Αποδοτική Δεικτοδότηση Αποθήκευσης για Δομημένα και Αδόμητα Δεδομένα

Η ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

Τμήματος Μηχανικών Η/Υ και Πληροφορικής
Εξεταστική Επιτροπή

από τον

Γεώργιο Μαργαρίτη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Απρίλιος 2014

Τριμελής Συμβουλευτική Επιτροπή

- Στέργιος Αναστασιάδης, Επίκουρος Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Λεωνίδας Παληός, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Παναγιώτης Βασιλειάδης, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων

Επταμελής Εξεταστική Επιτροπή

- Στέργιος Αναστασιάδης, Επίκουρος Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Λεωνίδας Παληός, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Παναγιώτης Βασιλειάδης, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Ευαγγελία Πιτουρά, Καθηγήτρια του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Αλέξης Δελής, Καθηγητής του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών
- Παναγιώτης Τριανταφύλλου, Καθηγητής του School of Computer Science, University of Glasgow, UK
- Νικόλαος Κούδας, Καθηγητής του Department of Computer Science, University of Toronto, Canada

Dedication

To my family.

To the giants on whose shoulders we stand to see further.

Στην οικογένειά μου.

Στους γίγαντες στων οποίων τους ώμους στεκόμαστε για να δούμε πιο μακρυά.

Acknowledgements

First and foremost, I would like to thank my supervisor Professor Stergios Anastasiadis for his help and research guidance, his full support and invaluable input, both theoretical and technical. He taught me to set high standards and provided the support required to meet those standards. I learned a great deal from him.

I would also like to thank the rest members of the examination committee, Prof. Leonidas Palios, Prof. Panos Vassiliadis, Prof. Evaggelia Pitoura, Prof. Peter Triantafillou, Prof. Alex Delis, and Prof. Nick Koudas for their kind comments and healthy criticism.

I am very grateful to my parents for their continuous psychological (and economical) support, for their encouragement and their patience during my research. This dissertation would definitely not be possible without their help, especially on these tough times.

Charles Bukowski[1] once said: "*Some people never go crazy. What truly horrible lives they must lead*". My last years were truly amazing, so I would like to thank the following for sharing their craziness and time with me: Konstantinos Karras, for all the laughs, beers, and hangovers we had; Evaggelia Liggouri, for sharing with me 10 wonderful years; Argyris Kalogeratos and Andreas Vasilakis, for the countless hours we spent brainstorming about the killer app that would make us rich without effort (I'm sure our book of "Epic Ideas" will someday be published); Andromachi Hatzieleftheriou[2], for her extraordinary tasty meals, pies, cakes and cookies; Giorgos Kappes, Eirini Micheli, Vasilis Papadopoulos and Christos Theodorakis, all members of the Systems Research Group, for all the interesting scientific and not–so–scientific talks we had within these four walls; and Nikos Papanikos, for reminding me that I occasionally needed to take a break from research

[1] Charles Bukowski (1920 – 1994) was a German-born American poet, novelist and short story writer.

[2] It took me about 10 seconds to write this name, and I also had to double-check it for correct spelling.

(coincidentally, this happened every time he needed to smoke). Last but definitely not least, many thanks to Vassillis Delis and Stathis Moraitidis, for saving me countless times from zombies in Left 4 Dead 2; I owe you my life guys.

I would also like to thank Antonis Mpalasas and Fotis Pindis, ever-lasting friends from high-school, and Kostas Karabelas, Nikos Giotis, Maria Panagiotidou, Maria Goutra and Pavlos Xouplidis, all friends from Ioannina with whom I enjoyed many hours of surrealistic and non-sense discussions about the universe and everything. Special thanks goes also to Maria Alexiou, the living encyclopedia of beer, Mitsos Papageorgiadis, the living legend of drinking beer, and Eleni Marmaridou, the sweet "cat lady" that tries to make me a vegan (you know its futile, right?), with all of whom I share more than just a strong friendship.

Finally, the last 13 years I spent in Ioannina would certainly not have been the same without the Takis and Sakis restaurant (I owe about 5Kg of my weight to their delicious food and tsipouro), and the bars "Berlin", "Lemon" and "Parenthesis" where I practiced for hours –and perfected– my air-guitaring techniques. I would also like to personally thank the anonymous inventor of tsipouro, which I consider the third most important invention after the wheel and Super Nintendo.

This research was partially funded by the Bodossaki Foundation, to which I am very thankful. It has also been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Text Search 4

1.3 Scalable Datastores 5

1.4 Thesis Contribution 6

1.5 Thesis Organization 8

2 Background and Related Research **10**

2.1 Full-Text Search 10

2.1.1 Preliminaries 11

2.1.2 Online Index Maintentance 14

2.1.3 Real-Time Search 18

2.2 Large-Scale Data Management 19

2.2.1 Scalable Datastores 19

2.2.2 Storage Organization 22

2.2.3 Related Issues 25

3 Incremental Text Indexing for Fast Disk-Based Search **28**

3.1 Introduction 28

3.2 Background 31

3.3 Motivation 34

3.3.1 The Search Cost of Storage Fragmentation 34

4 Selective Range Flush and Unified Range Flush Methods **37**

4.1 Problem Definition 38

4.2 System Architecture . . . 39
4.3 The Selective Range Flush Method . . . 40
4.4 Evaluation of Selective Range Flush . . . 43
4.5 Sensitivity of Selective Range Flush . . . 45
4.6 The Unified Range Flush Method . . . 46
4.7 Prototype Implementation . . . 49
4.7.1 Memory Management and I/O . . . 51

5 Performance Evaluation of Incremental Text Indexing 53
5.1 Experimentation Environment . . . 54
5.2 Building the Inverted File . . . 55
5.3 Query Handling . . . 57
5.4 Sensitivity of Unified Range Flush . . . 59
5.5 Storage and Memory Management . . . 61
5.6 Scalability across Different Datasets . . . 63
5.7 Summary . . . 65

6 Range-Based Storage Management for Scalable Datastores 67
6.1 Introduction . . . 67
6.2 Motivation . . . 69
6.3 System Assumptions . . . 72
6.4 Design and Architecture . . . 73
6.4.1 The Rangetable Structure . . . 73
6.4.2 The Rangemerge Method . . . 75
6.5 Prototype Implementation . . . 77
6.6 Summary . . . 79

7 Performance Evaluation of Rangemerge 80
7.1 Experimentation Environment . . . 81
7.2 Query Latency and Disk Files . . . 82
7.3 Insertion Time . . . 85
7.4 Sensitivity Study . . . 87
7.5 Memory Size . . . 89

7.6 Key Distribution . . . 89
7.7 Solid-State Drives . . . 90
7.8 Discussion . . . 92
7.8.1 Compaction I/O Intensity . . . 92
7.8.2 Queries . . . 93
7.8.3 Updates . . . 94
7.8.4 Availability and Recovery . . . 94
7.8.5 Caching . . . 95
7.9 Summary . . . 95

8 Implementation of Rangemerge in a Production System 97
8.1 LevelDB Implementation . . . 97
8.1.1 Memory Management . . . 99
8.1.2 Logging . . . 99
8.1.3 Recovery . . . 101
8.1.4 Other Merging Strategies . . . 102
8.2 Performance Evaluation . . . 102
8.2.1 Logging Performance . . . 103
8.2.2 Insertion Time . . . 104
8.2.3 Interference of Queries and Inserts . . . 106
8.3 Summary . . . 109

9 Theoretical Analysis 110
9.1 I/O Complexity of Unified Range Flush . . . 110
9.2 I/O Complexity of Rangemerge . . . 114
9.3 Summary . . . 116

10 Conclusions and Future Work 117
10.1 Conclusions . . . 117
10.2 Future Work . . . 118

List of Figures

2.1 (a) A simple text collection of six documents. (b) The lexicon and the inverted lists for the specific document collection. 12

2.2 Merges and files produced after the first 10 memory flushes for the (a) Immediate Merge or Remerge, and (b) Nomerge methods. Numbers within nodes represent size. 15

2.3 Merge sequence of Geometric Partitioning with $r = 3$, for the first 10 memory flushes. Numbers within nodes represent size. 16

2.4 Index maintenance approach for the (a) Logarithmic Merge and (b) Hybrid Immediate Merge methods. 17

2.5 (a) Each table is partitioned into a number of tablets for load balancing, which are subsequently assigned to servers. (b) A master node keeps the mapping between tablets and servers. Clients must first contact the master node to store or access data. 19

2.6 Dynamo decentralized architecture. Any node on the ring can coordinate a request from a client. We assume the ring space is (0,400) and nodes A, B, C, D are assigned values 100, 200, 300 and 400 respectively. 20

3.1 Hybrid Immediate Merge only applies partial flushing to long (frequent) terms, while Selective Range Flush (SRF) and Unified Range Flush (URF) partially flush both short (infrequent) and long terms. Unlike SRF, URF organizes all postings in memory as ranges, allows a term to span both the in-place and merge-based indices, and transfers postings of a term from the merge-based to the in-place index every time they reach a size threshold T_a (see also Section 4.6). 32

4.1 We index 426GB using Wumpus with 1GB memory. The x axis refers to the time instances at which memory contents are flushed to disk. (a) HSM maintains up to 2 merge-based runs on disk, and (b) HLM periodically merges the runs created on disk so that their number is logarithmic in the current size of the on-disk index. 43

4.2 We break down the index building time into document parsing and postings flushing parts across different maintenance policies. Parsing includes the time required to clean dirty pages from page cache to free space for newly read documents. Proteus parsing performance is pessimistic as it uses an unoptimized implementation (Section 5.1). We also include the number of merge-based runs each method maintains. SRF has lower time than HIM and HSM, and only 12% higher build time than HLM, even though it maintains contiguously all lists on disk. 44

4.3 (a) The prototype implementation of *Proteus*. (b) We maintain the hashtable in memory to keep track of the postings that we have not yet flushed to disk. 49

4.4 (a) Each entry of the rangetable corresponds to a term range, and points to the search bucket, which serves as partial index of the corresponding rangeblock. (b) Each entry of the termtable corresponds to a term and points to the blocklist that keeps track of the associated termblocks on disk. 50

5.1 We consider the index building time for different indexing methods across Wumpus and Proteus, both with full stemming. Over Wumpus, we examine Nomerge (Nomerge$_W$), Hybrid Logarithmic Merge (HLM$_W$), Hybrid Square Root Merge (HSM$_W$) and Hybrid Immediate Merge (HIM$_W$). Over Proteus, we include Hybrid Immediate Merge (HIM$_P$), Selective Range Flush (SRF$_P$) and Unified Range Flush (URF$_P$). URF$_P$ takes 421min to process the 426GB of GOV2 achieving roughly 1GB/min indexing throughput (see also Figure 5.7 for other datasets). 55

5.2 We consider Hybrid Immediate Merge over Wumpus (HIM_W) or Proteus (HIM_P), along with Selective Range Flush (SRF_P) and Unified Range Flush (URF_P) over Proteus. (a) We measure the average query time with alternatively disabled and enabled the system buffer cache across different queries in the two systems with full stemming. (b) We look at the distribution of query time over the two systems with enabled the buffer cache. 58

5.3 (a) Setting the rangeblock size B_r below 32MB or above 64MB raises the build time of Unified Range Flush. Increasing the B_r tends to (b) decrease the number of flushes, and (c) increase the data amount transferred during merges. We use Proteus with light stemming. 59

5.4 (a) Flushing more than few tens of megabytes (M_f) leads to longer build time for Unified Range Flush (URF). This results from the more intense I/O activity across term and range flushes. (b) Setting the append threshold to T_a = 256KB minimizes the total I/O time of range and term flushes. (c) The build time of range merge in URF decreases approximately in proportion to the increasing size of posting memory (M_p). The Proteus system with light stemming is used. 60

5.5 We examine the behavior of Unified Range Flush over Proteus with the following storage allocation methods (i) contiguous (CNT), (ii) doubling (DBL), and (iii) fragmented (FRG) with termblock sizes 1MB, 2MB, 8MB and 32MB. (a) CNT achieves the lowest query time on average closely followed by DBL. We keep enabled the system buffer cache across the different queries. (b) Build time across the different allocation methods varies within 5.7% of 386min (FRG/1MB and DBL). (c) Unlike CNT and DBL, FRG tends to increase the index size especially for larger termblock. 62

5.6 We consider three methods of memory allocation during index building by Unified Range Flush: (i) default (D), (ii) single-call (S), and (iii) chunkstack (C). The sensitivity of build time to memory management is higher (up to 8.6% decrease with C) for larger values of M_p. We use Proteus with light stemming. 63

5.7 We show the scaling of build time with Selective Range Flush (SRF) and Unified Range Flush (URF). We use the ClueWeb09 (first TB), GOV2 (426GB) and Wikipedia (200GB) datasets over Proteus with light stemming. URF takes 53.5min (7%) less time for ClueWeb09, about the same for Wikipedia, and 16.4min (4%) more for GOV2 in comparison to SRF. . 65

6.1 The query latency at the Cassandra client varies according to a quasi-periodic pattern. The total throughput of queries and inserts also varies significantly. 70
6.2 Assumed datastore architecture. 72
6.3 The organization of the Rangetable structure, and control flow of a handled range query. For presentation clarity we use alphabetic characters as item keys. 74
6.4 Prototype framework with several compaction methods as plugins. 77
6.5 We observe similar compaction activity between Cassandra and our prototype implementation of SMA (k=4). The height (y-axis value) of each mark denotes the transfer size of the respective compaction. 78

7.1 During concurrent inserts and queries, (a) the get latency of Geometric (r=2) and SMA (k=4) has substantially higher variability and average value than Rangemerge, and (b) the get throughput of Geometric (r=2) drops as low as 15.5req/s during compactions (grey background). 83
7.2 (a) At the insertion of 10GB with M=512MB using Geometric partitioning (r=2), get latency (at load 10req/s) is closely correlated to the number of files created. (b) We show the number of files maintained per key range for six methods. 84
7.3 I/O intensity of compactions. The disk traffic of compactions in Rangemerge is comparable to that of Nomerge with M=512MB. 85
7.4 Scaling configuration parameters. The insertion progress is similar between the configuration of M=256MB with 5GB dataset (left y-axis) and M=2GB with 40GB (right y-axis) for Geometric (r=2), SMA (k=4) and Rangemerge. 86

7.5 (a) The insertion time (log y axis) of Rangemerge is about half the insertion time of Remerge and closely tracks that of Geometric (p=2). (b) With M=4GB and 80GB dataset size Rangemerge has lower insertion time than Geometric (p=2) and (r=3) while storing each key at a single disk location. 87
7.6 Performance sensitivity to put load assuming concurrent get requests at rate 20req/s and scan size 10. 88
7.7 Sensitivity to range get size assuming concurrent load of 2500req/s put rate and 20req/s get rate. 88
7.8 Sensitivity of insertion time to get rate of scan size 10 with concurrent put rate set at 2500req/s. 89
7.9 Impact of M to insertion time. With M=2GB, Rangemerge approaches Nomerge and stays by at least 21% below the other methods. 89
7.10 Sensitivity of insertion time to key distribution, as we generate put requests back-to-back with zero get load. 90
7.11 (a) Over an SSD, the insertion time of Rangemerge lies halfway between that of Nomerge and Remerge. (b) Rangemerge reduces the variability of get latency in comparison to SMA ($k = 4$) and Geometric ($r = 2$). 91

8.1 Files are hierarchically organized in LevelDB. When memtable is full, it is flushed into an SSTable at level 0. Thus, level-0 files may contain overlapping ranges of keys. When the size of a level L exceeds its threshold, a single file from level L (or all level files, if $L = 0$) along with all overlapping files from level $L + 1$ are merged and stored as a number of 2MB files at level $L + 1$. The maximum size of a level is expressed either as maximum number of files ($L = 0$) or as maximum total size ($L > 0$). 98
8.2 Rangemerge logging in LevelDB. 100
8.3 Various merging strategies, as we implemented them in LevelDB. 102
8.4 (a) We show the total disk space consumed by log files in our Rangemerge implementation within LevelDB. Log size is at least equal to the memory size M, and normally between $2M$ and $3M$. (b) There is a small overhead involved in tracking the log files referenced by each range and deleting the unreferenced ones. 104

8.5 Comparison of the insertion time of various methods implemented in LevelDB and in our prototype system. 105
8.6 Get latency in various compaction methods implemented in LevelDB, assuming a concurrent load of 2500put/s and 20get/s of scan size 10. Background compactions (gray background) severely affect queries in all methods except for Rangemerge. 106
8.7 Get throughput in various compaction methods implemented in LevelDB, assuming a concurrent load of 2500put/s and 20get/s of scan size 10. Rangemerge manages to keep the rate at which queries are served above 15req/s. In all remaining methods the get throughput is seriously affected during the background compactions (gray background). 107
8.8 Get latency (above) and throughput (below) for point queries in three methods, assuming puts at 2500req/s and point gets at 20req/s. 108

List of Tables

2.1 Summary of storage structures typically used in datastores. We include their I/O complexities for insertion and range query in one-dimensional search over single-key items. 25

3.1 Summary of the asymptotic cost (in I/O operations) required to incrementally build inverted files and retrieve terms for query handling. N is the number of indexed postings and M is the amount of memory used for postings gathering. The parameter a (e.g., $a = 1.2$) refers to the Zipfian distribution (Section 9.1). 31

3.2 Main functional differences among existing and our new methods of incremental text indexing. 33

3.3 Average search latency (ms) and the fraction of it spent on I/O, using the GOV2 dataset over the Zettair search engine. 35

3.4 Average, median and 99th percentile of search latency (ms) when different numbers of stop words are applied with and without page caching in GOV2/Zettair. 35

4.1 Sensitivity to interactions between rangeblock size B_r and preference factor F_p. We underline the lowest measurement on each row. The highest measured time is 62.18min, i.e., 53.8% higher than the lowest 40.43min. . 45

4.2 Parameters of Selective Range Flush (SRF) and Unified Range Flush (URF). In the last column we include their default values used in our prototype. . 47

5.1 We examine the effect of alternative optimizations to the query and build time of Unified Range Flush. Preallocation reduces the average query time, while prefetching and chunkstack reduce the build time. 64

6.1 Storage management on the server occupies more than 80% of the average query latency measured at the client. 71

7.1 Amount of flushed and totally transferred data per compaction, delay per compaction, and total insertion time for different rangefile sizes of Rangemerge. 93

List of Algorithms

4.1 Pseudocode of Selective Range Flush 41
4.2 Pseudocode of Unified Range Flush 48
6.3 Pseudocode of Rangemerge 76

Glossary

HIM	Hybrid Immediate Merge
HLM	Hybrid Logarithmic Merge
HSM	Hybrid Square Root Merge
SRF	Selective Range Flush
SMA	Stepped Merge Array
URF	Unified Range Flush

B_r	Rangeblock
B_t	Termblock
F_p	Preference Factor
M_f	Flush Memory
M_p	Posting Memory
T_a	Append Threshold
T_t	Term Threshold

Abstract

Margaritis, Giorgos, D.

Phd, Department of Computer Science and Engineering, University of Ioannina, Greece.

April, 2014.

Efficient Storage Indexing of Structured and Unstructured Data.

Thesis Supervisor: Stergios V. Anastasiadis.

Commercial and public organizations currently strive to manage massive amounts of structured and unstructured data in all fields of society. The data collected across different local and online services, such as news websites, social media, mail servers and file systems, is inherently semi-structured or unstructured. Therefore, effective text indexing and search is crucial for data usability and exploration. Moreover, the exploding amount of structured data that needs to be managed and the demanding workloads that include both throughput-oriented batch jobs and latency-sensitive data serving drive the development of horizontally-expandable, distributed storage systems, called scalable datastores. In this thesis, we study the analysis, design, and implementation of storage systems to efficiently store, access, and search both structured and unstructured data.

Real-time text search requires to incrementally ingest content updates and make them searchable almost immediately, but also serve search queries at low latency. Recent methods for incremental index maintenance substantially increase search latency with the index fragmented across multiple disk locations. For the support of fast indexing and search over disk-based storage, we introduce a method called Selective Range Flush (SRF). We organize the disk index over blocks, which allow to selectively update only the parts of the index that can be efficiently updated based on SRF. We show that SRF reduces the indexing time, but requires substantial experimental effort to tune specific parameters for performance efficiency. Subsequently, we propose the Unified Range Flush (URF) method,

which is conceptually simpler than SRF, achieves similar or better performance with fewer parameters and less tuning, and is amenable to I/O complexity analysis. We implement the two methods in the Zettair open-source search engine, using carefully optimized storage and memory management. Then, we do extensive experiments with three different web datasets of size up to 1TB. Across different open-source systems, we show that our methods offer search latency that matches or reduces up to half the lowest achieved by existing disk-based methods. In comparison to an existing method of comparable search latency on the same system, our methods reduce by a factor of 2.0–2.4 the I/O part of build time, and by 21–24% the total build time.

Scalable datastores are required to manage enormous amounts of structured data for online serving and analytics applications. Across different workloads, they weaken the relational and transactional assumptions of traditional databases to achieve horizontal scalability and availability, and meet demanding throughput and latency requirements. Efficiency tradeoffs at each storage server often lead to design decisions that sacrifice query responsiveness for higher insertion throughput. In order to address this limitation, we introduce the Rangetable storage structure and Rangemerge method so that we efficiently manage structured data in granularity of key ranges. We develop both a general prototype framework and a storage system based on Google's LevelDB open-source key-value store. In these two platforms, we implement several representative methods as plugins to experimentally evaluate their performance under common operating conditions. We conclude that our approach incurs range-query latency that is minimal and has low sensitivity to concurrent insertions, while it achieves insertion performance that approximates that of write-optimized methods under modest query load. Our method also reduces down to half the reserved disk space, improves the write throughput proportionally to the available main memory, and naturally exploits the key skewness of the inserted dataset.

Εκτεταμενη Περιληψη στα Ελληνικα

Γεώργιος Μαργαρίτης του Δημητρίου και της Παναγιώτας.
PhD, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Απρίλιος, 2014.
Αποδοτική Δεικτοδότηση Αποθήκευσης για Δομημένα και Αδόμητα Δεδομένα.
Επιβλέποντας: Στέργιος Β. Αναστασιάδης.

Δημόσιοι οργανισμοί και ιδιωτικές επιχειρήσεις αντιμετωπίζουν σήμερα το πρόβλημα της διαχείρισης μεγάλου όγκου δομημένων και αδόμητων δεδομένων. Τα δεδομένα αυτά συχνά συλλέγονται από ένα πλήθος τοπικών υπηρεσιών ή υπηρεσιών του διαδικτύου, όπως τα συστήματα αρχείων, οι ιστοσελίδες ενημέρωσης, τα κοινωνικά δίκτυα και οι διακομιστές ηλεκτρονικού ταχυδρομείου, και είναι εγγενώς ημιδομημένα ή αδόμητα. Για το λόγο αυτό, η αποτελεσματική δεικτοδότηση και αναζήτηση κειμένου είναι μία εξαιρετικά σημαντική υπηρεσία για την αξιοποίηση και χρήση των δεδομένων αυτών. Επιπρόσθετα, το συνεχώς αυξανόμενο μέγεθος των δομημένων δεδομένων που πρέπει να διαχειριστούν, καθώς και ο υψηλός αλλά και ποικιλόμορφος φόρτος εργασίας, έχουν οδηγήσει στην ανάπτυξη οριζόντια-επεκτάσιμων κατανεμημένων συστημάτων τα οποία καλούνται κλιμακώσιμα συστήματα αποθήκευσης. Στη διατριβή αυτή μελετούμε την ανάλυση, το σχεδιασμό και την υλοποίηση αποδοτικών συστημάτων αποθήκευσης και αναζήτησης για δομημένα και αδόμητα δεδομένα.

