation, `https://github.com/coreos/etcd/tree/master/etcdctl`, accessed October 6th, 2017.

[107] *RocksDB Transactions*, RocksDB Meetup - A. Giardullo, December 2015, `https://www.youtube.com/watch?v=tMeon8FHF3I`, accessed October 9th, 2017.

[108] *tar (computing)*, In Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Tar_(computing)&oldid=804463187`, accessed October 10th, 2017.

[109] *Type assertions*, A Tour of Go, `https://tour.golang.org/methods/15`, accessed September 4th, 2017.

[110] *Golang: Working with Gzip and Tar*, S. Ralchev, `http://blog.ralch.com/tutorial/golang-working-with-tar-and-gzip/`, accessed August 7th, 2017.

[111] *Profiling Go Programs*, The Go Blog, `https://blog.golang.org/profiling-go-programs`, accessed December 29th, 2017.

[112] *Fine-tuning RocksDB for NVMe SSD*, P. Krishnamoorthy and C. Choi, Percona Live: Data Performance Conference, California, USA, April 2016, `https://www.percona.com/live/data-performance-conference-2016/sites/default/files/slides/Percona_RocksDB_v1.3.pdf`

[113] K. Ouaknine et al., *Optimization of RocksDB for Redis on Flash*, Proceedings of the International Conference on Compute and Data Analysis, ICCDA 2017, Florida, USA, May 2017.

[114] *Testing Distributed Systems in Go*, G. Lee, CoreOS blog, January 2017, `https://coreos.com/blog/testing-distributed-systems-in-go.html`, accessed December 30th, 2017.

[115] *New Functional Testing in etcd*, Y. Qin, CoreOS blog, May 2015, `https://coreos.com/blog/new-functional-testing-in-etcd.html`, accessed December 30th, 2017.

[116] Badger project, `https://github.com/dgraph-io/badger`, accessed January 21st, 2018.

[117] L. Lu et al., *WiscKey: Separating Keys from Values in SSD-Conscious Storage*, Proceedings of USENIX FAST '16: 14th USENIX Conference on File and Storage Technologies, California, USA, February 2016.

[118] *Introducing Badger: A fast key-value store written purely in Go*, M.R. Jain, May 2017, `https://blog.dgraph.io/post/badger/`, accessed September 4th, 2017.