Η αναζήτηση κειμένου σε πραγματικό χρόνο προϋποθέτει τη δυνατότητα συνεχούς εισαγωγής νέων ενημερώσεων στο σύστημα και την σχεδόν άμεση διάθεσή τους προς αναζήτηση, όπως επίσης και την εξυπηρέτηση ερωτημάτων αναζήτησης με χαμηλή καθυστέρηση. Πρόσφατες μέθοδοι για την αυξητική ενημέρωση του ευρετηρίου αναζήτησης κατακερματίζουν το ευρετήριο στο δίσκο, με αποτέλεσμα τη σημαντική αύξηση των χρόνων αναζήτησης. Έχοντας ως στόχο την υποστήριξη γρήγορης δεικτοδότησης και αναζήτησης, προτείνουμε τη μέθοδο Selective Range Flush (SRF). Επιλέγουμε να οργανώσουμε το ευρετήριο στο δίσκο σε μπλοκ, το οποίο επιτρέπει την επιλεκτική ενημέρωση μόνο των

τμημάτων του ευρετηρίου που μπορούν να ενημερωθούν αποδοτικά βάσει του αλγορίθμου SRF. Δείχνουμε πως ο SRF πετυχαίνει μείωση του χρόνου δεικτοδότησης, όμως απαιτεί σημαντική πειραματική προσπάθεια για την αποτελεσματική παραμετροποίηση του. Στη συνέχεια προτείνουμε τον αλγόριθμο Unified Range Flush (URF), ο οποίος είναι κατά βάση απλούστερος από τον SRF, πετυχαίνει παρόμοια ή και καλύτερη απόδοση με λιγότερες παραμέτρους και ευκολότερη ρύθμισή τους, ενώ επιτρέπει τη μελέτη της ασυμπτωτικής του πολυπλοκότητας. Αναπτύσσουμε τις δύο προτεινόμενες μεθόδους στη μηχανή αναζήτησης ανοιχτού κώδικα Zettair, χρησιμοποιώντας προσεκτικά υλοποιημένα υποσυστήματα διαχείρισης μνήμης και δίσκου. Έπειτα, εκτελούμε εκτεταμένα πειράματα με τρεις διαφορετικές συλλογές δεδομένων μεγέθους μέχρι 1TB. Μεταξύ διαφορετικών συστημάτων ανοιχτού κώδικα, δείχνουμε ότι οι μέθοδοί μας παρέχουν καθυστέρηση αναζήτησης που είναι παρόμοια ή μειωμένη έως και 50% σε σχέση με τις χαμηλότερες καθυστερήσεις που πετυχαίνουν υπάρχουσες μέθοδοι. Συγκριτικά με μία μέθοδο αντίστοιχης καθυστέρησης αναζήτησης, οι μέθοδοί μας μειώνουν κατά έναν παράγοντα 2.0–2.4 το κομμάτι του χρόνου δεικτοδότησης που αφορά την E/E, και κατά 21%–24% το συνολικό χρόνο δεικτοδότησης.

Τα κλιμακώσιμα συστήματα αποθήκευσης είναι σήμερα απαραίτητα για τη διαχείριση του τεράστιου όγκου δομημένων δεδομένων που απαιτούν οι υπηρεσίες διαδικτύου και οι διάφορες εφαρμογές ανάλυσης δεδομένων. Με σκοπό την επίτευξη οριζόντιας κλιμακωσιμότητας και διαθεσιμότητας, καθώς και την εξυπηρέτηση αιτημάτων με υψηλή ρυθμαπόδοση και χαμηλή καθυστέρηση, τα συστήματα αυτά δεν υιοθετούν το σχεσιακό μοντέλο και τις ACID ιδιότητες που παρέχουν οι παραδοσιακές βάσεις δεδομένων. Έχοντας ως κύριο στόχο την παροχή υψηλής απόδοσης αποθήκευσης εγγραφών, τα συστήματα αυτά συνήθως επιλέγουν να θυσιάσουν την απόδοση ανάγνωσης εγγραφών. Για να αντιμετωπίσουμε τον περιορισμό αυτό προτείνουμε την δομή αποθήκευσης Rangetable και τη μέθοδο Range-merge, βάσει των οποίων η διαχείριση των εγγραφών γίνεται αποδοτικά ομαδοποιώντας τις σε λεξικογραφικά εύρη. Αναπτύσσουμε τόσο μία γενική πρότυπη πλατφόρμα αποθήκευσης όσο και ένα αποθηκευτικό σύστημα βασισμένο στο LevelDB, ένα ανοιχτού κώδικα σύστημα διαχείρισης κλειδιού-τιμής από τη Google. Υλοποιούμε ένα πλήθος από αντιπροσωπευτικές μεθόδους στα δύο αυτά συστήματα και μελετούμε πειραματικά την απόδοσή τους. Δείχνουμε πως η απόδοση της προσέγγισής μας επιτυγχάνει καθυστέρηση απάντησης σε ερωτήματα εύρους (range-queries) που είναι ελάχιστη και έχει χαμηλή ευαισθησία σε ταυτόχρονες εισαγωγές δεδομένων. Παράλληλα, η απόδοση εγγραφής της μεθόδου μας προσεγγίζει

αυτές των μεθόδων που είναι σχεδιασμένες για υψηλή απόδοση εγγραφής όταν ταυτόχρονα εξυπηρετούνται και αιτήματα ανάγνωσης. Τέλος, η μέθοδός μας μειώνει στο μισό το δεσμευμένο αποθηκευτικό χώρο, βελτιώνει την ρυθμαπόδοση εισαγωγής δεδομένων αναλογικά με τη διαθέσιμη μνήμη του συστήματος, ενώ εκμεταλλεύεται την ασυμμετρία της κατανομής των κλειδιών που εισάγονται.

Chapter 1

Introduction

1.1 Motivation

1.2 Text Search

1.3 Scalable Datastores

1.4 Thesis Contribution

1.5 Thesis Organization

1.1 Motivation

We live in the era of *big data*, where commercial and public organizations strive to manage massive amounts of both structured and unstructured data in all fields of society. Even though there is no clear definition of big data, it is usually characterized by the three following properties: high volume, high velocity, and high variety [117]. The data collected across different local and online services, such as news websites, social media, mail servers and file systems, is inherently semi-structured or unstructured (high variety). Therefore, effective *text indexing and search* is very important for data usability and exploration. Moreover, the exploding amount of structured data that needs to be managed (high volume) and the demanding workloads that include both throughput-oriented batch jobs

and latency-sensitive data serving (high velocity) drive the development of horizontally-expandable, distributed storage systems, called *scalable datastores*. Not surprisingly, most major web companies such as Google, Yahoo!, Facebook and Microsoft deal with both the problems of text search and data management at large scale, and have developed their own indexing and storage systems to meet the requirements of their workloads.

In this thesis, we study the problem of big data management from the aspect of designing and implementing systems to efficiently store, access and search both structured and unstructured data at large scale. Although quite different in principal, the problems of scalable text search and storage management share some fundamental characteristics:

- *System architecture.* The distributed systems designed for large-scale indexing or storage usually follow a *two-tier, shared-nothing* architecture, where a number of *front-end* servers receive end-user requests and forward them to a number of *worker* servers. Front-end and worker servers may be separate physical nodes or hosted on the same machine.

- *Horizontal partitioning.* In these systems, scalability is usually achieved using a technique called *horizontal partitioning* (or *sharding*). The data is horizontally partitioned into a number of *disjoint partitions*, which are subsequently assigned to worker servers. For example, a large document collection may be partitioned into disjoint sets of documents for text indexing, while a large table may be split into groups of consecutive rows in case of datastores. Each worker then locally stores and indexes the partition it has been assigned and serves requests for it.

- *Data type and ingestion workflow.* The ingestion of new data at each worker usually follows the approach of accumulating in main memory items in the form of $\langle key, value \rangle$ pairs, until the memory is exhausted. When this happens, all memory items are flushed to disk and merged with the existing disk items. In the specific case of text indexing the *key* is a term that appeared in the document collection and the *value* is a list of documents it appeared into. Similarly, items are explicitly inserted as $\langle key, value \rangle$ pairs in datastores.

- *Storage fragmentation as read-write tradeoff.* The contiguous disk storage of items at the worker servers is critical for low read latency. Nevertheless, the majority of

existing storage management methods at the workers keep the items fragmented on disk to improve write or indexing throughput, at the cost of reduced read or search performance.

In spite of these similarities, text indexing and storage management of datastores do have some important differences:

- *Data preprocessing.* Text indexing handles text documents, which must first be parsed into ⟨*document-id, term*⟩ tuples before being accumulated in memory and grouped into ⟨*term, list-of-document-ids*⟩ key-value pairs. On the other hand, items inserted in datastores typically do not need any kind of preprocessing.

- *Data update.* During text indexing, when a new ⟨*document-id, term*⟩ pair is inserted in memory, the *document-id* must be appended to the list of documents for *term*. In contrast, a new ⟨*key, value*⟩ pair inserted in a datastore will replace any existing value for *key* (or create a new version of it).

- *Item size distribution.* The sizes of the individual items processed in text indexing approximately follow a Zipfian distribution: a few popular terms (e.g., "the", "of", "and") may have document lists of several tens or hundreds of MB in size, while the vast majority of terms appear infrequently in documents and have list sizes of a few bytes. Datastores on the other hand handle items of similar sizes, typically in the range of a few tens or hundreds of KB.

- *Workload types.* New documents are sent periodically to worker servers for indexing (for example, after a web crawler has fetched a batch of web pages). This means that most of the time the workers serve search queries, and only occasionally need to index a collection of new documents. Unlike text search servers, datastores constantly serve both reads and writes. In fact, due to the nature of web applications, datastores frequently experience write-intensive workloads although read-heavy workloads are also common.

In general, horizontal scalability enables these distributed systems to increase their capacity by simply adding more servers. Additionally, to a large extent the system performance is determined by the performance of the constituent worker servers. We therefore

focus on the efficiency of storing and serving the assigned partitions at each worker server. Recent methods that manage the disk and memory on the workers usually follow a write-optimized approach. As a consequence, reads are considerably affected with respect to the latency and the rate they are served. We thus shift our focus on improving the storage layer on the workers by designing, analyzing and implementing efficient methods for the management of items in memory and on disk. Our aim is to improve the read and search performance, while maintaining high the write and indexing throughput.

1.2 Text Search

Real-time text search requires to incrementally ingest content updates and almost immediately make them searchable, while serving search queries at low latency. To answer a text query, a search engine must first process a text dataset and create for each term that appears in the dataset an *inverted list* with pointers (postings) to all its occurrences. The set of the inverted lists make up the *inverted index* of the dataset. As new documents are added to the collection, inverted lists are accordingly updated by adding new postings to them. To evaluate a text query, a search engine typically fetches in memory the inverted list of each query term and combines them to calculate the set of documents that are relevant to the query (e.g., contain all query terms). Given that a substantial time fraction of query handling is spent on fetching the lists from disk, list contiguity is considered extremely important for fast query evaluation. Recent methods for incremental index maintenance improve indexing cost by relaxing the list contiguity requirement, but substantially increase search latency due to the storage fragmentation of lists.

For the support of fast search over disk-based storage, we take a fresh look at incremental text indexing in the context of current architectural features. We advocate to preserve the list contiguity but lower the indexing time, considering efficient algorithms and data structures, as well as carefully optimized storage-level and memory management implementations. To this end, we introduce a method called Selective Range Flush (SRF) to contiguously organize the index over disk blocks and dynamically update it at low cost. Block-based management simplifies the maintenance of the inverted index because it allows us to selectively update only the parts of the index that can be efficiently updated.

We show that SRF reduces the indexing time, but requires substantial experimental effort to tune specific parameters for performance efficiency. Subsequently, we propose the Unified Range Flush (URF) method, which is conceptually simpler than SRF, achieves similar or better performance with fewer parameters and less tuning, and is amenable to I/O complexity analysis. We implement interesting variations of the two methods in a prototype we developed using the Zettair open-source search engine, and do extensive experiments with three different web datasets of size up to 1TB. Across different systems, we show that our methods offer search latency that matches or reduces up to half the lowest achieved by existing disk-based methods. In comparison to an existing method of comparable search latency on the same system, our methods reduce by a factor of 2.0-2.4 the I/O part of build time, and by 21-24% the total build time.

1.3 Scalable Datastores

To meet the needs of write-heavy workloads that often emerge from applications that constantly create large amounts of data, a number of scalable datastores adopt an append-only, write-optimized storage layer [33, 15]. The majority of datastores, including the proprietary storage platforms of Google, Microsoft and Facebook [28, 25, 59] and their popular open-source alternatives [42, 81, 53], manage the data stored on disk using an approach similar to the Log-Structured Merge tree (LSM-tree) [79]. Using an LSM-tree, incoming updates in the form of key-value pairs are simply appended to a log file on disk and accumulated in memory, before control is returned to the client. When incoming data fills up the available memory, all memory entries are flushed to disk in a immutable, sorted file. Reads may need to merge entries from multiple disk files, so files are periodically merged in the background according to specific merge patterns. These merges (or *compactions*) can be performed efficiently since files are sorted. Nevertheless, they interfere with concurrent queries leading to latency spikes and throughput decrease, and they can last from several minutes to hours. Additionally, they require half of the available disk capacity to be reserved for the creation of new files. Deferring these compactions is not a viable solution, because deferred compactions would leave the data fragmented on disk for extended periods leading to low query performance.

To address all the above issues, the memory and storage management of write-optimized datastores should be reconsidered. The problem lies in the way the LSM-tree amortizes the cost of writes, deferring the flush of memory entries to disk files until memory is full —in which case all memory entries are written to a disk file— and occasionally merging the files produced. We modify the fundamental structure of data storage: instead of periodically performing a few intensive compactions that cause performance drop, we propose the use of smaller, more frequent, less aggressive but still efficient compactions. Our main insight is to keep the data in memory and disk sorted and partitioned across disjoint key ranges, and store each key range in a separate file. When memory is exhausted, we only flush to disk the range that occupies the larger part of memory space and merge it with its disk file. A range is split when needed to keep bounded the size of its file and the respective merge cost.

We develop both a general prototype framework and a storage system based on Google's LevelDB open-source key-value store. We show that the proposed method effectively reduces the variation in query latency caused by background flushes and compactions, while it minimizes the query latency by keeping each entry contiguously stored on disk. At the same time, the write performance achieved approximates or even beats those of other write-optimized stores under various moderate conditions. Our method also removes the need for excessive storage reservation, improves the ingest throughput proportionally to the increase of the main memory, and naturally exploits the key skewness of the inserted dataset.

1.4 Thesis Contribution

The work performed within this thesis contributes to two areas of computer science: text retrieval and storage management for structured data. The main goals of this thesis are as follows:

- to propose incremental text indexing methods and implement a prototype search engine that can serve search queries with low latency and achieve high indexing throughput;

- to design and develop an efficient storage layer for the nodes of scalable datastores in order to store and access items fast and keep the interference between ingesting and serving data low.

The scientific methodology for validating the proposed thesis includes:

- design of efficient methods;
- development of fully functional prototype systems;
- implementation of the proposed methods and related methods from the literature in the same prototype system for fair comparison;
- implementation of the methods evaluated in production systems to examine the applicability of our algorithms and data structures and the generality of our results;
- experimental evaluation using both real-world datasets and synthetic workloads;
- theoretical analysis of our methods.

The most important contributions of this thesis are the following:

- We introduce efficient text indexing methods to incrementally update the index on disk, and describe memory and disk management optimizations that further improve the indexing performance.
- We evaluate the proposed solution on a state-of-the-art open-source search engine using three different real-world web datasets. We demonstrate the feasibility of building text search engines that can preserve index contiguity on disk for fast disk-based search while maintaining high indexing throughput.
- We provide a unified consideration of known solutions for datastore storage management across different research fields. We identify several limitations in existing systems, which stem from the fundamental way memory and storage are managed in most write-optimized systems. We introduce a new storage structure and a new data management method to address them.

- We implement the proposed storage management approach in both a general prototype framework and a production storage system, and conduct extensive experimental evaluation using large synthetic workloads. We show that our method achieves minimal range-query latency of low sensitivity to concurrent inserts, has write performance that approximates or even beats those of other write-optimized methods, and reduces down to half the required reserved disk space.

- We perform asymptotic analysis for the data ingestion I/O cost of our methods. The proposed methods are theoretically shown to have similar asymptotic behavior to some existing methods, but are experimentally demonstrated to have superior performance.

1.5 Thesis Organization

The structure of the rest of the thesis is organized as follows:

In **Chapter 2**, we provide the background required to understand the problems of full-text search and large-scale data management. We then proceed to an overview of the related methods for incremental index maintenance and include a brief description of the architectures of the most important scalable datastores.

In **Chapter 3**, we define the problem of incremental text indexing, review and compare previous related research, and motivate our work by experimentally showing the problems caused by the storage fragmentation of index.

In **Chapter 4**, we introduce two new methods to efficiently manage the index on disk, describe the design and architecture of our prototype search engine, and provide details about our implementation.

In **Chapter 5**, we specify the characteristics of our experimentation platform, compare the index build and search performance across a representative collection of methods using three different datasets over two different systems, and evaluate the effect that important parameters and engineering optimizations have on the performance of the system.

In **Chapter 6**, we present the problem of large-scale storage management and experimentally motivate our work. We then introduce a new method and describe the

architecture of our prototype storage framework.

In **Chapter 7**, we evaluate the performance of queries and insertions, as well as their interference, across several methods implemented in our storage system. We also examine the performance sensitivity to various workload parameters and storage devices, and discuss about various issues and limitations of our design.

In **Chapter 8**, we describe the implementation of our storage management method in a production system. We provide details about the logging and recovery components, and evaluate the efficiency of these mechanisms. We then compare our method to alternative methods in terms of data ingestion throughput.

In **Chapter 9**, we analyze the asymptotic behavior of our methods by performing complexity analysis of their I/O cost.

In **Chapter 10**, we provide an overall review of the results of our research, summarize the basic conclusions, and indicate open issues and interesting directions for future work.

Chapter 2

Background and Related Research

2.1 Full-Text Search

2.2 Large-Scale Data Management

2.1 Full-Text Search

Full-text search refers to the set of algorithms and data structures that enable a user to search for a specific document in a text database. Today, it is an indispensable service for the automated retrieval of text documents, whether proprietary within an organization, or public across the web. A document may be a text file stored in the local file system, a page on the web, or a status update from a social network. The user submits a query into the system which is typically a set of words describing the contents of the document, and the response is a list of documents, each probably with different degree of relevance to the query. Full-text search is distinguished from search based on metadata, such as document title, author or date of publication.

When the text database consists of a relatively small number of documents, the search engine can directly scan the contents of the documents to find those relevant to each query. However, when dealing with a large number of documents or search queries, the problem of full-text search is usually decomposed into two stages: indexing and searching. The *indexing* stage takes as input the document collection and builds an index, which is then

used in the *searching* stage to evaluate the search queries. In the next sections, we provide background information and related literature about index build and query evaluation in full-text search systems.

2.1.1 Preliminaries

The most efficient index structure for text query evaluation is the *inverted file* [118]. An inverted file is an index that for each term stores a list of pointers to all documents that contain the term. Each pointer to a document is usually called *posting* and each list of postings for a particular term is called *inverted list*. In a *word-level* inverted list a posting specifies the exact position where a term occurs in the document, unlike a *document-level* inverted list that only indicates the appearance of a term in a document. The *lexicon* (or *vocabulary*) of the inverted file associates every term that appeared in the dataset to its inverted list. For each term t it stores a count f_t of the documents containing t and a pointer to the start of the corresponding inverted list on disk.

In a word-level index the inverted list for a term t contains pointers of the form:

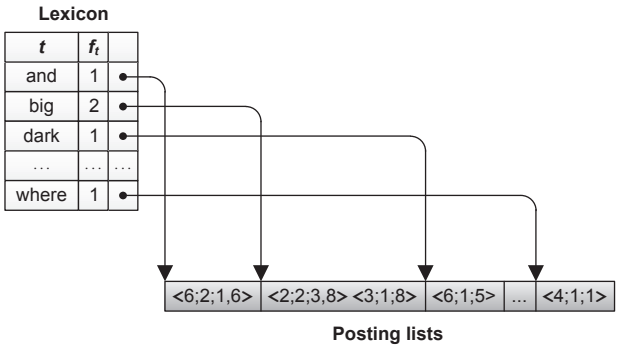
$$\langle d;\ f_{d,t};\ p_1,\ p_2,\ ...,\ p_{f_{d,t}} \rangle$$

where d is a document identifier, $f_{d,t}$ is the number of occurrences of t in d, and $p_1, ..., p_{f_{d,t}}$ are the positions within d where t appears. Word positions are valuable in text search because they are used to identify the adjacency or proximity of terms, e.g., in phrase queries [3, 112, 16, 118]. A simple text collection along with the corresponding word-level inverted file is illustrated in Figure 2.1.

Inverted List Organization. Modern search engines typically keep their inverted lists compressed on disk in order to reduce the space occupied by the inverted index and the time required for query evaluation. Index compression adds extra computation cost, but the gain of reduced data traffic to and from disk is relatively higher [118, 64]. Each new document added to the collection is assigned a monotonically increasing identifier. Thus, an inverted list consists of document identifiers sorted in increasing order (*document-ordered*) and can be represented as a sequence of differences between successive document identifiers (*d-gaps*). For example, an inverted list containing the documents $\langle 3, 5, 20, 21, 23, 76, 77, 78 \rangle$ can be represented as $\langle 3, 2, 15, 1, 2, 53, 1, 1 \rangle$. The differences are

d	Document contents
1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light

(a) Document collection



(b) Inverted file

Figure 2.1: (a) A simple text collection of six documents. (b) The lexicon and the inverted lists for the specific document collection.

usually smaller than the initial identifiers and can be efficiently encoded using an integer coding scheme [118]. The same technique can also be used to compress the sorted sequence of term occurrences within each document.

Document-ordered inverted lists are widely used for incremental index maintenance because they are updated simply by appending new postings at their end [63]. Depending on the query type and the system performance, query evaluation may require to retrieve in memory the entire document-ordered inverted list of each query term [64, 116, 50]. Alternatively, an inverted list can be sorted according to decreasing frequency (*frequency-ordered*) of term occurrence in a document or decreasing contribution (*impact-ordered*) to the query-document similarity score [118]. Such organizations allow inverted lists to be retrieved in blocks rather than in their entirety, which makes their contiguous storage relevant for the individual blocks. However, in comparison to a document-ordered list, the alternative organizations require additional cost (e.g., for I/O) to handle complex queries (e.g., term-proximity or Boolean queries) [118, 116]. Furthermore, a list update cannot be performed efficiently as it involves partial list reorganization with additional encoding or decoding, and thus these schemes are not usually used in incremental index maintenance [100].

Query Evaluation. A query to a text search engine is usually a list of terms (also called *bag-of-words* query), probably along with some constraints such as Boolean operators. The first step in evaluating a query is finding all documents that contain some

or all of the query terms and satisfy the constraints. Each document is then assigned a similarity score that denotes the "closeness" of the document to the textual query [118]. The underlying principle is that the higher the similarity score awarded to a document, the greater the estimated likelihood that the user would consider it relevant to his or her query. Finally, the documents are ranked based on their scores and the k highest-ranked documents are returned to the user.

The similarity of the indexed documents to a query can be calculated by evaluating the contribution of each query term to all document scores (*term-at-a-time*), all query terms to a single document score (*document-at-a-time*), or the postings with highest impact to document scores (*score-at-a-time*) [2]. Traditionally, document-at-a-time evaluation is commonly used in web search because it more efficiently handles context-sensitive queries for which the relation (e.g., proximity) among terms is crucial [18]. Given that a high percentage of users only examine a few tens of relevant documents, search engines may prune their index to compute fast the first batches of results for popular documents and keywords. Thus, a two-tier search architecture directs all incoming queries to a first-tier pruned index, but directs to a second-tier full index the queries not sufficiently handled by the first tier [78]. In order to overcome the bottleneck of disk-based storage, pruning of an impact-sorted index allows inverted lists to be stored in memory for significantly improved performance of score-at-a-time query evaluation [100].

Index Build. Published literature on text indexing separates *offline* index construction from *online* index maintenance [118, 89]. Offline indexing deals with handling static document collections. In order to index static datasets, the system needs to parse documents into postings maintained in memory and periodically flush the accumulated postings to disk creating a new partial index. Eventually, external sorting can be used to merge the multiple index files into a single file that handles queries for the entire dataset. During the indexing process the queries are handled using an older index.

Online indexing on the other hand handles dynamic document collections. Documents may be added to or deleted from the dataset at any time, and the index should reflect these changes. The system must be able to process queries during index updates, and the query results should include newly added documents or exclude any documents deleted. In comparison to online maintenance, offline index construction is simpler because it does not handle document queries until its completion, and has been addressed in the past

using efficient methods [51].

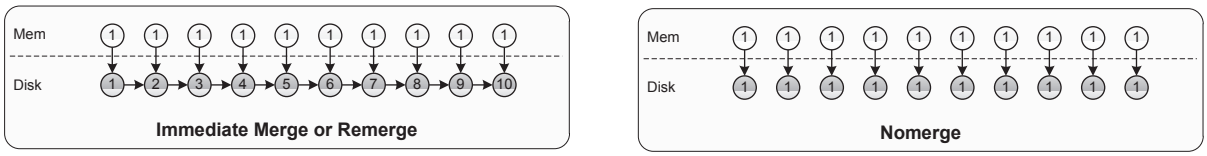
In the rest of this section we focus on dynamic datasets that allow insertions of new documents over time and examine online indexing methods that maintain inverted files efficiently on secondary storage. Index maintenance for the more general case of document updates and deletions is an interesting problem on its own that we won't consider further [29, 21, 49]. We assume word-level inverted lists that are sorted in document order.

2.1.2 Online Index Maintentance

Inserting a new document into a document collection involves in principle the addition of a new posting to every inverted list corresponding to a term in the document. If lists are document-ordered and new documents are assigned monotonically increasing numbers, new postings can be added to a list by simply appending them at the end of it. A single document may have a few hundred distinct terms, meaning that for every new document the system must update hundreds of inverted lists. In most cases, a list update can be carried out with one block read to fetch the list and one block write to store it back to disk after updating it. The cost of updating the index using this naive scheme is sufficiently high that in this raw form is not likely to be useful. The only practical solution is to amortize the cost of updating the lists over a batch of document insertions [118].

Index building typically involves parsing a batch of new documents into inverted lists that are temporarily maintained in main memory for improved efficiency [35]. When memory gets full, the system flushes the inverted lists to disk updating the on-disk index. During indexing, queries can be evaluated combining the disk index with the new in-memory inverted lists. Early work recognizes as main requirement in the above process the contiguous storage on disk of the postings belonging to each term [102]. Storage contiguity improves access efficiency for both query processing and index maintenance [118], but introduces the need for complex dynamic storage management and frequent or bulky relocations of postings [65]. On the other hand, if the system keeps the inverted lists non-contiguously on disk, then it avoids relocations but may need multiple seeks during query processing to retrieve an inverted list.

In-place methods build each inverted list incrementally as new documents are pro-



(a) (b)

Figure 2.2: Merges and files produced after the first 10 memory flushes for the (a) Immediate Merge or Remerge, and (b) Nomerge methods. Numbers within nodes represent size.

cessed. Documents are parsed into postings that are accumulated in memory, until memory is exhausted. Then, for each term that has postings in memory, the system fetches its inverted list from disk, appends the new postings to it, and writes the updated list back to disk. In the end, the memory is freed and the next batch of documents can be processed. The need for contiguity makes it necessary to relocate the lists when they run out of empty space at their end [102, 65, 64, 22]. One can amortize the cost of relocation by preallocating list space for future appends using various criteria [102]. Note that, due to the required list relocations, it is quite difficult –if not impossible– to keep the lists on disk sorted by term.

The *merge-based methods* merge the in-memory postings and the disk index into a single file on disk. The disk index stores all inverted lists in lexicographical order. When memory gets full, the index is sequentially read from disk list-by-list. Each list is then merged with new postings from memory and appended to a new file on disk, creating the new index. Finally, queries are redirected to the newly created index and the old index is deleted. This index update strategy is called *Immediate Merge* or *Remerge* (Figure 2.2a) [62, 21, 49]. Even though in-place index maintenance has linear asymptotic disk cost that is lower than the polynomial cost of Remerge, Remerge uses sequential disk transfers instead of random disk accesses and is experimentally shown to outperform in-place methods [64]. Nevertheless, each memory flush forces the entire index to be processed.

A trivial form of merge-based update is the *Nomerge* method which does not perform any merge operations (Figure 2.2b). Whenever the main memory is exhausted, the inverted lists from the memory bufferload are sorted and transferred to disk creating a

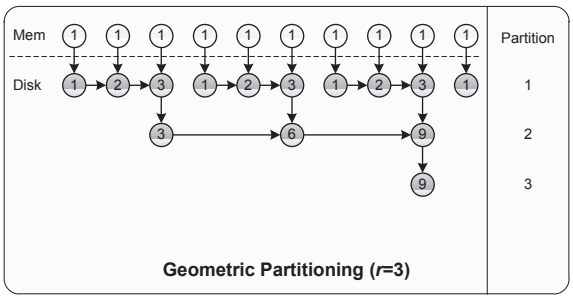
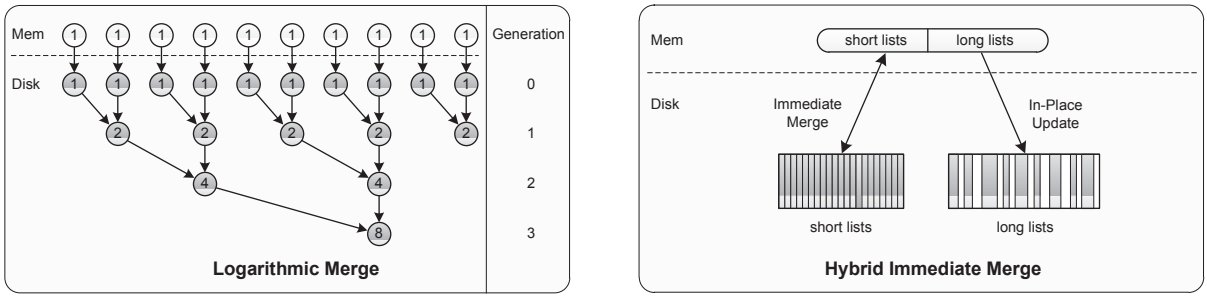


Figure 2.3: Merge sequence of Geometric Partitioning with $r = 3$, for the first 10 memory flushes. Numbers within nodes represent size.

new sub-index. This sub-index (also called *run*) corresponds to the latest batch of documents processed, and the set of all sub-indexes comprise the on-disk index for the whole document collection. Retrieving an inverted list to evaluate a query requires fetching its fragments from the multiple runs, which can easily become the dominant cost of query evaluation. Overall, this strategy maximizes indexing performance but leads to very poor query performance due to the excessive fragmentation of inverted lists.

Between the two extremes of Remerge and Nomerge there is a family of merge-based methods that permit the creation of multiple inverted files on disk to amortize the indexing cost, but bound the query latency by periodically merging them according to specific patterns. Essentially, they tradeoff query performance with index maintenance performance by having a controlled merging of runs.

In *Geometric Partitioning*, the disk index is composed by a tightly controlled number of *partitions* [62, 63]. Each partition stores a single sub-index and has a maximum size. The maximum sizes of the partitions form a geometric sequence with ratio r: the limit to the number of postings for the k-th partition is r times the limit of the (k-1)-th partition. In particular, if the memory bufferload has capacity M then the i-th partition has a maximum size of $(r-1)r^{i-1}M$. When the memory bufferload is full, it is merged with the sub-index at partition 1 and stored at partition 1. If the size of the sub-index created reaches the maximum size $(r-1)M$ of the partition, it is merged with the existing sub-index at partition 2 and placed there. In general, whenever the size of a sub-index created at partition k is more than $(r-1)r^{k-1}M$, it is merged with the sub-index at partition $k+1$ and stored there. Figure 2.3 illustrates the merge pattern produced by 10 memory flushes for Geometric Partitioning with $r = 3$. Alternatively, the method can dynamically



(a) (b)

Figure 2.4: Index maintenance approach for the (a) Logarithmic Merge and (b) Hybrid Immediate Merge methods.

adjust the parameter r to keep the number of disk sub-indexes at most p.

The *Logarithmic Merge* method introduces the concept of index *generation* to decide when to merge the sub-indices [21]. The sub-index created from the memory bufferload is of generation 0. A sub-index is said to be of generation g+1 if it is created by merging all sub-indices of generation g. A merge event is triggered whenever the creation of a new sub-index leads to a situation in which there are more than one sub-indices of the same generation g. All sub-indices of generation g are then merged to create a new disk sub-index of generation g+1 (Figure 2.4a).

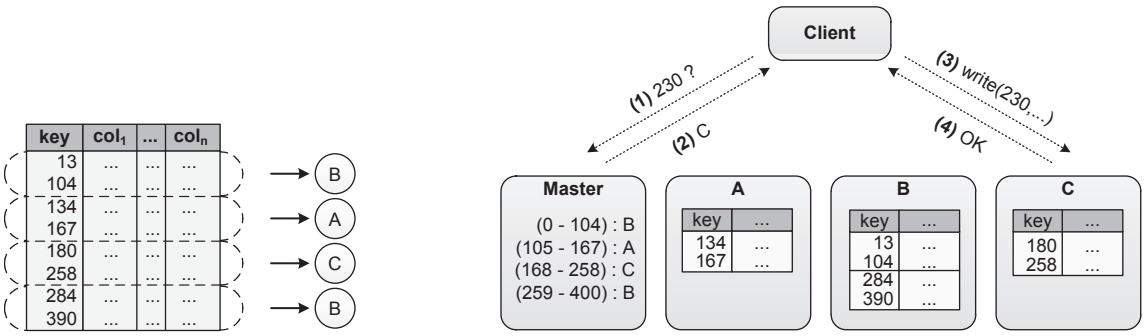
Hybrid methods separate terms into short and long. One early approach hashed short terms accumulated in memory into fixed-size disk regions called buckets. If a bucket filled up, the method categorized the term with the most postings as long and kept it at a separate disk region from that point on [102]. In several recent hybrid methods, the system uses a merge-based approach for the short terms (e.g. Immediate Merge or Logarithmic Merge) and in-place updates for the long ones (Figure 2.4b) [24]. They treat each term as short or long depending on the number of postings that have shown up in total until the current moment, or currently participate in the merging process. A recent hybrid method separates the terms into frequent and non-frequent according to their appearance in query logs, and maintains them in separate sub-indices of multiple partitions each [50]. Frequent terms use a merge strategy designed for better query performance, while infrequent terms rely on a merge strategy that attains better update performance.

2.1.3 Real-Time Search

Given the high cost of incremental updates and their interference with concurrent search queries, a main index can be combined with a smaller index that is frequently rebuilt (e.g. hourly) and a Just-in-Time Index (JiTI) [61]. JiTI provides (nearly) instant retrieval for content that arrives between rebuilds. Instead of dynamically updating the index on disk, it creates a small inverted file for each incoming document and chains together the inverted lists of the same term among the different documents. Earlier work on web search also pointed out the need to update an inverted file with document insertions and deletions in real time [29]. Instead of a word-level index, the Codir system uses a single bit to keep track of multiple term occurrences in a document block (*partial inverted index*), and processes search queries by combining a transient memory-based index of recent updates with a permanent disk-based index.

Twitter commercially provided the first real-time search engine, although other companies (e.g., Google, Facebook) are also launching real-time search features [46]. Real-time search at Twitter is recently supported by the Earlybird system that consists of inverted indices maintained in the main memory of multiple machines [20]. Earlybird reuses query evaluation code from the Lucene search engine [75], but also implements the term vocabulary as an optimized hash table, and the inverted list as a collection of document-ordered segments with increasing size.

In comparison to the batch scheme used until recently, the incremental update scheme of Percolator from Google reduces the average latency of document processing by a factor of 100, although it is still considered insufficient for real-time search [83, 20]. Stateful incremental processing has also been proposed as a general approach to improve the efficiency of web analytics over large-scale datasets running periodically over MapReduce [39, 68]. A different study shows that the throughput achieved by a method optimized for construction of inverted files across a cluster of multicore machines is substantially higher than the best performance achieved by algorithms based on MapReduce [110]. Earlier work on batch index building proposed a software-pipeline organization to parallelize the phases of loading the documents, processing them into postings, and flushing the sorted postings to disk as a sorted file [76].



(a) Bigtable tablet partitioning

(b) Bigtable centralized architecture

Figure 2.5: (a) Each table is partitioned into a number of tablets for load balancing, which are subsequently assigned to servers. (b) A master node keeps the mapping between tablets and servers. Clients must first contact the master node to store or access data.

2.2 Large-Scale Data Management

Scalable datastores (also referred to as NoSQL stores [27, 91]) are distributed storage systems capable of managing enormous amounts of structured data for online serving and analytics applications. Across different workloads, they weaken the relational and transactional assumptions of traditional databases to achieve horizontal scalability and availability, and meet demanding throughput and latency requirements. In this section, we present previous research activity related to the architecture and storage organization of datastores.

2.2.1 Scalable Datastores

Bigtable is a centralized structured storage system that partitions data across multiple storage servers, called tablet servers [28]. A *tablet* is simply a range of consecutive rows within a table (Figure 2.5a). A master node is responsible for assigning tablets to tablet servers, handling node joins and failures, and balancing tablet-server load. A client must first contact the master node to locate the server responsible for a key, and then communicates directly with the tablet server for reads and writes (Figure 2.5b). Incoming data to a tablet server is first logged to disk and then kept in the memory. When the occupied memory reaches a threshold, a *minor compaction* transfers memory data to an immutable disk file (*SSTable*). Read operations might need to merge updates from an

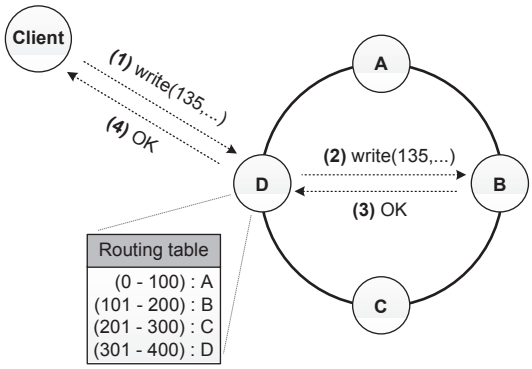


Figure 2.6: Dynamo decentralized architecture. Any node on the ring can coordinate a request from a client. We assume the ring space is (0,400) and nodes A, B, C, D are assigned values 100, 200, 300 and 400 respectively.

arbitrary number of SSTables. Periodic *merging compactions* performed in background transform multiple files into a single file, while a *major compaction* regularly merges all files to a single file free of deleted entries. Bloom filters are used to skip unnecessary key searches over SSTables.

Azure is another scalable datastore that partitions data by key range across different servers [25]. It provides storage in the form of blobs (user files), tables (structured storage), or queues (message delivery). All these data types are internally stored into tables which are partitioned into RangePartitions similar to Bigtable tablets (Figure 2.5a). The system keeps the data of each partition over multiple checkpoint files whose number is kept under control through periodic merging. With an emphasis on data analytics, LazyBase combines update batching with pipelining and allows per-read-query tradeoffs between freshness and performance [30]. The system reduces query cost through a tree-based merging of sorted files triggered by the number of leaves or a time period. HBase[1], Accumulo[2] and Hypertable[3] are open-source variations of Bigtable [33, 81]. Compaction tuning combined with special metadata files can improve the read performance of HBase [15].

Dynamo is a decentralized storage system which stores key-value pairs over a distributed hash table [40]. Its partitioning scheme relies on consistent hashing to distribute the load across multiple storage nodes. In consistent hashing, the output range of a hash

[1]http://hbase.apache.org/
[2]http://accumulo.apache.org/
[3]http://hypertable.com/

function is treated as a fixed circular space or "ring". Each node in the system is assigned a random value within this space which represents its "position" on the ring, and is responsible for the region between it and its predecessor on the ring (Figure 2.6). Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring. This hashing scheme is termed "consistent" because when the number of nodes (i.e., hash slots) changes –and thus a number of keys must be rehashed– only the nodes adjacent to the nodes that joined or left the system are affected. Dynamo accepts different pluggable persistent components for local storage of items. The system can trade off durability for performance by keeping incoming data in memory and periodically transferring it to disk. It supports eventual consistency, which allows for updates to be propagated to all replicas asynchronously. The Dynamo-inspired Project Voldemort[4] is an open-source datastore that supports efficient bulk updates through a custom read-only storage engine combined with a data deployment pipeline based on Hadoop [101]. Riak[5] is another open-source distributed datastore built upon the ideas and design decisions of Dynamo.

Cassandra[6] is an open-source write-optimized datastore, designed and implemented based on the data model of Bigtable and the architecture of Dynamo [59]. Similar to Dynamo the cluster is configured as a ring of nodes, and it uses asynchronous replication across multiple nodes along with the hinted handoff technique. There is no master node and the nodes use a gossip mechanism to propagate the current state of the cluster. The eventual consistency model is used, in which consistency level can be selected by the client. Similar to Bigtable, Cassandra uses a structure of immutable files on disk to support reads without locking. Disk files are created when the size of the data items accumulated in memory reaches a threshold and are periodically merged.

RAMcloud uses a log-structured approach to manage data on both memory and disk for fast crash recovery [80]. A coordinator node assigns objects to storage servers in units of tablets. When a server receives a write request, it appends the new object to its in-memory log and forwards that log entry to several backup servers. The backups buffer this information in memory and return immediately. The master server returns to the

[4]http://github.com/voldemort/voldemort

[5]http://basho.com/riak/

[6]http://cassandra.apache.org/

client once all backups have acknowledged receipt of the log data. When a backup's buffer fills, it writes the accumulated log data from other nodes to disk or flash and deletes the buffered data from memory.

For low power consumption, the FAWN key-value store uses a log file on flash storage indexed by an in-memory hash table [1]. Data is distributed across storage nodes using consistent hashing. All write requests to a storage node are simply appended to a log on flash storage. In order to satisfy reads with a single random access, FAWN maintains a DRAM hash table per node that maps its keys to an offset in the append-only log. The SILT key-value store combines flash storage with space-efficient indexing and filtering structures in memory [66]. When a key is inserted at a node, it is appended into a write-optimized, log called *LogStore*, and a corresponding in-memory index is updated. The amount of memory required to index objects on LogStore is drastically reduced using partial-key cuckoo hashing and entropy-coded tries. Once full, a LogStore is converted into an immutable sorted hash table (*HashStore*) that does not require any in-memory index to locate entries. Periodically, multiple HashStores are merged into a single extremely compact index representation called *SortedStore*, and all deleted or overwritten entries are garbage collected.

Masstree is a shared-memory, concurrent-access structure for data that fully fits in memory [72]. A collection of B^+-trees organized as a trie is used as a highly concurrent data structure in memory. The tree is shared among all cores and allows for efficient implementation of inserts, lookups and range queries (traverse subsets of database in sorted order by key). Lookups use no locks, while updates acquire locks only on the tree nodes involved. Similar to RAMcloud, all reads and writes are served from memory and data is additionally logged and checkpointed for durability. Haystack is a persistent storage system for photos that implements data volumes as large files over an extent-based file system across clusters of machines [10].

2.2.2 Storage Organization

In this section, we outline representative known methods for the problem of write-optimized data storage. We only consider external-memory data structures that handle one-dimensional range queries to report the points contained in a single-key interval, a type of query that

is commonly used by large-scale web applications (Section 6.2). Thus we do not examine spatial access methods (e.g., R-tree, k-d-B-tree) that directly store multidimensional objects (e.g., lines) or natively handle multidimensional queries (e.g., rectangles). Spatial structures have not been typically used in datastores until recently [44]; also, at worst case, the lower-bound cost of orthogonal *search* in d dimensions ($d > 1$) is fractional-power I/O for linear space and logarithmic I/O for nonlinear storage space [108].

A data structure is *static* if it remains searchable and immutable after it is built; it is *dynamic* if it supports both mutations and searches throughout its lifetime. The processing cost of a *static* structure refers to the total complexity to insert an entire dataset, and the insertion cost of a *dynamic* structure refers to the amortized complexity to insert a single item [12]. In a datastore, multiple static structures are often combined to achieve persistent data storage because filesystems over disk or flash devices are more efficient with appends rather than in-place writes [28, 80, 66].

Some datastores rely on the storage engine of a relational database at each server. For instance, PNUTS [32] uses the InnoDB storage engine of MySQL, and Dynamo [40] the Berkeley DB Transactional Data Store. In a relational database, data is typically stored on a B-tree structure. Let N be the total number of inserted items, B items the disk block size, and M items the memory size for caching the top levels of the tree. We assume unary cost for each block I/O transfer. One B-tree insertion costs $O(\log_B \frac{N}{M})$ and a range query of output size Z items costs $O(\log_B \frac{N}{M} + \frac{Z}{B})$ [114]. In contrast, the *Log-structured File System (LFS)* accumulates incoming writes into a memory-based buffer [90]. When the buffer fills up, data is transferred to disk in a single large I/O and deleted from memory. RAMCloud and FAWN use a logging approach for persistent data storage [1, 80].

Inspired from LFS, the *Log-Structured Merge-Tree (LSM-tree)* is a multi-level disk-based structure optimized for high rate of inserts/deletes over an extended period [79]. In a configuration with ℓ components, the first component is a memory-resident indexed structure (e.g., AVL tree), and the remaining components are modified B-trees that reside on disk. Component size is the storage space occupied by the leaf level. The memory and disk cost is minimized if the maximum size of consecutive components increases by a fixed factor r. When the size of a component C_i reaches a threshold, the leaves of C_i and C_{i+1} are merged into a new C_{i+1} component. The LSM-tree achieves higher insertion performance than a B-tree due to increased I/O efficiency from batching incoming updates

into large buffers and sequential disk access during merges. The insertion cost of the LSM-tree is $O(\frac{r}{B}\log_r \frac{N}{M})$, where $\ell = \log_r \frac{N}{M}$ is the number of components. However, a range query generally requires to access all the components of an LSM-tree. Thus, a range query costs $O(\log_r \frac{N}{M} + \frac{Z}{B})$ if search is facilitated by a general technique called *fractional cascading* [114]. Bigtable and Azure rely on LSM-trees to manage persistent data [25, 28, 98].

The *Stepped-Merge Algorithm (SMA)* is an optimization of the LSM-tree for update-heavy workloads [55]. SMA maintains $\ell + 1$ levels, with up to k B-trees (called *runs*) at each level $i = 0, \ldots, \ell - 1$, and 1 run at level ℓ. Whenever memory gets full, it is flushed to a new run on disk at level 0. When k runs accumulate at level i on disk, they are merged into a single run at level $i + 1$, $i = 0, \ldots, \ell - 1$. SMA achieves insertion cost $O(\frac{1}{B}\log_k \frac{N}{M})$, and query cost $O(k \log_k \frac{N}{M} + \frac{Z}{B})$ under fractional cascading. A compaction method based on SMA (with unlimited ℓ) has alternatively been called *Sorted Array Merge Tree (SAMT)* [98]. If we dynamically set $k = \frac{N}{M}$ to SMA, we get the *Nomerge* method, which creates new sorted files on disk without merging them (Section 2.1.2). Although impractical for searches, Nomerge is a baseline case for low index-building cost. A variation of SMA is applied with $k = 10$ by the Lucene search engine [34], or $k = 4$ by Cassandra and GTSSL [98].

Text indexing maps each term to a list of document locations (postings) where the term occurs. Merge-based methods flush postings from memory to a sorted file on disk and occasionally merge multiple files. Along a sequence of created files, Geometric Partitioning introduces the parameter r to specify an upper bound ($(r-1)r^{i-1}M$) at the size of the i-th file, $i = 1, 2, \ldots$, for memory size M (Section 2.1.2). Hierarchical merges guarantee similar sizes among the merged files and limit the total number of files on disk. The I/O costs of insertion and search in Geometric Partitioning are asymptotically equal to those of the LSM-tree [63, 114] and the Cache-Oblivious Lookahead Array (COLA) [11]. Geometric Partitioning can directly constrain the maximum number p of files with dynamic adjustment of r. Setting $p = 1$ leads to the *Remerge* method, which always merges the full memory into a single file on disk and requires one I/O to handle a query. In the particular case of $p = 2$, Geometric Partitioning is also known as *Square Root Merge* [24]. A variation of Geometric Partitioning with $r = 2$ is used by Anvil [71] and $r = 3$ by HBase [98]; SILT uses a single immutable sorted file (similar to $p = 1$) on

Table 2.1: Summary of storage structures typically used in datastores. We include their I/O complexities for insertion and range query in one-dimensional search over single-key items.

The I/O Complexity of Datastore Storage Structures

Dynamic Data Structure	Insertion Cost	Query Cost	System Example
B-tree	$O(\log_B \frac{N}{M})$	$O(\log_B \frac{N}{M} + \frac{Z}{B})$	PNUTS [32], Dynamo [40]
Log-structured File System (LFS)	$O(\frac{1}{B})$	N/A	RAMCloud [80], FAWN [1]
Log-structured Merge Tree (LSM-tree), Geometric, r-COLA	$O(\frac{r}{B} \log_r \frac{N}{M})$	$O(\log_r \frac{N}{M} + \frac{Z}{B})$	HBase [98], Anvil [71], Azure [25], Bigtable [28], bLSM [93]
Geometric with p partitions, Remerge (special case $p = 1$)	$O(\frac{1}{B} \sqrt[p]{\frac{N}{M}})$	$O(p + \frac{Z}{B})$	bottom layer of SILT [66]
Stepped-Merge Algorithm (SMA), Sorted Array Merge Tree (SAMT), Nomerge (special case $k = N/M$)	$O(\frac{1}{B} \log_k \frac{N}{M})$	$O(k \log_k \frac{N}{M} + \frac{Z}{B})$	Cassandra [52], GTSSL [98], Lucene [34]

flash storage [66].

We summarize the asymptotic insertion and range-query costs of the above structures in Table 2.1. Log-based solutions achieve constant insertion cost, but lack efficient support for range queries. SMA incurs lower insertion cost but higher query cost than the LSM-tree. Geometric Partitioning with p partitions takes constant time to answer a query, but requires fractional-power complexity for insertion.

2.2.3 Related Issues

Transactional Support. An evaluation of transactional support in commercial cloud database systems shows a diversity across the business models of different providers [58]. The PNUTS system applies a simple relational model to organize attribute records into tables of a geographically-distributed database [32]. For point or range queries, it uses alternative physical layers, such as a filesystem-based hash table or a MySQL/InnoDB database. The primary bottleneck of the system is the disk seek capacity required for data storage and messaging. PNUTS can achieve higher throughput of bulk record insertion

with a planning phase to minimize the sum of partition movement and insertion time [96]. Alternatively, snapshot text files can be created by Hadoop for direct data import into the MySQL tables of PNUTS [97].

Percolator extends Bigtable to support cross-row, cross-table transactions through versioning [83]. Megastore organizes structured data over a wide-area network as a collection of small databases, called *entity groups* [7]. Entities within an entity group are mutated with ACID transactions, while operations across entity groups typically apply looser semantics through asynchronous messaging. G-Store allows the dynamic creation of key groups over which multi-key transactional access is supported [36]. Anvil is a modular, extensible toolkit for database backends [71]. The system periodically digests written data into read-only tables, which are merged through Geometric Partitioning (also similar to generational garbage collection [106]).

The ecStore realizes a scalable range-partitioned storage system over a tree-based structure [109]. The system automatically organizes histogram buckets to accurately estimate access frequencies and replicate the most popular data ranges, while it bases transaction management on versioning and optimistic concurrency control. The ES^2 system supports both vertical and horizontal partitioning of relational data [26]. It also provides efficient bulk loading, small-range queries for transaction processing and sequential scans for batch analytical processing.

Data Structures. The indexed sequential access method (ISAM) refers to a disk-based tree used by database systems [87]. Each tree node has fixed size (e.g. 4KB), and the data is sequentially stored key-sorted in the leaf nodes before the non-leaf nodes are allocated. The ISAM structure is static after it is created, because inserts and deletes affect only the contents of leaf pages. If subsequent inserts overflow a leaf node, they are stored at arrival order in additional chained blocks allocated from an overflow area. For intense update loads and concurrent analytics queries, the Partitioned Exponential file (PE file) dynamically partitions data into distinct key ranges and manages separately each partition similarly to an LSM-tree [57]. Over time, the size of a partition changes by orders of magnitude, while it always has to fully fit in memory. Insertion cost varies significantly due to the required storage reorganization and data merging within each partition. Similarly, search cost varies because it involves all levels of a partition, uses

tree indexing at each level, and interferes with concurrent insertions.

A *dictionary* stores a mapping from keys to values. In the *external memory* model, a two-level memory hierarchy consists of internal memory and a disk with I/O block size B. The alternative *cache-oblivious* model is applied in multi-level memory hierarchies assuming that the block size B is unavailable for tuning. Bender et al. introduce the cache-oblivious lookahead array (g-COLA) as a multi-level structure, where g is the factor of size growth between consecutive levels [11]. Due to buffering and amortized I/O, g-COLA achieves faster random inserts than a traditional B-tree, but slower searches and sorted inserts. Brodal et al. proposed the xDict dynamic dictionary for optimal, configurable tradeoff in space, query and update costs at the cache-oblivious model [17].

A *versioned dictionary* is a dictionary with an associated version tree, which supports queries on any version, updates on leaf versions, and cloning on any version by adding a child. A structure is called *fully-versioned* if it supports arbitrary version trees, and *partially-versioned* if it only supports a linked list as a (degenerate) version tree. As a partially-versioned dictionary, the multiversion access structure achieves logarithmic update time, optimal space and optimal query time for a key range at a specific time and a key in a specified time range [107]. In the cache-oblivious model, Byde et al. recently proposed the stratified B-tree as a fully-versioned dictionary, which achieves optimal, configurable tradeoff among query time, update time, and space [105].

Benchmarking. The performance and scalability of several data serving systems has been studied under the benchmark framework called Yahoo! Cloud Serving Benchmark (YCSB) [33]. With measurements, it was found that Cassandra achieves higher performance at write-heavy workloads, PNUTS at read-heavy workloads, and HBase at range queries. The YCSB++ adds extensions to YCSB for advanced features that include bulk data loading, server-side filtering and fine-granularity access control [81]. BigBench is an end-to-end big data benchmark whose underlying business model is a product retailer [47]. Wang et al. introduce BigDataBench, a benchmark covering a broad range of application scenarios and real-world datasets, including structured, semi-structured and unstructured text data and graph datasets [45].

Chapter 3

Incremental Text Indexing for Fast Disk-Based Search

3.1 Introduction

3.2 Background

3.3 Motivation

3.1 Introduction

Digital data is accumulated at exponential rate due to the low cost of storage space, and the easy access by individuals to applications and web services that support fast content creation and data exchange. Traditionally, web search engines periodically rebuild in batch mode their entire index by ingesting tens of petabytes of data with the assistance of customized systems infrastructure and data processing tools [16, 39, 38]. This approach is sufficient for websites whose content changes relatively infrequently, or their enormous data volume makes infeasible their continuous tracking.

Today, users are routinely interested to search the new text material that is frequently added across different online services, such as news websites, social media, mail servers and file systems [61, 94, 22, 13, 46, 20]. Indeed, the sources of frequently-changing content are highly popular web destinations that demand almost immediate search visibility of

their latest additions [61, 63, 22, 83, 20]. Real-time search refers to the fast indexing of fresh content and the concurrent support of interactive search (Section 2.1.3); it is increasingly deployed in production environments (e.g., Twitter, Facebook) and actively investigated with respect to the applied indexing organization and algorithms. Text-based retrieval remains the primary method to identify the pages related to a web query, while the inverted file is the typical index structure used for web search (Section 2.1.1).

A web-scale index applies a distributed text-indexing architecture over multiple machines [3, 8, 16, 5, 60]. Scalability is commonly achieved through an index organization called *document partitioning*. The system partitions the document collection into disjoint sub-collections across multiple machines, and builds a separate inverted index (*index shard*) on every machine. A client submits a search query to a single machine (*master* or *broker*). The master broadcasts the query to the machines of the search engine and receives back disjoint lists of documents that satisfy the search criteria. Subsequently, it collates the results and returns them in ranked order to the client. Thus, a standalone search engine running on a single machine provides the basic building block for the distributed architectures that provide scalable search over massive document collections.

When a fresh collection of documents is crawled from the web, an *offline indexing* method can be used to rebuild the index from scratch (Section 2.1.1). Input documents are parsed into postings, with the accumulated postings periodically flushed from memory into a new partial index on disk. Techniques similar to external sorting merge at the end the multiple index files into a single file at each machine. Due to fragmentation of each inverted list across multiple partial indices on a machine, search is supported by an older index during the update. Instead, *online indexing* continuously inserts the freshly crawled documents into the existing inverted lists and periodically merges the generated partial indices to dynamically maintain low search latency (Section 2.1.2).

Disk-based storage is known as a performance bottleneck in search. Thus, index-pruning techniques have been developed to always keep in memory the inverted lists of the most relevant keywords or documents, but lead to higher complexity in index updating and context-sensitive query handling [18, 118, 2, 78, 100]. Although the latency and throughput requirements of real-time search are also currently met by distributing the full index on the main memory of multiple machines [20], the purchase cost of DRAM is two orders of magnitude higher than that of disk storage capacity [92]. Therefore, it is

crucial to develop disk-based data structures, algorithmic methods and implementation techniques for incremental text indexing to interactively handle queries without the entire index in memory.

In this chapter, we examine the fundamental question of whether disk-based text indexing can efficiently support incremental maintenance at low search latency. We focus on incremental methods that allow fast insertions of new documents and interactive search over the indexed collection. We introduce two new methods, the *Selective Range Flush* and *Unified Range Flush*. Incoming queries are handled based on postings residing in memory and the disk. Our key insight is to simplify index maintenance by partitioning the inverted file into disk blocks. A block may contain postings of a single frequent term or the inverted lists that belong to a range of several infrequent terms in lexicographic order. We choose the right block size to enable sequential disk accesses for search and update. When memory gets full during index construction, we only flush to disk the postings of those terms whose blocks can be efficiently updated. Due to the breadth of the examined problem, we leave outside the study scope several orthogonal issues that certainly have to be addressed in a production-grade system, such as concurrency control [61], automatic failover [60], or the handling of document modifications and deletions [67, 49].

For comparison purposes, we experiment with a software prototype that we developed, but we also apply asymptotic analysis. In experiments with various datasets, we achieve search latency that depends on the number of retrieved postings rather than fragmentation overhead, and index building time that is substantially lower than that of other methods with similar search latency. To the best of our knowledge, our indexing approach is the first to group infrequent terms into lexicographic ranges, partially flush both frequent and infrequent terms to disk, and combine the above with block-based storage management on disk. Prior maintenance methods for inverted files randomly distributed the infrequent terms across different blocks [102], or handled each term individually [119, 19]. Alternatively, they partially flushed to disk only the frequent terms [23, 22], or used disk blocks of a few kilobytes with limited benefits [19, 102].

Table 3.1: Summary of the asymptotic cost (in I/O operations) required to incrementally build inverted files and retrieve terms for query handling. N is the number of indexed postings and M is the amount of memory used for postings gathering. The parameter a (e.g., $a = 1.2$) refers to the Zipfian distribution (Section 9.1).

Index Maintenance Method	**Build Cost**	**Search Cost**
Nomerge [102, 63, 24, 51]	$\Theta(N)$	N/M
Immediate Merge [63, 35, 22]	$\Theta(N^2/M)$	1
Logarithmic Merge [24] Geometric Partitioning [62, 63]	$\Theta(N \cdot \log(N/M))$	$\log(N/M)$
Geometric Partitioning with $\leq p$ partitions [63]	$\Theta(N \cdot (N/M)^{1/p})$	p
Hybrid Immediate Merge [24, 22] Unified Range Flush [Section 9.1]	$\Theta(N^{1+1/a} / M)$	1 or 2 (according to the list threshold)
Hybrid Logarithmic Merge [24]	$\Theta(N)$	$\log(N/M)$

3.2 Background

In this section, we summarize the current general approaches of incremental text indexing, and factor out the relative differences of existing methods with respect to the new methods that we introduce.

As discussed in Section 2.1.2, merge-based methods maintain on disk a limited number of files that contain fragments of inverted lists in lexicographic order. During a merge, the methods read sequentially the lists from disk, merge each list with the new postings from memory, and write the updated lists back to a new file on disk. The methods amortize the I/O cost if they create on disk multiple inverted files and merge them in specific patterns. In-place methods avoid to read the whole disk index, and incrementally build each inverted list by appending new memory postings at the end of the list on disk.

Hybrid methods separate terms into short and long according to the term popularity, and use a merge-based approach for the short terms and in-place appends for the long ones. The system treats a term as short or long depending on the number of postings that either have shown up in total until now (*contiguous*) [23], or participate in the current merging process (*non-contiguous*) [24]. In the non-contiguous case, if a term contributes more than T (e.g., $T = 1$MB) postings to the merging process, the method moves the postings from the merge-based index to the in-place index; this reduces the build time,

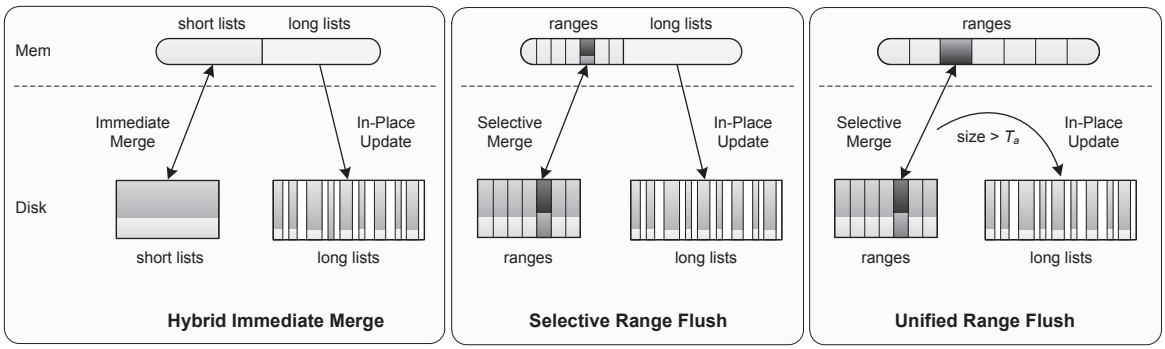


Figure 3.1: Hybrid Immediate Merge only applies partial flushing to long (frequent) terms, while Selective Range Flush (SRF) and Unified Range Flush (URF) partially flush both short (infrequent) and long terms. Unlike SRF, URF organizes all postings in memory as ranges, allows a term to span both the in-place and merge-based indices, and transfers postings of a term from the merge-based to the in-place index every time they reach a size threshold T_a (see also Section 4.6).

but may slightly increase the retrieval time of long terms due to their storage on both the in-place and merge-based indices.

As shown in Table 3.1 for representative methods, the asymptotic complexity of index building is estimated by the number of I/O operations, expressed as function of the number of indexed postings. We include the search cost as the number of partial indices (or runs) across which an inverted list is stored. The Nomerge method flushes its postings to a new run on disk every time memory gets full and provides a baseline for the minimum indexing time. The Immediate Merge method repeatedly merges the postings in memory with the entire inverted file on disk every time memory gets full. The Geometric Partitioning and Logarithmic Merge methods keep multiple runs on disk and use a hierarchical pattern to merge the postings of memory and the runs on disk. The Geometric Partitioning method with $\leq p$ partitions adjusts continuously the fan-out of the merging tree to keep the number of runs on disk at most p. Hybrid versions of the above methods partition the index into in-place and merge-based indices.

Our methods, Selective Range Flush and Unified Range Flush, differ from existing ones, because we organize the infrequent terms into ranges that fit into individual disk blocks, and store each frequent term into dedicated disk blocks (Figure 3.1). Additionally, we only *partially* flush frequent and infrequent terms from memory to preserve the disk

Table 3.2: Main functional differences among existing and our new methods of incremental text indexing.

Index Maintenance Method	**Update Scheme**	**Threshold Count**	**Merging Pattern**	**Partial Flushing**	**Flushing Criterion**	**Storage Unit**
Nomerge	new run	none	none	none	full mem.	runs
Immediate Merge	merge	none	sequential	none	full mem.	single run
Geometric Partition.	merge	none	hierarchical	none	full mem.	partitions
Hybrid Log. Merge	hybrid	merge/total	hierarchical	none	full mem.	segments
Hybrid Imm. Merge	hybrid	merge/total	sequential	in-place	list size	segments
Selective Range Flush	hybrid	total	range-based	both	list ratio	blocks
Unified Range Flush	hybrid	merge	range-based	both	range size	blocks

I/O efficiency. The two methods differ from each other with respect to the criteria that they apply to categorize the terms as short or long, and also to determine which terms should be flushed from memory to disk. In Table 3.1 we include the asymptotic costs of Unified Range Flush as estimated in Section 9.1.

According to experimental research, build time may additionally depend on system structures and parameters not always captured by asymptotic cost estimates [64, 22]. Thus, although the Hybrid Immediate Merge and Unified Range Flush have the same asymptotic complexity as shown in Table 3.1, we experimentally find their measured merge performance to substantially differ by a factor of 2. More generally, the potential discrepancy between theoretical and empirical results is a known issue in literature. For instance, online problems are the type of optimization problems that receive input and produce output in online manner, but each output affects the cost of the overall solution. Several paging algorithms are examples of online algorithms that theoretically incur the same relative cost (competitive ratio) to an optimal algorithm, but they clearly differ from each other with respect to experimentally measured performance [14].

In Table 3.2, we factor out the main functional differences among the representative methods that we consider. The index update varies from simple creation of new runs, to purely merge-based and hybrid schemes. In hybrid schemes, term postings are respectively stored at the merge-based or in-place index according to their count in the entire index (*Total*) or the index part currently being merged (*Merge*). The merging pattern varies

from sequential with a single run on disk, to hierarchical that tightly controls the number of runs, and range-based that splits the index into non-overlapping intervals of sorted terms. When the memory fills up, most existing methods flush the entire memory to disk except for the Hybrid Immediate Merge that partially flushes frequent terms; in contrast, our methods apply partial flushing to both frequent and infrequent terms (Figure 3.1). The criterion of partial memory flushing alternatively considers the posting count of individual terms and term ranges or their ratio. Most methods allocate the space of disk storage as either one or multiple runs (alternatively called partitions or segments [24]) of overlapping sorted terms, while we use blocks of non-overlapping ranges.

3.3 Motivation

In this section we present the motivation of our work. We experimentally highlight that search latency is primarily spent on disk I/O to retrieve inverted lists. Across different queries, latency can be relatively high even when stop words are used or caching is applied, which makes the efficiency of storage access highly relevant in fast disk-based search [118, 6].

3.3.1 The Search Cost of Storage Fragmentation

Early research on disk-based indexing recognized as main requirement the contiguous storage of each inverted list [35, 102]. Although storage contiguity improves access efficiency in search and update, it also leads to complex dynamic storage management and frequent or bulky relocations of postings. Recent methods tend to relax the contiguity of inverted lists so that they lower the cost of index building. One particular study partitioned the postings of each term across multiple index files and stored the inverted list of each long term as a chain of multiple non-contiguous segments on disk [24]. Not surprisingly, it has been experimentally shown across different systems that multiple disk accesses (e.g., 7 in GOV2) may be needed to retrieve a fragmented inverted list regardless of the list length [63]. List contiguity is particularly important for infrequent terms because they dominate text datasets and are severely affected by list fragmentation. From the Zipfian distribution of term frequency, the inverted file of a 426GB text collection has more than

Table 3.3: Average search latency (ms) and the fraction of it spent on I/O, using the GOV2 dataset over the Zettair search engine.

Queries	Avg	I/O
50%	105	67%
75%	255	58%
90%	680	58%
95%	1,053	61%
100%	1,726	64%

Table 3.4: Average, median and 99th percentile of search latency (ms) when different numbers of stop words are applied with and without page caching in GOV2/Zettair.

stop	without caching			with caching		
words	avg	med	99th	avg	med	99th
0	1,726	291	19,616	1,315	274	13,044
10	640	247	5,283	508	217	4,182
20	489	242	3,221	413	204	2,873
100	411	232	2,398	341	188	1,959

99% of inverted lists smaller than 10KB [35, 24]. If a list of such size is fragmented into k runs, the delay of head movement in a hard disk typically increases the list retrieval time by a factor of k.

We examine the importance of query I/O efficiency using the Zettair search engine with an indexing method that stores the inverted lists contiguously on disk [115]. Using the index of the GOV2 text collection (426GB), we evaluate 1,000 standard queries [103] in a server as specified in Section 5.1 with the buffer cache disabled. Thus, we measure the time to return the 20 most relevant documents per query along with the percentage of time spent on I/O. We sort the queries by increasing response time and calculate the average query time for the 50%, 75%, 90%, 95% and 100% fastest of them. According to the last row of Table 3.3, 64% of the average query time is spent on I/O for reading inverted lists from disk. The percentage becomes 67% for the 50% fastest queries, which mostly consist of non-frequent terms with small inverted lists.

Caching keeps in memory the postings of frequently queried terms, while stop words are frequent terms usually ignored during query handling [118, 6]. From Table 3.4 it follows that enabling the page cache decreases by 24% the average latency, 6% the median, and 34% the 99th percentile. Caching is generally known to reduce the latency of interactive services, but it cannot directly address the problem of variable responsiveness in distributed systems, unless the entire working set resides in main memory [38]. If we additionally omit the 10, 20 or 100 most common stop words during query handling, the enabled buffer cache still decreases latency by about 18% on average. For instance, using

10 stop words combined with caching lowers the average latency by 71% from 1.7s to 508ms. Nevertheless, 45% of the average query time is still spent on I/O.

Query latency is often evaluated using average measurements [24, 62], which do not convey the high variations across different queries. In Table 3.3, the average query latency is about 1726ms, even though the 50% fastest queries only take an average of 105ms. If we presumably *double* the duration of the 50% fastest queries, the average latency across all the queries is only increased by 3%. Similarly, the discrepancy between the average and median latency measurements in Table 3.4 further demonstrates the effect from the few long queries to the measured statistics. Therefore, the average statistic understates the problem of list fragmentation.

Given the substantial time fraction of query handling spent on I/O and the significant increase that list fragmentation causes in I/O time, we advocate to preserve the list contiguity of frequent and infrequent terms through the design and storage-level implementation of the indexing method. Additionally, we aim to achieve low query latency both on average and across different percentiles.

Chapter 4

Selective Range Flush and Unified Range Flush Methods

4.1 Problem Definition

4.2 System Architecture

4.3 The Selective Range Flush Method

4.4 Evaluation of Selective Range Flush

4.5 Sensitivity of Selective Range Flush

4.6 The Unified Range Flush Method

4.7 Prototype Implementation

In this section, we first describe the studied problem along with our goals, and then explain the data structures and the Selective Range Flush (SRF) method to solve it. Motivated by our long experimental effort to tune SRF, we then proceed to the description of the Unified Range Flush (URF) method with simpler structure but similar (or even better sometimes) build and search performance. Finally, we describe the Proteus prototype implementation that we developed using the Zettair open-source search engine to evaluate the efficiency of the proposed methods.

4.1 Problem Definition

In this study we mainly focus on the incremental maintenance of inverted files for efficient index building and search. We do not examine the related problems of parsing input documents to extract new postings, or the ranking of retrieved postings for query relevance. We primarily aim to minimize the I/O time required to retrieve the term postings of a query and the total I/O time involved in index building. More formally we set the following two goals:

$$\text{query handling:} \quad \text{minimize} \quad \sum_i \text{I/O time to read the postings of term}_i \tag{4.1}$$

$$\text{index building:} \quad \text{minimize} \quad \sum_j \text{I/O time to flush posting}_j, \tag{4.2}$$

where i refers to the terms of a query, and j refers to the postings of the indexed document collection. The I/O time of query handling depends on the data volume read from disk along with the respective access overhead. Similarly, the total I/O time of index building is determined by the volume of data transferred between memory and disk along with the corresponding overhead. Accordingly, we aim to minimize the amount of read data during query handling, the amount of read and written data during index building, and the access overheads in both cases.

One challenge that we face in index building is that we do not know in advance the term occurrences of the incoming documents. As a result, we cannot optimally plan which postings to flush for maximum I/O efficiency every time memory gets full. Ideally, for efficient query handling we would store the postings of each term contiguously on disk in order to retrieve a requested term with a single I/O. Also, for efficient index building, we would prefer to flush new postings from memory to disk with a single write I/O and without any involvement of reads.

In fact, the above goals are conflicting because the I/O efficiency of query handling depends on the organization of term postings by index building. In the extreme case that we write new postings to disk without care for storage contiguity, query handling becomes impractical due to the excessive access overhead involved to read the fragmented postings from disk. As a reasonable compromise, we only permit limited degree of storage fragmentation in the postings of a term, and also ensure sufficient contiguity to read a term roughly sequentially during query handling. At the same time, we limit the volume

of data read during index building but with low penalty in the I/O sequentiality of disk reads and writes. Next, we explain in detail how we achieve that.

4.2 System Architecture

As we add new documents to a collection, we accumulate their term postings in memory and eventually transfer them to disk. We lexicographically order the terms and group them into ranges that fit into disk blocks (called *rangeblocks*) of fixed size B_r. Rangeblocks simplify the maintenance of inverted files because they allow us to selectively update parts of the index. We flush the postings of a range R from memory by merging them into the respective rangeblock on disk. If the merged postings overflow the rangeblock, we equally divide the postings –and their range– across the original rangeblock and any number of additional rangeblocks that we allocate as needed. For several reasons, we do not store all the lists in rangeblocks:

- First, the list of a frequent term may exceed the size of a single rangeblock.
- Second, the fewer the postings in a rangeblock the lower the update cost, because the merge operation transfers fewer bytes from disk to memory and back.
- Third, we should defer the overflow of a rangeblock, because the ranges that emerge after a split will accumulate fewer postings than the original range, leading to higher merging cost.
- Finally, we experimentally confirm that merge-based management involves repetitive reads and writes that are mostly efficient for collections of infrequent terms, while in-place management uses list appends that are preferable for terms with large number of postings.

Consequently, we store the list of a frequent term on exclusively occupied disk blocks that we call *termblocks*. We dynamically allocate new termblocks as existing termblocks run out of empty space. For efficiency, we choose the size B_t of the termblock to be different from the rangeblock B_r (Section 4.7). Where clarity permits, we collectively call *posting blocks* the rangeblocks and termblocks.

The lexicon is expected to map each term to the memory and disk locations where we keep the respective postings. The B-tree provides an attractive mapping structure, because it concisely supports ranges, and flexibly handles large numbers of indexed items. However, when the size of the dataset is at the range of hundreds of GB, as the ones we consider, we experimentally noticed that the B-tree introduces multiple disk seeks during lookups, which substantially increase the latency of index search and update. As an alternative lexicon structure we considered a simple sorted table (called *indextable*) that fully resides in memory. For each range or frequent term, the indextable uses an entry to store the locations of the postings across the memory and disk. For terms within a range, the indextable plays the role of a sparse structure that only approximately specifies their position through the range location. For every terabyte of indexed dataset, the indextable along with the auxiliary structures occupy memory space in the order of few tens of megabytes. Therefore, the memory configuration of a typical server makes the indextable an affordable approach to build an efficient lexicon. We explain in detail the indextable structure at Section 4.7.

4.3 The Selective Range Flush Method

We call *posting memory* the space of capacity M_p that we reserve in main memory to temporarily accumulate the postings from new documents. When it gets full, we need to flush postings from memory to disk. We consider a term *short* or *long*, if it respectively occupies total space up to or higher than the parameter *term threshold* T_t. For conciseness, we also use the name short or long to identify the postings and inverted lists of a corresponding term.

Initially all terms are short, grouped into ranges, and transferred to disk via merging. Whenever during a range merge the posting space of a term exceeds the threshold T_t, we permanently categorize the term as long and move all its postings into a new termblock. Any subsequent flushing of new postings for a long term is simply an append to a termblock on disk (Section 4.7). We still need to determine the particular ranges and long terms that we will flush to disk when memory gets full. Long postings incur an one-time flushing cost, while short ranges require repetitive disk reads and writes for

Algorithm 4.1 Pseudocode of SELECTIVE RANGE FLUSH

```
1: Sort long terms by memory space of postings
2: Sort ranges by memory space of postings
3: while (flushed memory space < M_f) do
4:     T := long term of max memory space
5:     R := range of max memory space
6:     // Compare T and R by memory space of postings
7:     if (R.mem_postings < F_p × T.mem_postings ) then
8:         // Append postings of T to on-disk index
9:         if (append overflows the last termblock of list) then
10:            Allocate new termblocks (relocate the list if needed)
11:        end if
12:        Append memory postings to termblocks
13:        Delete the postings of T from memory
14:    else
15:        // Merge postings of R with on-disk index
16:        Read the lists from the rangeblock of R
17:        Merge the lists with new memory postings
18:        if (list size of term w > T_t) then
19:            Categorize term w as long
20:            Move the inverted list of w to new exclusive termblock
21:        end if
22:        if (rangeblock overflows) then
23:            Allocate new rangeblocks
24:            Split merged lists equally across rangeblocks
25:        else
26:            Store merged lists on rangeblock
27:        end if
28:        Delete the postings of R from memory
29:    end if
30: end while
```

flushing over time. From an I/O efficiency point of view, we prefer to only perform a few large in-place appends and totally avoid merges or small appends. Although writes appear to occur asynchronously and return almost instantly, they often incur increased latency during subsequent disk reads due to the cleaning delays of dirty buffers [9].

For efficient memory flushing, next we introduce the *Selective Range Flush* method (Algorithm 4.1). We maintain the long terms and the term ranges sorted by the space their postings occupy in memory (Lines 1-2). We compare the memory space (bytes) of the largest long list against that of the largest range (Line 7). Subsequently, we flush

the largest long list (Lines 8-13), unless its postings are F_p times fewer than those of the respective range, in which case we flush the range (Lines 15-28). We repeat the above process until *flushed memory* (M_f) bytes of postings are flushed to disk. Our approach generalizes a previous method of partial memory flushing [22] in two ways:

(i) We avoid inefficiently flushing the entire posting memory because we only move to disk M_f bytes per memory fill-up.

(ii) In addition to long terms we also selectively flush ranges, when their size becomes sufficiently large with respect to that of long terms.

The constant F_p is a fixed configuration parameter that we call *preference factor*. Its choice reflects our preference for the one-time flushing cost of a long list rather than the repetitive transfers between memory and disk of a range. We flush a range only when the size of the largest long list becomes F_p times smaller. Then the flushing overhead of the long list takes too much for the amount of data flushed. We also prefer to keep the short postings in memory and avoid their merging into disk. The parameter F_p may depend on the performance characteristics of the system architecture, such as the head-movement overhead, the sequential throughput of the disk, and the statistics of the indexed document collection, such as the frequency of terms across the documents. We summarize the parameters of SRF in Table 4.2.

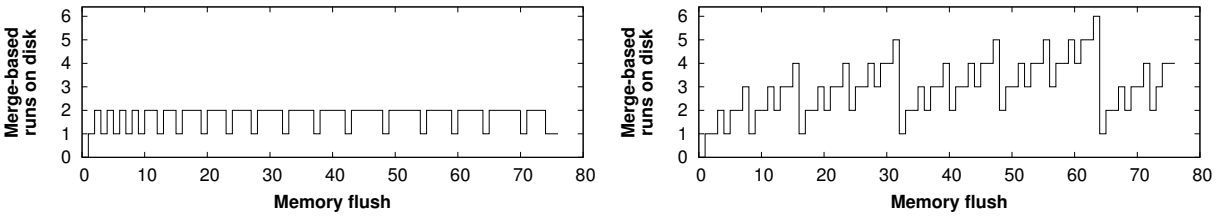
The SRF method behaves greedily because it only considers the memory space occupied by a range or long term, and simply estimates the flushing cost of a range as F_p times that of a long term. We extensively experimented with alternative or complementary flushing rules, including:

(i) directly estimating the disk I/O throughput of ranges and long terms to prioritize their flushing,

(ii) aggressively flushing the long terms with memory space exceeding a minimum limit to exploit the append I/O efficiency,

(iii) flushing the ranges with fewest postings currently on disk to reduce the merging cost,

(iv) periodically flushing the ranges or long terms with low rate of posting accumulation.



(a) HSM (b) HLM

Figure 4.1: We index 426GB using Wumpus with 1GB memory. The x axis refers to the time instances at which memory contents are flushed to disk. (a) HSM maintains up to 2 merge-based runs on disk, and (b) HLM periodically merges the runs created on disk so that their number is logarithmic in the current size of the on-disk index.

In the context of the SRF algorithm, the approach of selectively flushing a few tens of megabytes from the largest terms or ranges in memory performed more robustly overall.

4.4 Evaluation of Selective Range Flush

In this section, we study the behavior of SRF as implemented in our prototype (Proteus) against alternative configurations of the Wumpus search engine [113]. We consider a subset of hybrid merge-based index maintenance methods that are known to cover a wide range of tradeoffs between index building and search efficiency (Table 3.1), and compare them with SRF. We explain in detail the architecture of our prototype in Section 4.7, and describe the experimentation environment and the configuration used for Proteus and Wumpus to ensure systems are functionally comparable in Section 5.2. We use the full 426GB GOV2 text dataset [103] and give both systems 1GB of RAM for the gathering of postings in memory. The parameters of SRF are set according to Table 4.2.

As explained in Section 3.2, hybrid methods separate terms into short and long according to the size of their inverted list. They improve the indexing performance using a merge-based method for the short terms and in-place appends for the long ones. Hybrid Immediate Merge (HIM) uses the Immediate Merge method described in Section 2.1.2 for the short terms, storing the list of each short term contiguously in 1 merge-based run

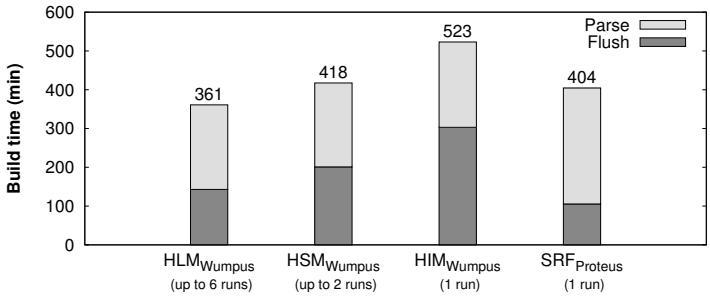


Figure 4.2: We break down the index building time into document parsing and postings flushing parts across different maintenance policies. Parsing includes the time required to clean dirty pages from page cache to free space for newly read documents. Proteus parsing performance is pessimistic as it uses an unoptimized implementation (Section 5.1). We also include the number of merge-based runs each method maintains. SRF has lower time than HIM and HSM, and only 12% higher build time than HLM, even though it maintains contiguously all lists on disk.

and the list of each long term in 1 in-place and 1 merge-based run[1]. Hybrid Square Root Merge (HSM) maintains the merge-based index using the Square Root Merge method discussed in Section 3.2, which keeps each short term in at most 2 merge-based runs and each long term in 1 in-place and in at most 2 merge-based runs (Figure 4.1a). Similarly, Hybrid Logarithmic Merge (HLM) uses the Logarithmic Merge approach explained in Section 2.1.2 for the merge-based part of the index. For the GOV2 dataset, it stores the list of a short term in up to 6 runs and the list of a long term in 1 in-place and up to 6 merge-based runs (Figure 4.1b).

We now study the build time of the methods mentioned above as implemented in Wumpus, and compare them to SRF as implemented in Proteus. From Figure 4.2, as expected, the more runs a method maintains on disk the lower build time it achieves from less frequent merges. On the other hand, the storage fragmentation introduced can significantly impact the list retrieval times. For example, having a short list fragmented in k runs typically causes a k-fold increase on the time required to fetch the list from disk due to disk access overhead. Given also that more than 99% of lists are short and that more

[1]We use the *non-contiguous* variations of the hybrid methods in Wumpus, which store each long list in both the in-place index and the merge-based runs. This reduces the indexing time, but may slightly increase the retrieval time of long lists (Section 3.2).

Table 4.1: Sensitivity to interactions between rangeblock size B_r and preference factor F_p. We underline the lowest measurement on each row. The highest measured time is 62.18min, i.e., 53.8% higher than the lowest 40.43min.

Index Building Time (min) of Selective Range Flush

	F_p						
B_r (MB)	5	10	20	40	80	160	max inc
8	42.83	<u>42.57</u>	42.83	43.55	44.97	47.52	+11.6%
16	42.58	41.63	<u>41.22</u>	41.48	42.08	43.28	+5.0%
32	43.42	41.68	41.38	<u>40.43</u>	40.73	41.18	+7.4%
64	46.90	42.85	41.77	<u>41.02</u>	41.15	41.28	+14.3%
128	51.77	46.82	43.73	41.87	41.75	<u>41.52</u>	+24.7%
256	62.18	53.28	46.75	43.57	43.13	<u>42.40</u>	+46.7%
max inc	+46.0%	+28.0%	+13.4%	+7.7%	+10.4%	+15.4%	

than 60% of query latency is spent on list retrieval (Section 3.3), it immediately follows that fragmentation can cause a substantial increase in query latency. SRF contiguously stores the postings of each term in a single rangeblock or in a number of successive termblocks[2]. Nevertheless, it has 23% lower build time and 65% lower flush time than HIM which also keeps lists contiguous. It is worth pointing out that SRF has even lower build time than HSM and only 12% higher than HLM, even though these methods may need up to 2 and 6 disk accesses respectively to fetch a list from disk regardless of the list length.

4.5 Sensitivity of Selective Range Flush

SRF combines low indexing time with list contiguity on disk, but also has several shortcomings.

- First, if the statistics of a processed dataset change over time, it is possible that a term categorized as long reserves some memory space but then stops accumulating new postings to trigger flushing.

[2]We use the CNT maintenance approach for the long lists that keeps them always contiguous (Section 4.7)

- Second, in order for SRF to behave efficiently across different datasets, it requires tuning of several parameters and their interactions for specific datasets or systems [22]. For example, the optimal preference factor F_p and term threshold T_t may vary across different term frequencies, or interact in complex ways with other system parameters such as the rangeblock size B_r.

- Third, as the dataset processing progresses, the number of ranges increases due to rangeblock overflows; consequently, the number of memory postings per range decreases, leading to lower flushing efficiency.

In Table 4.1 we examine the variation of the SRF index building time across all possible combinations of 7 values for B_r and 6 for F_p (42 pairs in total). Due to the large number of experiments involved, we limit the indexing to the first 50GB of GOV2. The elements in the last row and column of the table report the largest increase of build time with respect to the minimum measurement of the respective column and row. We notice that as B_r increases, e.g., due to restrictions from the filesystem [48, 95], we have to tune the preference factor to retain low build time. Otherwise, the build time may increase as high as 47% with respect to the lowest measurement for a specific B_r value.[3] The respective increase of the highest measured value to the lowest in the entire table is 53.8%. After a large number of experiments across different combinations of parameters, we identified as default values for build and search efficiency those specified in Table 4.2 (see also Sections 4.5 and 5.4).

4.6 The Unified Range Flush Method

In order to facilitate the practical application of SRF, we evolved it to a new method that we call *Unified Range Flush* (*URF*). In this method, we assign each memory posting to the lexicographic range of the respective term without the categorization as short or long. Thus, we omit the term threshold T_t and preference factor F_p of SRF along with their

[3]Note that large B_r values such as 128MB and 256MB are not uncommon in such systems. For example, Hadoop [95] uses block sizes of 128MB, while Bigtable [28] uses files of 100-200 MB to store its data (Section 2.2.1). Moreover, even for B_r=256MB, a proper tuning of F_p can keep the build time within 6% of the total lowest time (42.40min versus 40.43min).

Table 4.2: Parameters of Selective Range Flush (SRF) and Unified Range Flush (URF). In the last column we include their default values used in our prototype.

Symbol	**Name**	**Description**	**Value**
B_r	Rangeblock	Byte size of rangeblock on disk	32MB
B_t	Termblock	Byte size of termblock on disk	2MB
M_p	Posting Memory	Total memory for accumulating postings	1GB
M_f	Flushed Memory	Bytes flushed to disk each time memory gets full	20MB
F_p	Preference Factor	Preference to flush short or long terms by SRF	20
T_t	Term Threshold	Term categorization into short or long by SRF	1MB
T_a	Append Threshold	Amount of postings flushed to termblock by URF	256KB

interactions against other parameters. When the posting memory gets full, we always pick the range with the most postings currently in memory and merge it to disk including the terms that SRF would normally handle as long. In order to reduce the data volume of merge, we introduce the *append threshold* (T_a) parameter. If the postings of a term in a merge occupy memory space more than T_a, we move them (*append postings*) from the rangeblock to an exclusive termblock. Subsequently, the range continues to accumulate the new postings of the term in the rangeblock, until their amount reaches the number T_a again (Figure 3.1).

The pseudocode of URF is shown in Algorithm 4.2. In comparison to SRF, it is quite simpler because we skip the distinct handling of short and long terms. Algorithm 4.2 simply identifies the range with the most postings in memory at line 3 and merges it with the corresponding rangeblock on disk at lines 5-6 and 14-20. If there are terms whose amount of postings exceed the threshold T_a, URF flushes them to their corresponding termblocks at lines 7-13.

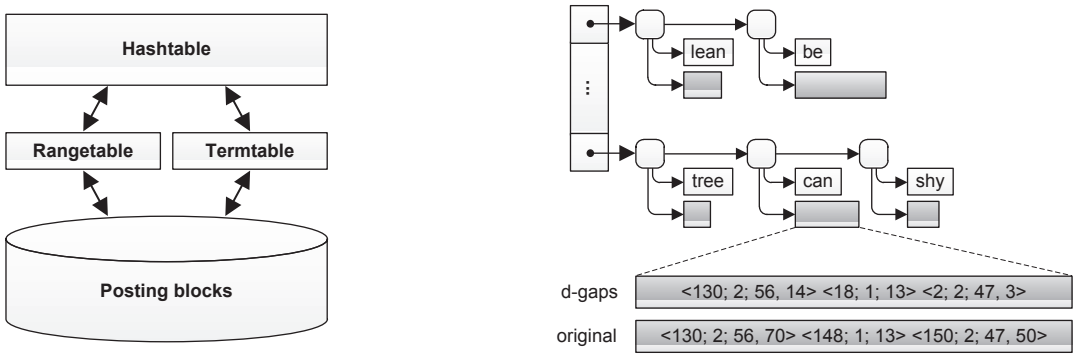
The term threshold T_t of SRF permanently categorizes a term as long and prevents it from merge-based flushing, even at low amount of posting memory occupied by the term. Additonally, it depends on the characteristics of the dataset and it interacts with other system parameters such as the preference factor F_p, since it implicitly controls the size of the long lists flushed. The general approach of dynamic threshold adjustment followed by previous research would only further complicate the method operation (e.g., in Section 5.2 we examine the automated threshold adjustment $\theta_{PF} = \text{auto}$ [22]). Instead,

Algorithm 4.2 Pseudocode of UNIFIED RANGE FLUSH

```
1: Sort ranges by total memory space of postings
2: while (flushed memory space < M_f) do
3:     R := range of max memory space
4:     // Merge postings of R with on-disk index
5:     Read the inverted lists from the rangeblock of R
6:     Merge the lists with new memory postings
7:     if (list size of term w > T_a) then
8:         // Move postings of w to exclusive termblock
9:         if (w does not have termblock or append will overflow last termblock) then
10:            Allocate new termblocks (relocate list, if needed)
11:        end if
12:        Append memory postings to termblocks
13:    end if
14:    if (rangeblock overflows) then
15:        Allocate new rangeblocks
16:        Split merged lists equally across rangeblocks
17:    else
18:        Store merged lists on rangeblock
19:    end if
20:    Remove the postings of R from memory
21: end while
```

the parameter T_a controls the disk efficiency of the append operation and primarily depends on performance characteristics of the I/O subsystem, such as the geometry of the disk. Thus, it can be tuned independently of the dataset characteristics and other system parameters.

The description of T_a bears some similarity to the definition of long-term threshold T introduced previously [24]. However, the URF algorithm has fundamental differences from the hybrid approach of Büttcher et al. First, every invocation of hybrid merge flushes all the postings currently gathered in memory. Instead, we only flush the largest ranges with total amount of postings in memory at least M_f. Second, the choice of T_a only affects the efficiency of the index building process, because we separately control the search efficiency through the termblock size B_t (Sections 4.7, 5.5). In contrast, T determines the storage fragmentation of each long list; choosing small T improves the update performance but reduces the efficiency of query processing. Indeed, we experimentally found that it is possible to achieve lowest building time and search latency for T_a around 128KB–256KB,



(a) Proteus architecture

(b) Hashtable

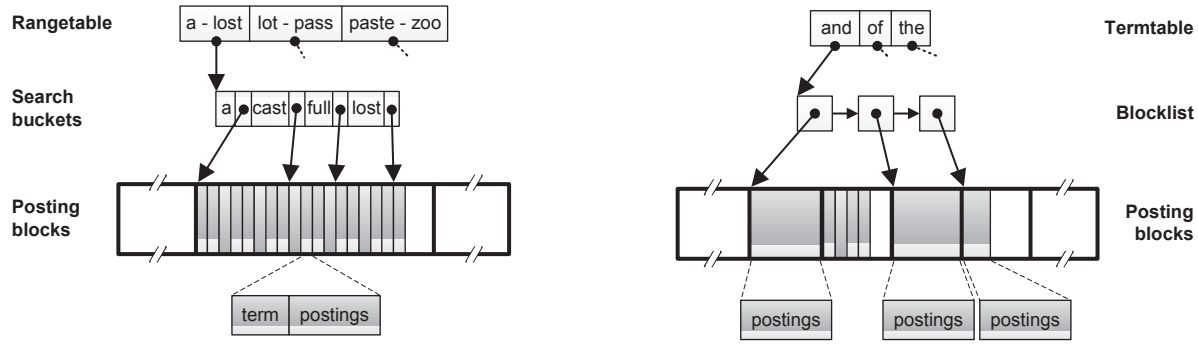
Figure 4.3: (a) The prototype implementation of *Proteus*. (b) We maintain the hashtable in memory to keep track of the postings that we have not yet flushed to disk.

but such small values for T would significantly lower query processing performance due to the excessive fragmentation in long lists.

We summarize the parameters of our architecture in Table 4.2. In the following sections, we show that URF in comparison to SRF (i) has fewer parameters and lower sensitivity to their values, (ii) has similar index maintenance performance (or better over a large dataset) and search performance, and (iii) has more tractable behavior that allows us to do complexity analysis of index building in Section 9.1.

4.7 Prototype Implementation

The *Proteus* system is a prototype implementation that we developed to investigate our inverted-file management (Figure 4.3a). We retained the parsing and search components of the open-source Zettair search engine (v0.9.3) [115]. Unlike the original implementation of Zettair that builds a lexicon for search at the end of index building, we dynamically maintain the lexicon in Proteus throughout the building process. We store the postings extracted from the parsed documents in a memory-based hash table that we call *hashtable* (Figure 4.3b). The inverted list of each term consists of the document identifiers along with the corresponding term locations in ascending order (Section 2.1.1). We store each list as an initial document identifier followed by a list of gaps compressed



(a) Ranges and rangetable

(b) Long terms and termtable

Figure 4.4: (a) Each entry of the rangetable corresponds to a term range, and points to the search bucket, which serves as partial index of the corresponding rangeblock. (b) Each entry of the termtable corresponds to a term and points to the blocklist that keeps track of the associated termblocks on disk.

with variable-length byte-aligned encoding [118]. The same compression scheme is used to store the locations within each document. Compression reduces considerably the space requirements of postings across memory and disk.

We keep track of the term ranges in a memory-based sorted array that we call *rangetable* (Figure 4.4a). Each entry corresponds to the range of a single rangeblock and contains the space size of the disk postings along with the names of the first and last term in the range. In a sparse index that we call *search bucket* we maintain the name and location of the term that occurs every 128KB along each rangeblock. The search bucket allows us to only retrieve the exact 128KB that may contain a term instead of the entire rangeblock. In our experience, any extra detail in rangeblock indexing tends to significantly increase the maintenance overhead and lookup time without considerable benefits in performance of query evaluation (Section 5.3). We use a sorted array (*termtable*) to keep track of the termblocks that store the long terms of SRF or the append postings of URF respectively (Figure 4.4b). We organize the termtable as an array of *descriptors*. Each descriptor contains the term name, a pointer to the memory postings, their size, the amount of free space at the last termblock on disk, and a linked list of nodes called *blocklist*. Each node contains a pointer to a termblock on disk.

The rangetable along with the termtable together implement the indextable in our

system (Section 4.2). The inverted lists in memory that belong to the same range are connected through a linked list. Initially, the termtable is empty and the rangetable contains a single range that covers all possible terms. If the inverted lists after a merge exceed the capacity of the respective rangeblock, we split the range into multiple half-filled rangeblocks. Similarly, if we exceed the capacity of the last termblock, we allocate new termblocks and fill them up. After a flush, we update the tables to accurately reflect the postings that they currently hold. When the capacity of a termblock is exceeded, we allocate a new termblock following one of three alternative approaches:

1. *Fragmented (FRG)*: Allocate a new termblock of size B_t to store the overflown postings.

2. *Doubling (DBL)*: Allocate a new termblock of twice the current size to store the new postings of the list.

3. *Contiguous (CNT)*: Allocate a termblock of twice the current size and relocate the entire list to the new termblock to keep the list contiguous on disk. This is our default setting.

For a term, the DBL allocation leads to number of termblocks that is logarithmic with respect to the number of postings, while FRG makes it linear. In our evaluation, we consider the implementation of the above approaches over Proteus for both SRF and URF (Section 5.5).

4.7.1 Memory Management and I/O

For every inverted list in memory, the hashtable stores into a *posting descriptor* information about the inverted list along with pointers to the term string and the list itself (Figure 4.3b). For the postings of the inverted list, we allocate a simple byte array whose size is doubled every time it fills up. When an inverted list is flushed to disk, we free the respective posting descriptor, term string and byte array. The efficiency of these memory (de-)allocations is crucial for the system performance because they are invoked extremely often.

Initially, we relied on the standard `libc` library for the memory management of the inverted lists. On allocation, the library traverses a list of free memory blocks (*free list*)

to find a large enough block. On deallocation, the freed block is put back into the free list and merged with adjacent free blocks to reduce external fragmentation. We refer to this scheme as *default*.

If a program runs for long time and uses a large amount of memory, the free list becomes long and the memory fragmented, increasing the management cost. In order to handle this issue, we use a single call to allocate both the descriptor and term, or accordingly to deallocate them after an inverted list is flushed to disk. The byte array is not included in the above optimization, because we cannot selectively free or reallocate portions of an allocated chunk every time we double the array size. We refer to this scheme as *single-call*.

We further reduce the management cost by using a single call to get a memory chunk (typically 4KB) and store there all the posting descriptors and term strings of a range. In a chunk, we allocate objects (strings and descriptors) in a stack-like manner. Processor cache locality is also improved when we store together the objects of each range. If the current chunk has insufficient remaining space, we allocate an object from a new chunk that we link to the current one. When we flush a range to disk we traverse its chunk list and free all chunks, consequently freeing all term strings and descriptors of the range. We refer to this scheme as *chunkstack*.

In our prototype system, we store the disk-based index over the default filesystem. Hence, we cannot guarantee the physical contiguity of disk files that are incrementally created and extended over time. Disk blocks are allocated on demand as new data is written to a storage volume leading to file fragmentation across the physical storage space. To prevent the file fragmentation caused by the system, we examined to use the *preallocation* of index files.

Index building includes document parsing, which reads documents from disk to memory (I/O-intensive) and then processes these documents into postings (CPU-intensive). During the processing of a part of the dataset, *prefetching* allows the system to fetch in advance the next part of the dataset –during the processing of the current part– to prevent the blocking of subsequent reads to the disk [82]. We evaluate all the above approaches of memory management and I/O optimization in Section 5.5.

Chapter 5

Performance Evaluation of Incremental Text Indexing

5.1 Experimentation Environment

5.2 Building the Inverted File

5.3 Query Handling

5.4 Sensitivity of Unified Range Flush

5.5 Storage and Memory Management

5.6 Scalability across Different Datasets

5.7 Summary

In this section, we compare the index build and search performance across a representative collection of methods (from Table 3.1) over Wumpus and Proteus. In our experiments we include the performance sensitivity of the URF method across several configuration parameters, storage and memory allocation techniques, and other I/O optimizations. We also explore the relative build performance of SRF and URF over different datasets.

5.1 Experimentation Environment

We execute our experiments on servers running the Debian distribution of Linux kernel (v2.6.18). Each server is equipped with one quad-core x86 2.33GHz processor, 3GB RAM and two SATA disks. We store the generated index and the document collection on two different disks over the Linux ext3 filesystem. Different repetitions of an experiment on the same server lead to negligible measurement variations (<1%).

We mostly use the full 426GB GOV2 standard dataset from the TREC Terabyte track [103]. Additionally, we examine the scalability properties of our methods with the 200GB dataset from Wikipedia [111], and the first 1TB of the ClueWeb09 dataset from CMU [31]. We mainly use 7200RPM disks of 500GB capacity, 16MB cache, 9-9.25ms seek time, and 72-105MB/s sustained transfer rate. In some experiments (ClueWeb, Section 5.6), we store the data on a 7200RPM SATA disk of 2TB capacity, 64MB cache, and 138MB/s sustained transfer rate.

We use the latest public code of Wumpus [113], and set the threshold T equal to 1MB, as suggested for a reasonable balance between update and query performance. We measure the build performance of HIM in Wumpus with activated partial flushing and automated threshold adjustment [22]. In both systems we set M_p = 1GB. In Proteus, unless otherwise specified, we set the parameter values B_t = 2MB, B_r = 32MB, M_f = 20MB, F_p = 20, T_t = 1MB and T_a = 256KB (Table 4.2, Sections 4.5, 5.4). The auxiliary structures of URF and SRF for GOV2 in Proteus occupy less than 42MB in main memory. In particular, with URF (SRF) we found the hashtable to occupy 4MB, the termtable and rangetable together 0.5MB, the blocklists 0.25MB (0.12MB), and the range buckets 31.2MB (36.5MB).

To keep Wumpus and Proteus functionally comparable, we activate full stemming across both systems (Porter's option [85]). Full stemming reduces terms to their root form through suffix stripping. As a result, document parsing generates smaller index and takes longer time; also query processing often takes more time due to the longer lists of some terms, and matches approximately the searched terms over the indexed documents. In Proteus we use an unoptimized version of Porter's algorithm as implemented in Zettair. This makes the parsing performance of Proteus pessimistic and amenable to further optimizations. When we examine the performance sensitivity of Proteus to configuration

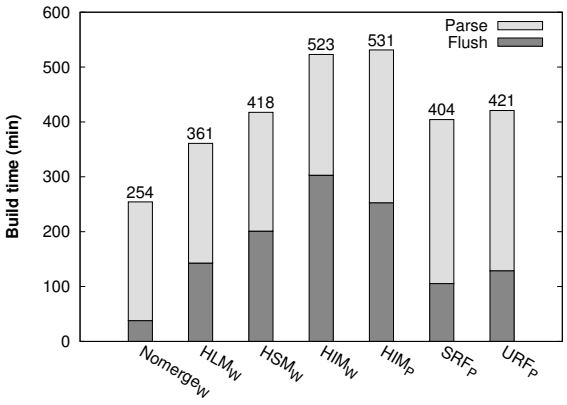


Figure 5.1: We consider the index building time for different indexing methods across Wumpus and Proteus, both with full stemming. Over Wumpus, we examine Nomerge (Nomerge$_W$), Hybrid Logarithmic Merge (HLM$_W$), Hybrid Square Root Merge (HSM$_W$) and Hybrid Immediate Merge (HIM$_W$). Over Proteus, we include Hybrid Immediate Merge (HIM$_P$), Selective Range Flush (SRF$_P$) and Unified Range Flush (URF$_P$). URF$_P$ takes 421min to process the 426GB of GOV2 achieving roughly 1GB/min indexing throughput (see also Figure 5.7 for other datasets).

parameters, we use a less aggressive option called *light stemming*, which is the default in Zettair.

5.2 Building the Inverted File

First we examine the build time of several methods implemented over Wumpus and Proteus. According to Section 4.4, after indexing the full GOV2 dataset Hybrid Immediate Merge (HIM) keeps each short term in 1 merge-based run, and each long term in 1 in-place and 1 merge-based run. Hybrid Square Root Merge (HSM) keeps each short term in 2 merge-based runs, and each long term in 1 in-place and 2 merge-based runs. Hybrid Logarithmic Merge (HLM) has each short term over 4 merge-based runs, and each long term over 1 in-place run and 4 merge-based runs. Nomerge fragments the postings across 42 runs. SRF maintains the postings of each term in a unique rangeblock or termblock, while URF keeps each infrequent term in 1 rangeblock and each frequent term in up to 1 rangeblock and 1 termblock.

In Figure 5.1 we consider the build time of the methods Nomerge$_W$, HLM$_W$, HSM$_W$ and HIM$_W$ in Wumpus, and the methods HIM$_P$, SRF$_P$, and URF$_P$ as we implemented them in Proteus. HIM$_W$ is the contiguous version of HIM (HIM$_C$ [22], Section 3.2, Section 9.1) with all the applicable optimizations and the lowest build time among the Wumpus variations of HIM as we experimentally verified. According to the Wumpus implementation of contiguous and non-contiguous methods, the postings of a long term are dynamically relocated to ensure storage on multiple segments of size up to 64MB each [113]. In order to ensure a fair comparison of different methods on the same platform, we also implemented HIM$_C$ in Proteus (HIM$_P$) with the CNT storage allocation by default. The index size varied from 69GB for URF$_P$ down to 60GB for SRF$_P$ and HIM$_P$, due to about 10GB difference in the empty space within the respective disk files.

The Wumpus methods take between 254min (baseline Nomerge$_W$) and 523min (HIM$_W$). HSM$_W$ and HLM$_W$ reduce the time of HIM$_W$ by 20% and 31% respectively, but they fragment the merge-based index across 2 and 6 runs (Figure 4.1). This behavior is known to substantially increase the I/O time of query processing, and consequently we do not consider HSM$_W$ and HLM$_W$ any further [22, 73]. The 531min of HIM$_P$ is comparable to the 523min required by HIM$_W$; in part, this observation validates our HIM implementation over Proteus. Instead, SRF$_P$ and URF$_P$ take 404min and 421min, respectively, which is 24% and 21% below HIM$_P$. URF$_P$ takes 4.2% more than SRF$_P$ to incrementally index GOV2, although URF$_P$ is faster than SRF$_P$ in a different dataset (Section 5.6).

In Figure 5.1, we also break down the build time into *parse*, to read the datasets and parse them into postings, and *flush*, to gather the postings and transfer them to disk. The implementation of HIM in Proteus (HIM$_P$) reduces the flush time of the corresponding implementation in Wumpus (HIM$_W$) from 303min to 253min, but HIM$_P$ has longer parse time partly due to the unoptimized stemming. Instead, SRF$_P$ and URF$_P$ only take 105min and 129min for flushing, respectively, thanks to their I/O efficiency. Therefore, our methods reduce the flush time of HIM$_P$ by a factor of 2.0-2.4, and that of HIM$_W$ by a factor of 2.4-2.9.

Somewhat puzzled by the longer parse time of Proteus, we recorded traces of disk transfer activity during index building. Every time we retrieved new documents for processing, we noticed substantial write activity with tens of several megabytes transferred to the index disk. Normally, parsing should only create read activity to retrieve docu-

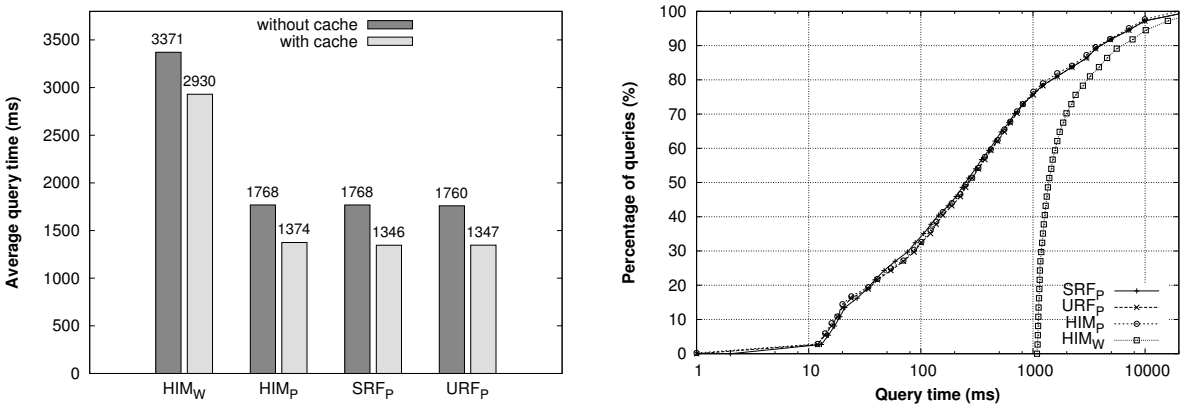
ments and no write activity at all. However, when we flush index postings to disk, the system temporarily copies postings to the system buffer cache. In order to accommodate new documents in memory later during parsing, read requests clean dirty buffers and free memory space. Overall, SRF_P and URF_P reduce by about a factor of 2-3 the flush time of HIM_P and HIM_W, and achieve a reduction of the respective total build time by 20-24%.

5.3 Query Handling

Next we examine the query time across different indexing methods and systems. In our experiments, we use the GOV2 dataset and the first 1,000 queries of the Efficiency Topics query set in the TREC 2005 Terabyte Track [103]. We consider both the alternative cases of having the buffer cache disabled and enabled during query handling. As representative method of Wumpus we study the HIM_W, while in Proteus we consider HIM_P, SRF_P and URF_P.

In the latest publicly available version of Wumpus (but also the older versions), we noticed that the implemented contiguous variation (HIM_C) of HIM was constantly crashing during search at a broken assertion. For that reason, in our search experiments we used the non-contiguous variation instead (HIM_{NC} [24]). Although the above two Wumpus variations of HIM differ in their efficiency of index building, they have similar design with respect to query handling. They both store each short term in 1 run; however, HIM_{NC} allows a long term to be stored in 2 runs (1 in-place and 1 merge-based), while HIM_C always stores it in 1 run. Given the long transfer time involved in the retrieval I/O of long terms, we do not expect the above difference by 1 disk positioning overhead to practically affect the query performance.

From Figure 5.2a, caching activation reduces the query time of HIM_W by 13%, and by about 22-24% that of HIM_P, SRF_P and URF_P. Across both the caching scenarios, HIM_W over Wumpus takes about twice the average query time of the Proteus methods. Given that our HIM_P implementation is based on the published description of HIM_W, we attribute this discrepancy to issues orthogonal to the indexing method, such as the query handling and storage management of the search engine. In Proteus, the average query times of HIM_P, SRF_P and URF_P remain within 2% of each other. Therefore, both SRF_P



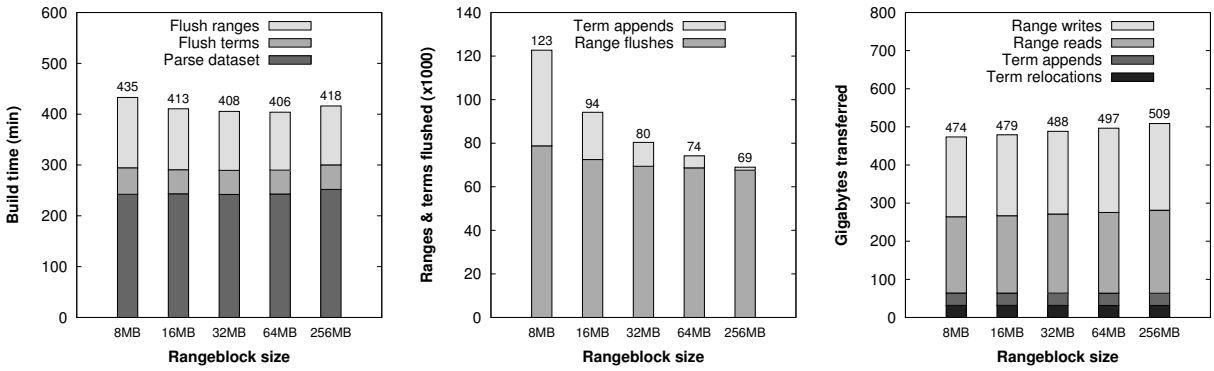
(a) Average Query Time

(b) Query Time Distribution

Figure 5.2: We consider Hybrid Immediate Merge over Wumpus (HIM_W) or Proteus (HIM_P), along with Selective Range Flush (SRF_P) and Unified Range Flush (URF_P) over Proteus. (a) We measure the average query time with alternatively disabled and enabled the system buffer cache across different queries in the two systems with full stemming. (b) We look at the distribution of query time over the two systems with enabled the buffer cache.

and URF_P achieve the query performance of HIM_P, even though they are considerably more efficient in index building (Section 5.2).

We use measurement distributions to further compare the query time of the four methods with enabled system caching (Figure 5.2b). Although the median query time of Proteus lies in the range 246-272ms, that of HIM_W is 1.378s, i.e., a factor of 5 higher. In fact, HIM_W requires about 1s to handle even the shortest queries. Also, the 99th percentile of HIM_W is 68% higher than that of the Proteus methods. Instead, the 99th percentiles of the Proteus methods lie within 2% each other, while the median measurements within 10%. We conclude that HIM_P, URF_P and SRF_P are similar to each other in query performance, but they are faster by a factor of 2 on average with respect to HIM_W.



(a) Index Building (b) Number of Flushes (c) I/O Activity

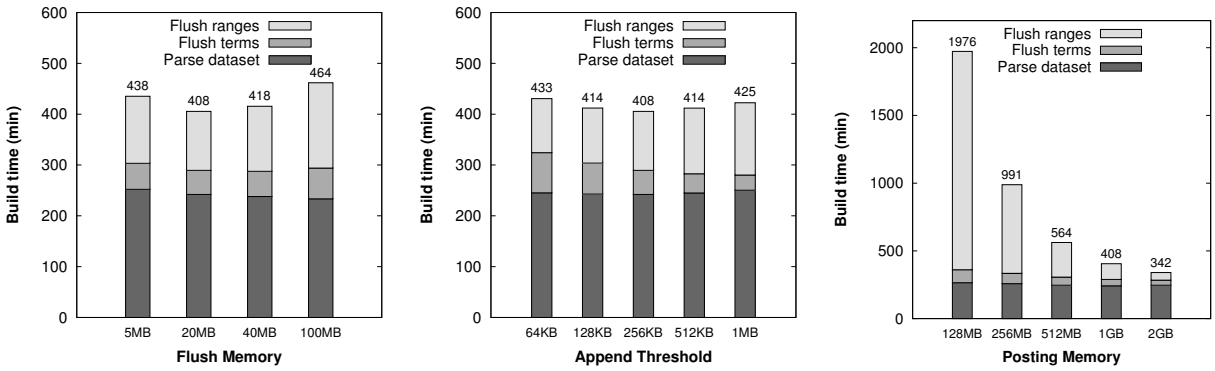
Figure 5.3: (a) Setting the rangeblock size B_r below 32MB or above 64MB raises the build time of Unified Range Flush. Increasing the B_r tends to (b) decrease the number of flushes, and (c) increase the data amount transferred during merges. We use Proteus with light stemming.

5.4 Sensitivity of Unified Range Flush

Subsequently, we consider the sensitivity of the URF build performance to the rangeblock size B_r, flush memory M_f, append threshold T_a, and posting memory M_p.

Rangeblock B_r. The rangeblock size B_r determines the posting capacity of a range; it directly affects the data amount transferred during range flushes and the I/O time spent across range and term flushes. We observed the lowest build time for B_r at 32-64MB (Figure 5.3a). Setting B_r less than 32MB generates more ranges, and raises the total number of term and range flushes (Figure 5.3b). On the contrary, setting B_r higher than 64MB increases the amount of transferred data during range merges (Figure 5.3c) leading to longer I/O. Our default value B_r = 32MB balances the above two trends into build time equal to 408min. For sensitivity comparison with SRF, we also measured the URF build time for the first 50GB of GOV2. With B_r in the interval 8MB-256MB, we found the maximum increase in build time equal to 9.1%, i.e., almost x6 times lower than the respective 53.8% of SRF (Table 4.1).

As we continue to increase the rangeblock size, we partition the index into fewer rangeblocks. Each range merge thus flushes more postings and updates a bigger part of the index. Accordingly, index updates become less "selective" and eventually approach the



(a) Sensitivity to M_f (b) Sensitivity to T_a (c) Sensitivity to M_p

Figure 5.4: (a) Flushing more than few tens of megabytes (M_f) leads to longer build time for Unified Range Flush (URF). This results from the more intense I/O activity across term and range flushes. (b) Setting the append threshold to T_a = 256KB minimizes the total I/O time of range and term flushes. (c) The build time of range merge in URF decreases approximately in proportion to the increasing size of posting memory (M_p). The Proteus system with light stemming is used.

HIM behavior, which flushes all short postings and merges the whole merge-based index on each memory flush. Setting $B_r = \infty$ would practically emulate the HIM method; for example, with B_r = 1GB (not shown) we measured 510min build time. This emphasizes the benefits of the *selective* index update approach followed by our methods.

Flush Memory M_f. The parameter M_f refers to the amount of bytes that we flush to disk every time posting memory gets full (Figure 5.4a). Build time is reduced to 408min if we set M_f = 20MB, i.e., 2% of the posting memory M_p=1GB. Despite the Zipfian distribution of postings [24], setting M_f below 20MB leaves limited free space to gather new postings at particular ranges (or terms) for efficient I/O. At M_f much higher than 20MB, we end up flushing terms and ranges with small amounts of new postings leading to frequent head movement in appends and heavy disk traffic in merges. If we set M_f = M_p (= 1GB) we actually deactivate partial flushing, and build time becomes 632min (not shown).

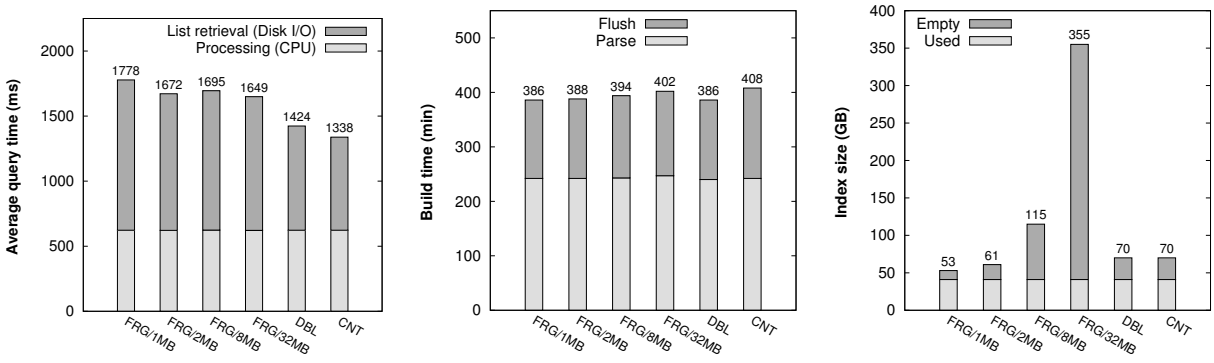
Append Threshold T_a. This parameter specifies the minimum amount of accumulated postings required during a merge to flush a term to the in-place index. It affects directly the efficiency of term appends, and indirectly their relative frequency to range flushes. In

Figure 5.4b we observe that T_a = 256KB minimizes the URF build time. If we increase T_a to 1MB (=T_t) we end up with build time higher by 6%. Unlike T_t of SRF that permanently categorizes a term as long, T_a specifies the minimum append size and tends to create larger merged ranges by including postings that SRF would permanently treat as long instead. Overall, the URF performance shows little sensitivity across reasonable values of T_a.

Posting Memory M_p. The parameter M_p specifies the memory space that we reserve for temporary storage of postings. Smaller values of M_p increase the cost of range flushes, because they enforce frequent range flushes and limit the gathering of postings from frequent terms in memory. As we increase M_p from 128MB to 1GB in Figure 5.4c, the time spent on range merges drops almost proportionally, resulting into substantial decrease of the total build time. Further increase of M_p to 2GB only slightly reduces the build time, because at M_p = 1GB most time (59.3%) is already spent on parsing.

5.5 Storage and Memory Management

Next we examine the effect of storage allocation to the build and query time of URF. Based on the description of Section 4.7, we consider FRG with alternative termblock sizes 1MB, 2MB, 8MB and 32MB (respectively denoted as FRG/1MB, FRG/2MB, FRG/8MB, FRG/32MB), and also the DBL and CNT allocation methods. In Figure 5.5a we show the average CPU and I/O time of query processing in a system with activated buffer cache. The average query time varies from 1649min with FRG/32MB to 1778min with FRG/1MB, while it drops to 1424min by DBL and 1338min by CNT. Essentially, CNT reduces the query time of FRB/1MB by 25% and of DBL by 6%. The above variations are mainly caused by differences in I/O time given that the CPU time remains almost constant at 622min (<47% of total). Unlike query time, from Figure 5.5b we notice the build time to only slightly vary from 386min for both FRG/1MB and DBL to 408min for CNT (5.7% higher). In these measurements, the flush time is about 40% of the total build time. Due to differences in the empty space of termblocks, the index size varies from 53GB for FRG/1MB to 355GB for FRG/32MB, and 70GB for DBL and CNT (Figure 5.5c). We conclude that our CNT default setting is a reasonable choice because it achieves improved



(a) Query Time (b) Build Time (c) Index Size

Figure 5.5: We examine the behavior of Unified Range Flush over Proteus with the following storage allocation methods (i) contiguous (CNT), (ii) doubling (DBL), and (iii) fragmented (FRG) with termblock sizes 1MB, 2MB, 8MB and 32MB. (a) CNT achieves the lowest query time on average closely followed by DBL. We keep enabled the system buffer cache across the different queries. (b) Build time across the different allocation methods varies within 5.7% of 386min (FRG/1MB and DBL). (c) Unlike CNT and DBL, FRG tends to increase the index size especially for larger termblock.

query time at low added build time or index size.

In Section 4.7.1 we mentioned three alternative approaches to manage the memory of postings: (i) default (D), (ii) single-call (S), and (iii) chunkstack (C). The methods differ in terms of function invocation frequency, memory fragmentation and bookkeeping space overhead. Memory allocation affects the time spent on dataset parsing when we add new postings to memory, and the duration of term and range flushes when we remove postings. In Figure 5.6 we consider the three allocation methods with URF across different values of posting memory. Memory management increasingly affects build time as posting memory grows from 512MB to 2GB. More specifically, the transition from the default policy to chunkstack reduces build time by 3.4% for $M_p = 512$MB, 4.7% for $M_p = 1$GB, and 8.6% for $M_p = 2$GB. Therefore, larger amounts of memory space require increased efficiency in memory management to accelerate index building.

In Table 5.1 we compare the effects of several memory and I/O optimizations to the build and search time of SRF. File preallocation of the index lowers by 14-17% the average query time as a result of reduced storage fragmentation at the filesystem level.

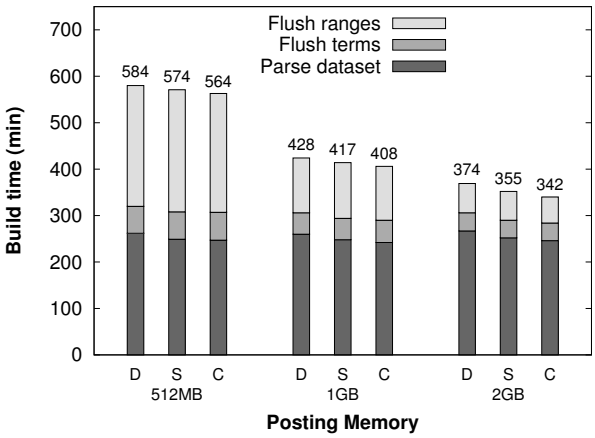


Figure 5.6: We consider three methods of memory allocation during index building by Unified Range Flush: (i) default (D), (ii) single-call (S), and (iii) chunkstack (C). The sensitivity of build time to memory management is higher (up to 8.6% decrease with C) for larger values of M_p. We use Proteus with light stemming.

For aggressive prefetching, we increase the Linux readahead window to 1MB making it equal to the size of the parsing buffer. Thus, during the processing of 1MB text, we fetch in the background the next 1MB from disk. As a result, parse time drops by 30% and the total build time drops by 20% from 534min to 429min. When we activate the chunkstack method in memory management, build time further drops by 5% from 429min to 408min. We have all these optimizations activated throughout the experimentation with Proteus.

5.6 Scalability across Different Datasets

Finally, we measure the total build time of the CNT variants of SRF and URF for three different datasets: ClueWeb09 (first 1TB), GOV2 (426GB) and Wikipedia (200GB). In our evaluation, we use the default parameter values shown in Table 4.2. In Figures 5.7a and 5.7d we break down into parse and flush time the SRF and URF build time for the ClueWeb09 dataset. Even though SRF better balances the flush time of ranges and terms against each other, URF actually reduces the total build time of SRF by 7% from 815min to 762min. This improvement is accompanied by a respective reduction of the total flush time by 82min (23%) from 353min to 271min.

In Figures 5.7b and 5.7e we examine the scaling of build time for the GOV2 dataset.

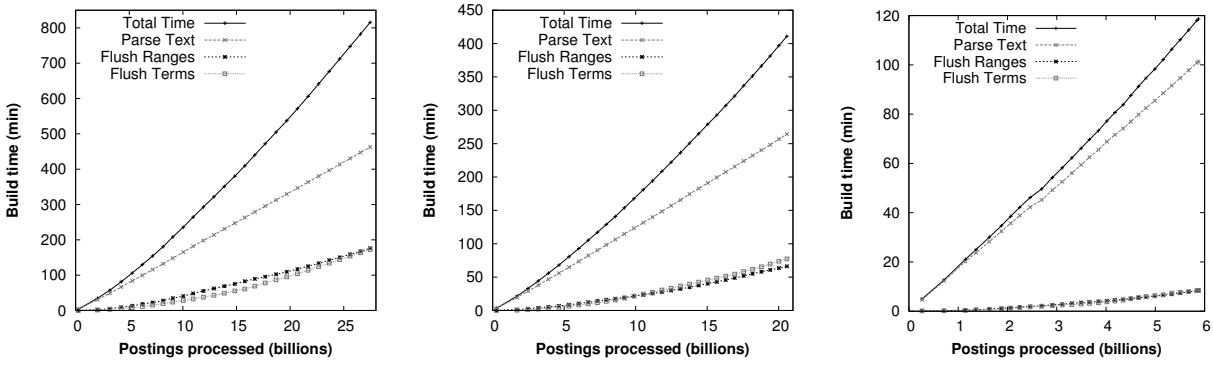
Table 5.1: We examine the effect of alternative optimizations to the query and build time of Unified Range Flush. Preallocation reduces the average query time, while prefetching and chunkstack reduce the build time.

Average Build and Query Time - Unified Range Flush

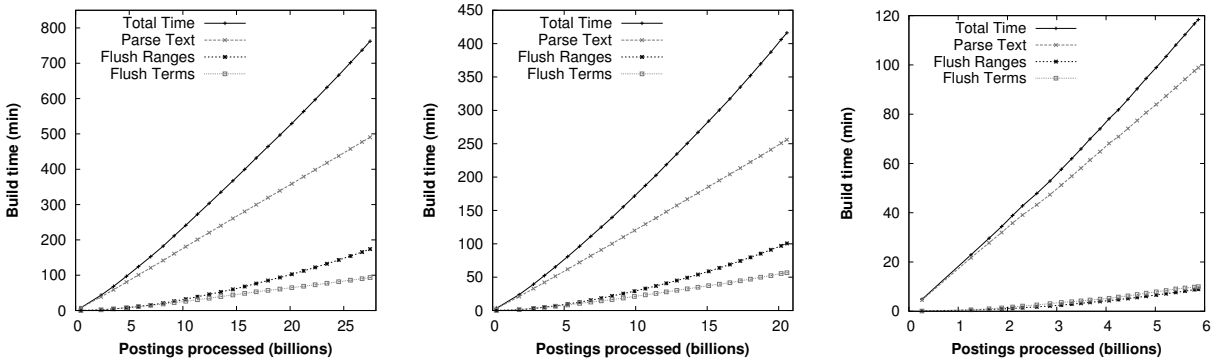
Memory and I/O Optimizations	Total Build (min)	Parse Time (min)	Flush Time		Query	
			Ranges (min)	Terms (min)	W/out (ms)	W/Cache (ms)
None	543	374	124	40	2082	1537
Preallocation	534	373	112	45	1728	1316
Preallocation+Prefetching	429	260	118	47	1724	1318
Preallocation+Prefetching+Chunkstack	408	242	116	48	1726	1315

SRF reduces the build time of URF by 16min (4%) from 420min to 404min. The total number of indexed postings is 20.58bn in GOV2 (426GB) and 27.45bn in ClueWeb09 (1TB). However, GOV2 has about half the text size of ClueWeb09, and the index building of GOV2 takes almost half the time spent for ClueWeb09. In fact, the parsing of GOV2 seems to take more than 70% of the total build time partly due to cleaning of pages written during flushing (Section 5.2). In the Wikipedia dataset, parsing takes about 84-85% of the total build time, but both URF and SRF require the same time (about 118.5min) to build the index (Figures 5.7c and 5.7f).

Across Figure 5.7, the total build time of URF and SRF (e.g., ClueWeb09 and GOV2) demonstrates a nonlinearity mainly caused by the range flush time rather than the parsing and term flushing. We explored this issue by using the least-squares method to approximate the build time of GOV2 as function of the number of postings. In our regression, we alternatively consider the linear function $f(x) = a_1 + b_1 \cdot x$ and the polynomial function $f(x) = a_2 \cdot x^{b_2}$. Using the coefficient of determination R^2 to quantify the goodness of fit, we find that both the total build time and the time of range flushing are accurately tracked by the polynomial function [56]. Instead, the respective times of parsing and term flushing achieve good quality of fit with linear approximation.



(a) SRF - ClueWeb09 (b) SRF - GOV2 (c) SRF - Wikipedia



(d) URF - ClueWeb09 (e) URF - GOV2 (f) URF - Wikipedia

Figure 5.7: We show the scaling of build time with Selective Range Flush (SRF) and Unified Range Flush (URF). We use the ClueWeb09 (first TB), GOV2 (426GB) and Wikipedia (200GB) datasets over Proteus with light stemming. URF takes 53.5min (7%) less time for ClueWeb09, about the same for Wikipedia, and 16.4min (4%) more for GOV2 in comparison to SRF.

5.7 Summary

We investigate the problem of incremental maintenance of a disk-based inverted file. Our objective is to improve both the search latency and index building time at low resource requirements. We propose a simple yet innovative disk organization of inverted files based on blocks, and introduce two new incremental indexing methods, the Selective Range Flush (SRF) and Unified Range Flush (URF). We implemented our two methods in the Proteus prototype that we built. We extensively examine their efficiency and performance robustness using three different datasets of size up to 1TB. SRF requires considerable tuning effort across different parameter combinations to perform well. In

comparison to SRF, URF has similar or even better performance, while it is also simpler, easier to tune and amenable to I/O complexity analysis (Section 9.1).

Both in Proteus and the existing Wumpus system, we experimentally examine the search performance of the known Hybrid Immediate Merge (HIM) method with partial flushing and automatic threshold adjustment. Our two methods achieve the same search latency as HIM in Proteus, while they reduce into half the search latency of HIM in Wumpus. Additionally, our methods reduce by a factor of 2-3 the I/O time of HIM during index building, and lower the total build time by 20% or more.

Chapter 6

Range-Based Storage Management for Scalable Datastores

6.1 Introduction

6.2 Motivation

6.3 System Assumptions

6.4 Design and Architecture

6.5 Prototype Implementation

6.6 Summary

6.1 Introduction

Scalable datastores (or simply datastores) are distributed storage systems that scale to thousands of commodity servers and manage petabytes of structured data. Today, they are routinely used by online serving, analytics and bulk processing applications, such as web indexing, social media, electronic commerce, and scientific analysis [28, 40, 32, 52, 33, 15, 101]. Datastores differ from traditional databases because they: (i) Horizontally

partition and replicate the indexed data across many servers, (ii) Provide weaker concurrency model and simpler call interface, and (iii) Allow dynamic expansion of records with new attributes. Depending on the application needs, they organize data as collections of key-value pairs, multidimensional maps or relational tables.

System scalability across multiple servers is necessitated by the enormous amount of handled data and the stringent quality-of-service requirements [28, 40, 104]. Production systems keep the high percentiles of serving latency within tens or hundreds of milliseconds [40, 104]. General-purpose datastores target good performance on both read-intensive and write-intensive applications [28, 32]. Furthermore, applications that ingest and mine event logs accelerate the shift from reads to writes [93].

The data is dynamically partitioned across the available servers to handle failures and limit the consumed resources. To a large extent, the actual capacity, functionality and complexity of a datastore is determined by the architecture and performance of the constituent servers [72, 99, 98]. For instance, resource management efficiency at each storage server translates into fewer hardware components and lower maintenance cost for power consumption, redundancy and administration time. Also, support of a missing feature (e.g., range queries) in the storage server may require substantial reorganization with overall effectiveness that is potentially suboptimal [84, 30].

A storage layer at each server manages the memory and disks to persistently maintain the stored items [98]. Across diverse batch and online applications, the stored data is typically arranged on disk as a dynamic collection of immutable, sorted files (e.g., Bigtable, HBase, Azure and Cassandra in Section 2.2.1, Hypertable [53]). Generally, a query should reach all item files to return the eligible entries (e.g., in a range). As the number of files on disk increases, it is necessary to merge them so that query time remains under control. Datastores use a variety of file merging methods but without rigorous justification. For instance, Bigtable keeps bounded the number of files on disk by periodically merging them through *compactions* [28] (also HBase, Cassandra, LazyBase in Section 2.2.1, Anvil in Section 2.2.3). In the rest of the document we interchangeably use the terms merging and compaction.

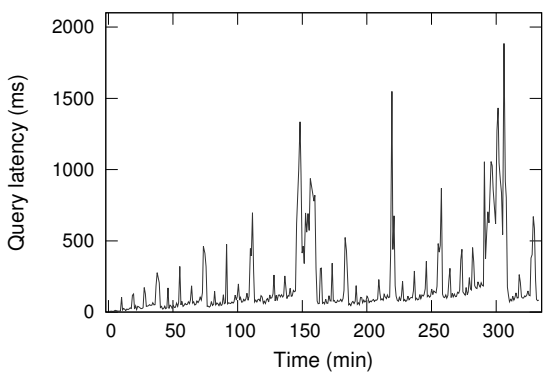
Despite the prior indexing research (e.g., in relational databases, text search), datastores suffer from several weaknesses. Periodic compactions in the background may last for hours and interfere with regular query handling leading to latency spikes [98, 71, 69,

70, 81]. To avoid this problem, production environments schedule compactions on a daily basis, thus leaving fragmented the data for several hours [98]. This leads to reduced query performance as the performance of these systems (e.g HBase) is sensitive to the number of disk files per key range [15]. Frequent updates in distinct columns of a table row further fragment the data [43]. When several files on a server store data with overlapping key ranges, query handling generally involves multiple I/Os to access all files that contain a key. Bloom filters can defray this cost, but are only applicable to single-key (but not range) queries, and have diminishing benefit at large number of files (e.g. 40) [98]. Finally, several merge-based methods require roughly half of the storage space to remain free during merging for the creation of new files [98].

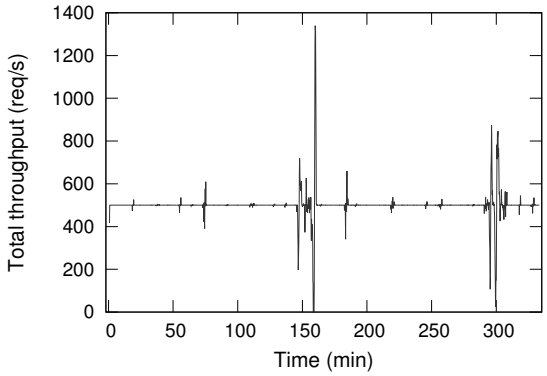
In this thesis we study the storage management of online datastores that concurrently support both range queries and dynamic updates. Over inexpensive hardware, we reduce the data serving latency through higher storage contiguity; improve the performance predictability with limited query-update interference and configurable compaction intensity; and decrease the storage space required for file maintenance through incremental compactions. Our main insight is to keep the data of the memory and disk sorted and partitioned across disjoint key ranges. In contrast to existing methods (e.g., Section 2.2.2), when incoming data fills up the available memory of the server, we only flush to disk the range that occupies the most memory space. We store the data of each range in a single file on disk, and split a range to keep bounded the size of the respective file as new data arrives at the server.

6.2 Motivation

Range queries are often used by data serving and analytics applications [32, 33, 15, 53, 25, 30, 72], while time-range queries are applied on versioned data for transactional updates [83]. Accordingly, typical benchmarks support range queries in addition to updates and point queries as workload option [33, 81]. In a distributed system, variability in the latency distribution of individual components is magnified at the service level; effective caching cannot directly address tail latency unless the entire working set of an application resides in the cache [38]. In this section, over a distributed datastore we experimentally



(a) Cassandra query latency



(b) Cassandra total operations throughput

Figure 6.1: The query latency at the Cassandra client varies according to a quasi-periodic pattern. The total throughput of queries and inserts also varies significantly.

demonstrate the range query latency to vary substantially over time with a high percentage of it to be spent in the storage layer.

We use a cluster of 9 machines with the hardware configuration described in Section 7. We apply the Apache Cassandra version 1.1.0 as datastore with the default Size-Tiered compaction and the Yahoo! YCSB version 0.1.4 as workload generator [52, 33]. An item has 100B key length and 1KB value size. A range query requests a random number of consecutive items that is drawn uniformly from the interval [1,100]. Initially we run Cassandra on a single node. On a different machine, we use YCSB with 8 threads to generate a total of 500req/s out of which 99% are inserts and 1% are range queries (see Section 7.2). We disregarded much higher loads (e.g., 1000req/s) because we found them to saturate the server. The experiment terminates when a total of 10GB is inserted into the server concurrently with the queries.

For average size of queried range at 50 items, the generated read load is 250items/s, i.e., almost half the write load of 495items/s. An I/O on our hard disk takes on average 8.5-10ms for seek and 4.16ms for rotation. Accordingly, the time to serve 5 range queries is 67.2ms, while the time to sequentially write 495 items is 21.9ms. Although the read time appears 3 times higher than that of the writes, the actual write load is practically higher as a result of the compactions involved.

In Figure 6.1 we show the query latency measured every 5s and smoothed with a window of size 12 for clarity. The query latency varies substantially over time following

Table 6.1: Storage management on the server occupies more than 80% of the average query latency measured at the client.

Latency (ms) of Range Queries on Cassandra

# Servers	Client			Server Storage Mgmt		
	Avg	**90th**	**99th**	**Avg**	**90th**	**99th**
1	204.4	420	2282	178.8	382	1906
4	157.6	313	1601	130.8	269	1066
8	132.2	235	1166	111.7	218	802

some quasi-periodic pattern which is independent of the random query size. In fact, the latency variation approximates the periodicity at which the server flushes from memory to disk the incoming data and merges the created files. In the same figure, we additionally show the measured throughput of queries and inserts to also vary considerably over time, and actually drop to zero for 90 consecutive seconds at minutes 157 and 295.

We repeat the above experiment with Cassandra over 1, 4 and 8 server machines. We linearly scale the generated request rate up to 4000req/s and the inserted dataset size up to 80GB, while we fix to 8 the number of YCSB threads at the client. We instrument the latency to handle the incoming query requests at each server. Table 6.1 shows the query latency respectively measured at the YCSB client and the storage layer of all the Cassandra servers. The difference mainly arises from time spent on network transfer, request redirection among the servers, and RPC handling. As we increase the number of servers, the query latency drops because the constant (8) number of YCSB threads results into reduced concurrency (and contention) per server in the cluster. Across different system sizes, the storage management accounts for more than 80% of the average latency and the 90th percentile, and more than 65% of the 99th percentile. Overall, compactions cause substantial latency variations, and storage management is dominant in the online performance of Cassandra-like datastores (Section 2.2.1). In the following sections we introduce a new storage structure and method to effectively control the compaction impact, and improve the datastore performance.

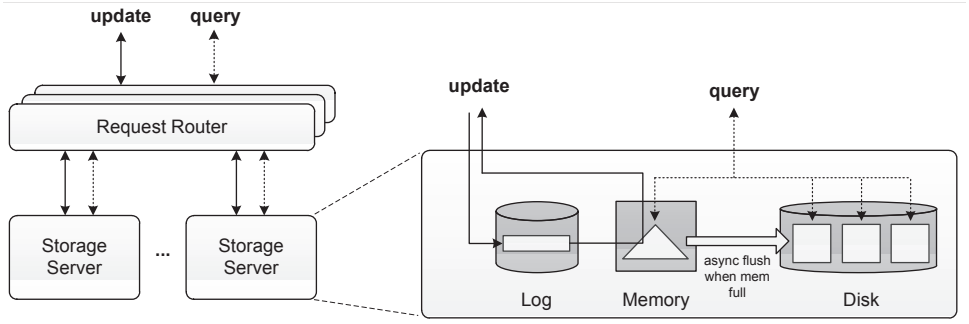


Figure 6.2: Assumed datastore architecture.

6.3 System Assumptions

We mainly target interactive applications of online data serving or analytics processing. The stored data is a collection of key-value pairs, where the key and the value are arbitrary strings of variable size from a few bytes up to several kilobytes. The system supports the operation of a *point query* as value retrieval of a single key, and a *range query* as retrieval of the values in a specified key range. Additionally, the system supports an *update* as insertion or full overwrite of a single-key value. We do not examine the problems of query handling over versioned data, or data loading in bulk.

A datastore uses a centralized (Figure 2.5b) or distributed index (Figure 2.6) to locate the server of each stored item. Data partitioning is based on interval mapping for efficiency in handling range queries (Section 2.2.2). We focus on the storage functionality of individual servers rather than the higher datastore layers. All accepted updates in a storage server are made immediately durable through write-ahead logging and then inserted in a memory search structure, before an acknowledgment is sent to the client [28, 40]. When the memory structure reaches a predefined threshold, its contents are sorted and stored in an immutable file on disk. The storage layer is implemented as a dynamic collection of sorted files, and a query must typically access multiple files. Thus, updates are commonly handled at sequential disk throughput, and queries involve synchronous random I/O. Figure 6.2 illustrates the path of an update or query through the request router and the storage servers, before returning the respective response back to the datastore client.

With data partitioning, each storage server ends up locally managing up to a few terabytes. The data is indexed by a memory-based sparse index, i.e., a sorted array with

pairs of keys and pointers to disk locations every few tens or hundreds of kilobytes. For instance, Cassandra indexes 256KB blocks, while Bigtable, HBase and Hypertable index 64KB blocks [28, 52]. With a 100B entry for every 256KB, we need 400MB of memory to sparsely index 1TB. Compressed trees can reduce the occupied memory space by an order of magnitude at the cost of extra decompression processing [66]. We provide additional details about our assumptions in Section 7.8.

6.4 Design and Architecture

In the present section we propose a novel storage layer to efficiently manage the memory and disks of datastore servers. Our design sets the following primary goals:

(i) Provide sequential disk scans of sorted data to queries and updates.

(ii) Store the data of each key range at a single disk location.

(iii) Selectively batch updates and free memory space.

(iv) Avoid storage fragmentation or reorganization and minimize reserved storage space.

Below, we describe the proposed Rangetable structure and the accompanying Rangemerge method. Then we outline the prototype software that we developed to fairly compare our approach with representative storage structures of existing systems.

6.4.1 The Rangetable Structure

The main insight of Rangetable is to keep the data on disk in key order, partitioned across large files by key range. We store the data of a range at a single file to avoid multiple seeks for a point or range query. The disk blocks of a file are closely located in typical filesystems, with allocators based on block groups or extents (e.g., ext3/4, Btrfs). If the size of a data request exceeds a few MB, the disk geometry naturally limits the head movement overhead to below 10%. For instance, if the average rotation and seek take 6.9ms in a 10KRPM SAS drive, the overhead occupies 8.6% of the total time to access 10MB [37]. We do not need enormous files to achieve sequential I/O, as long as each file has size in the tens of megabytes. We avoid frequent I/O by gathering incoming updates

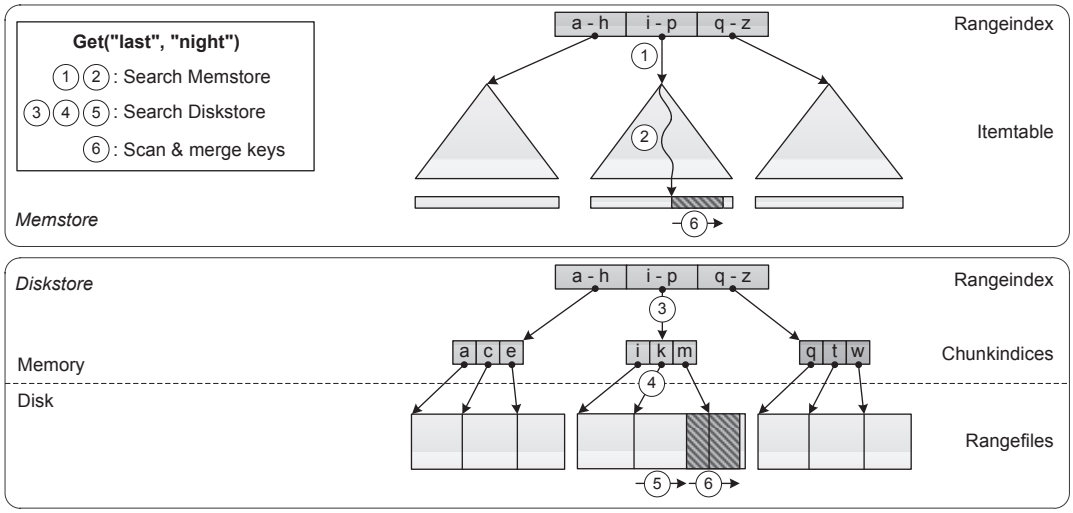


Figure 6.3: The organization of the Rangetable structure, and control flow of a handled range query. For presentation clarity we use alphabetic characters as item keys.

in memory, and inexpensively preserve range contiguity on disk by only flushing those ranges that ensure I/O efficiency.

New updates at a server are durably logged, but also temporarily accumulated in memory for subsequent batched flushing to a new file on disk (Figure 6.2). For effective I/O management, we partition the data of every server into key-sorted ranges using a memory-based table, called *rangeindex*. Each slot of the rangeindex maps a range to the respective items stored on disk and in memory (Figure 6.3). For fast key lookup and range scan we keep the data in memory sorted through a mapping structure, called *itemtable*. We use a concurrent balanced tree (specifically, a red-black tree) for each range, although a multicore-optimized structure is preferable if the stored data fully resides in memory [72].

New data are first inserted to the respective tree in memory and later flushed to disk. We avoid external fragmentation and periodic reorganization on disk by managing the space in files, called *rangefiles*, of maximum size F (e.g., 256MB). Each rangefile is organized as a contiguous sequence of *chunks* with fixed size C (e.g., 64KB). In order to easily locate the rangefile chunks, we maintain a memory-based sparse index per rangefile, called *chunkindex*, with entries the first key of each chunk and the offset within the rangefile. From the steps shown in Figure 6.3, an incoming range query (1) finds the respective tree in memstore using the rangeindex and (2) searches this tree. Then, the

query searches (3) the rangeindex, (4) the chunkindex and (5) the rangefile of the diskstore. Finally, (6) the requested items from both the itemtable and rangefile are merged into a single range by the server and returned.

6.4.2 The Rangemerge Method

In order to serve point and range queries with roughly one disk I/O, the Rangemerge method merges items from memory and disk in range granularity. When we merge items, we target to free as much memory space as possible at minimal flushing cost. The choice of the flushed range affects the system efficiency in several ways: (i) Every time we flush a range, we incur the cost of one rangefile read and write. The more new items we flush, the higher I/O efficiency we achieve. (ii) A flushed range releases memory space that is vital for accepting new updates. The more space we release, the longer it will take to repay the merging cost. (iii) If a range frequently appears in queries or updates, then we should skip flushing it to avoid repetitive I/O.

Memory flushing and file merging are generally regarded as two distinct operations. When memory fills up with new items, the server has to free memory space quickly to continue accepting new updates. Existing systems sequentially transfer to disk the entire memory occupied by new items. Thus, they defer merging to avoid blocking incoming updates for extended time period. This approach has the negative effect of increasing the files and incurring additional I/O traffic to merge the new file with existing ones [93]. To avoid this extra cost, Rangemerge treats memory flushing and file merging as a single operation rather than two. It also limits the duration of update blocking because a range has configurable maximum size, typically a small fraction of the occupied memory at the server (Section 7.8).

We greedily victimize the range with the largest amount of occupied memory space. The intuition is to maximize the amount of released memory space along with the I/O efficiency of the memory flush. For simplicity, we take no account of the current rangefile size, although this parameter affects the merging cost, and the probability of having future I/O requests to a particular range. Despite its simplicity, this victimization rule has proved robust across our extensive experimentation.

The pseudocode of Rangemerge appears in Algorithm 6.3. The server receives items

Algorithm 6.3 Pseudocode of RANGEMERGE

Input: Rangetable with memory size >= M

Output: Rangetable with memory size < M

1: *// Victimize a range*
2: R := range whose tree occupy max total memory
3: *// Flush memory items of R to its rangefile*
4: Merge rangefile f_R of R with its tree m_R into empty buffer b
5: **if** (sizeof(b) > F) **then** *// If block will overflow*
6: $k := \lceil \text{sizeof}(b)/F \rceil$
7: **else**
8: $k := 1$
9: **end if**
10: Allocate k new rangefiles $f_R^1, \ldots, f_R^k$ on disk
11: Split b into k subranges $R^1, \ldots, R^k$ of equal disk size
12: Transfer subranges to respective $f_R^1, \ldots, f_R^k$
13: Build chunkindexes for $f_R^1, \ldots, f_R^k$
14: Update rangeindex with entries for $R^1, \ldots, R^k$
15: *// Clean up memory and disk*
16: Free tree m_R
17: Delete rangefile f_R and its chunkindex

in the key interval assigned by the datastore index. We insert new items in their trees until the occupied memory space reaches the *memory limit* M. At this point, we pick as *victim* R the range of maximum memory space (line 2), read its rangefile f_R from disk, merge it with the respective tree m_R in memory, and move the merged range back to disk (lines 4-13). The addition of new items may lead the size of range R to exceed the rangefile capacity F (line 5). In this case, we equally split R into k (usually, $k = 2$) subranges and move the data to k new rangefiles on disk (line 11). Finally, we free the itemtable space occupied by R, and delete the old rangefile from the disk (lines 16-17). Practically, flushing a single range is sufficient to reduce the occupied memory below the memory limit.

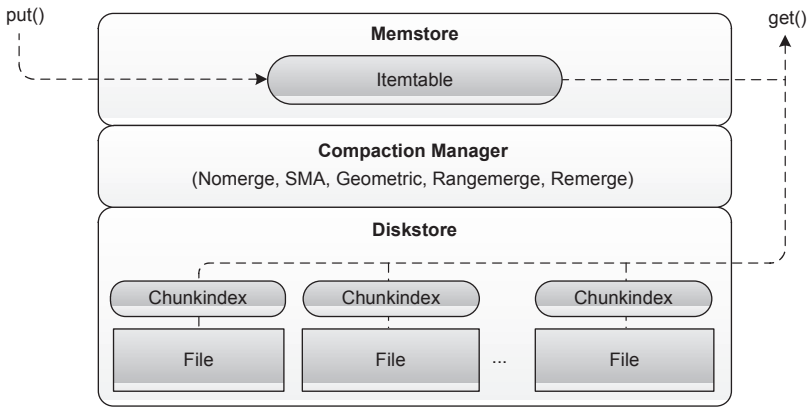


Figure 6.4: Prototype framework with several compaction methods as plugins.

6.5 Prototype Implementation

We developed a general storage framework to persistently manage key-value items over local disks. The interface supports the `put(k,v)` call to insert the pair `(k,v)`, the `get(k)` call to retrieve the value of key `k`, the `get(k,n)` call to retrieve `n` consecutive records from key $\geq$ `k`, and the `get(`k_1`,`k_2`)` call to retrieve the records with keys in the range $[k_1, k_2]$. Our prototype adopts a multithreaded approach to support the concurrent execution of queries and updates, and it is designed to easily accept different compaction methods as pluggable modules. The implementation consists of three main components, namely the *Memstore*, the *Diskstore*, and the *Compaction manager* (Figure 6.4).

The Memstore uses a thread-safe red-black tree in memory to maintain incoming items in sorted order (or multiple trees, in case of Rangemerge), and the Diskstore accesses each sorted file on disk through a sparse index maintained in memory. The Compaction manager implements the file merging sequences of the following methods: Nomerge, SMA, Geometric, Rangemerge and Remerge. We implemented the methods using C++ with the standard template library for basic data structures and 3900 uncommented lines of new code.

To validate the accuracy of our experimentation, we compared the compaction activity of our storage framework with that of Cassandra. From review of the published literature and the source code, we found that Cassandra implements a variation of the SMA (k=4) algorithm [52]. Accordingly, the stored data is organized into levels of up to k=4 files; every time the threshold of k=4 files is reached at one level, the files of this level are merged

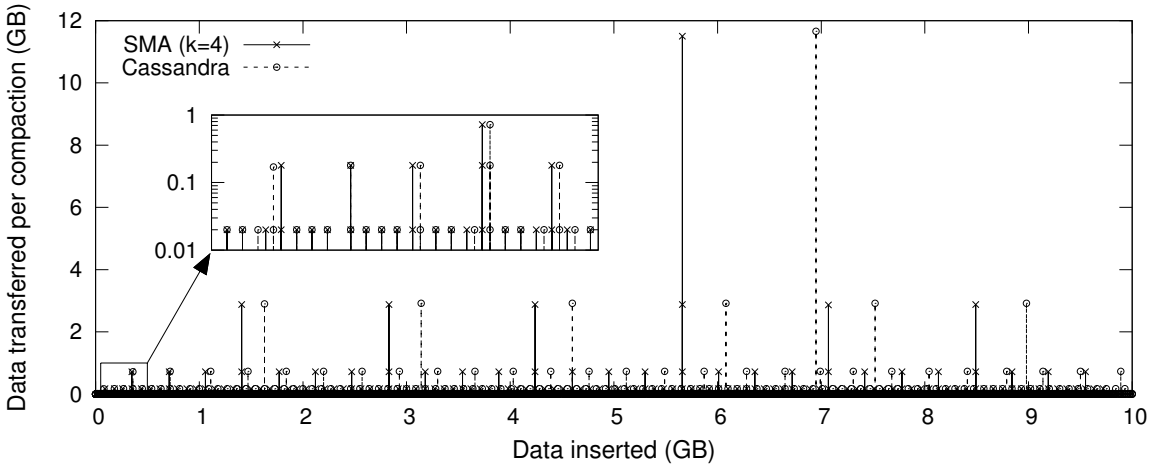


Figure 6.5: We observe similar compaction activity between Cassandra and our prototype implementation of SMA (k=4). The height (y-axis value) of each mark denotes the transfer size of the respective compaction.

into a single file of the next level (Section 2.2.2). In our framework we set M=25MB because we found that Cassandra by default flushes to disk 25MB of data every time memory gets full. For comparison fairness we disable data compression and insertion throttling in Cassandra. We create the Cassandra workload using YCSB with 2 clients, which respectively generate puts at 500req/s and gets at 20req/s. The stored items are key-value pairs with 100B key and 1KB value, while the size of the get range is drawn uniformly from the interval [1,20]. The experiment terminates when 10GB of data is inserted. We generate a similar workload in our framework with two threads.

In Figure 6.5 we show the amount of transferred data as we insert new items into the Cassandra and our prototype system respectively. The height of each mark refers to the total amount of transferred data during a compaction. Across the two systems we notice quasi-periodic data transfers of exactly the same size. In the case that a merge at one level cascades into further merges at the higher levels, in our prototype we complete all the required data transfers before we accept additional puts. Consequently, it is possible to have multiple marks at the same x position. Instead Cassandra allows a limited number of puts to be completed between the cascading merges, which often introduces a lag between the corresponding marks. Overall the two systems transfer equal amount of data using the same compaction pattern during the dataset insertion.

6.6 Summary

To achieve fast ingestion of new data, scalable datastores usually follow a write-optimized approach. Incoming updates are simply logged to disk and accumulated in memory, before the system returns control to the client. When later on the available memory is exhausted, a flush operation will sort all memory updates and transfer them to a new file on disk. To improve query performance and reclaim space from obsolete entries, the system periodically selects and merges multiple disk files into a single file. These flush and merge operations are collectively called compactions and are usually executed in the background.

This general approach of amortizing the insertion cost over the periodic compactions is adopted by most production datastores as it achieves high ingestion throughput. However, it increases the latency of range queries as it fragments the data of each key in several disk files, causing multiple random I/Os per query. Furthermore, even though background compactions are only periodically executed, they are very resource-intensive and have a significant impact on the serving of concurrent queries. Finally, this approach requires roughly half of the storage space to be reserved for the creation of new files during merges.

To address these issues, we present the Rangetable storage layer and the Rangemerge method to efficiently manage the memory and disks of datastore servers. We also describe our prototype storage framework and provide details about the implementation. Rangemerge improves query latency by keeping the entries contiguously stored on disk and minimizes the interference between compactions and queries by only partially flushing entries from memory to disk using lighter compactions. The indexing throughput is maintained high by scheduling the merges of memory and disk entries based on their I/O efficiency. Furthermore, Rangemerge implicitly avoids the excessive storage reservation of other methods.

Chapter 7

Performance Evaluation of Rangemerge

7.1 Experimentation Environment

7.2 Query Latency and Disk Files

7.3 Insertion Time

7.4 Sensitivity Study

7.5 Memory Size

7.6 Key Distribution

7.7 Solid-State Drives

7.8 Discussion

7.9 Summary

In the present section, we experimentally evaluate the query latency and insertion time across several compaction methods. We show that Rangemerge achieves minimal query latency of low sensitivity to the I/O traffic from concurrent compactions, and approximates or even beats the insertion time of write-optimized methods under various conditions. We also examine the performance sensitivity to various workload parameters

and storage devices. Although not explicitly shown, Rangemerge also trivially avoids the 100% overhead in storage space of other methods [98].

7.1 Experimentation Environment

We did our experiments over servers running Debian Linux 2.6.35.13. Each machine is equipped with one quad-core 2.33GHz processor (64-bit x86), one activated gigabit ethernet port, and two 7200RPM SATA2 disks. Unless otherwise specified, we configure the server RAM equal to 3GB. Each disk has 500GB capacity, 16MB buffer size, 8.5-10ms average seek time, and 72MB/s sustained transfer rate. Similar hardware configuration has been used in a recent related study [86]. We store the data on one disk over the Linux ext3 filesystem. In Rangemerge, we use rangefiles of size F=256MB. We also examine Remerge, Nomerge, Geometric (r=2, r=3, or p=2) and SMA (k=2 or k=4, with unlimited ℓ). In all methods, we use chunks of size C=64KB. From Section 2.2.2, variations of these methods are used by Bigtable, HBase (Geometric, r=3), Anvil and bLSM (Geometric, r=2), GTSSL (SMA, k=4), and Cassandra (SMA, k=4).

We use YCSB to generate key-value pairs of 100 bytes key and 1KB value. On a single server, we insert a dataset of 9.6M items with total size 10GB. Similar dataset sizes per server are typical in related research (e.g., 1M [32], 9M [81], 10.5GB [86], 16GB [98], 20GB [33]). The 10GB dataset size fills up the server buffer several times (e.g., 20 for 512MB buffer space) and creates interesting compaction activity across the examined algorithms. With larger datasets, we experimentally found the server behavior to remain qualitatively similar, while enormous datasets are typically partitioned across multiple servers. For experimentation flexibility and due to lack of public traces [4], we use synthetic datasets with keys that follow the uniform distribution (default), Zipfian distribution, or are partially sorted [33]. We take average measurements every 5s, and smooth the output with window size 12 (1-min sliding window) for readability. Our default range query reads 10 consecutive items.

The memory limit M refers to the memory space used to buffer incoming updates. Large system installations use dynamic assignment to achieve load balancing by having a number of service partitions (*micro-partitions*) that is much larger than the number

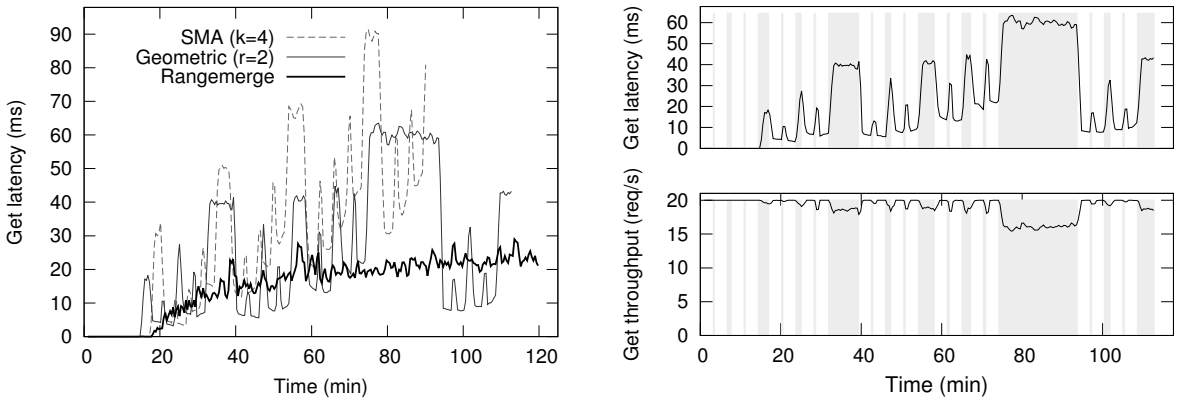
of available machines [38]. For instance, the Bigtable system stores data in tablets with each machine managing 20 to 1,000 tablets. Consequently, the default buffer space per tablet lies in the range 64-256MB [41]. Other related research configures the memory buffer with size up to several GB [98, 33]. As a compromise between these choices, we set the default memory limit equal to M=512MB; thus we keep realistic (1/20) the ratio of memory over the 10GB dataset size and ensure the occurrence of several compactions throughout an experiment. In Section 7.3 we examine memory limit and dataset size up to 4GB and 80GB respectively. We further study the performance sensitivity to memory limit M in Figure 7.9.

7.2 Query Latency and Disk Files

First we measure the query latency of a mixed workload with concurrent puts and gets. An I/O over our disk takes on average 13.4ms allowing maximum rate about 74req/s (can be higher for strictly read workloads). We configure the get load at 20req/s so that part of the disk bandwidth can be used by concurrent compactions. We also set the put rate at 2500req/s, which is about half of the maximum possible with 20get/s (shown in Figure 7.6c). The above combined settings occupy roughly two thirds of the total disk bandwidth and correspond to a write-dominated workload (get/put ratio about 1/100 in operations and 1/25 in items) [86]. We examine other combinations of put and get loads in Section 7.4.

We assume that when memory fills up, the put thread is blocked until we free up memory space. Although write pauses can be controlled through early initiation of memory flushing [93], their actual effect to insertion performance additionally depends on the flushing granularity and duration (explored in Section 7.8). In order to determine the concurrency level of query handling in the server, we varied the number of get threads between 1 and 20; then we accordingly adjusted the request rate per thread to generate total get load 20req/s. The measured get latency increased with the number of threads, but the relative performance difference between the methods remained the same. For clarity, we only illustrate measurements for one put and one get thread.

In Figure 7.1a we examine three representative methods: SMA (k=4), Geometric



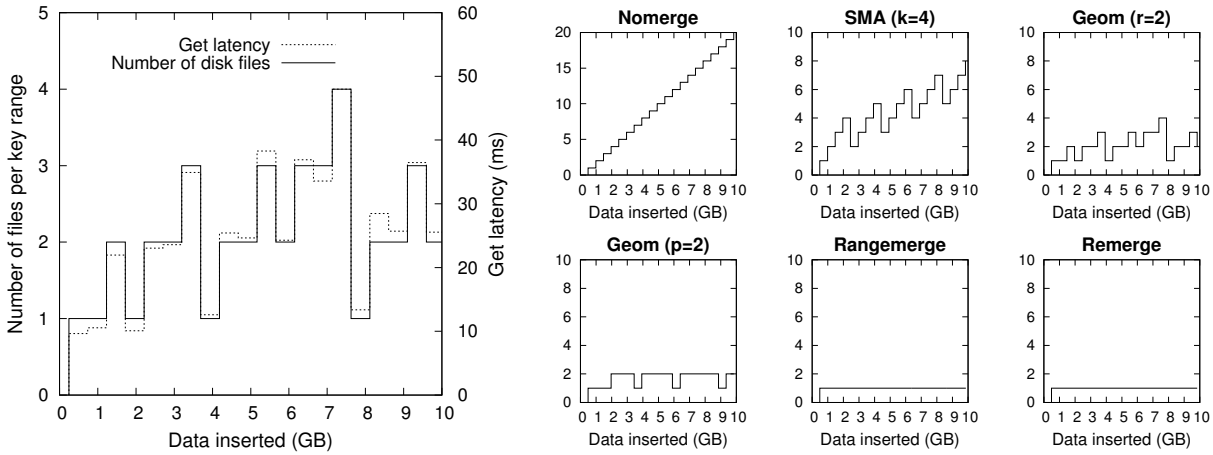
(a) Get latency in three methods

(b) Compactions in Geom (r=2)

Figure 7.1: During concurrent inserts and queries, (a) the get latency of Geometric (r=2) and SMA (k=4) has substantially higher variability and average value than Rangemerge, and (b) the get throughput of Geometric (r=2) drops as low as 15.5req/s during compactions (grey background).

(r=2), and Rangemerge. The experiment runs separately for each method until loading 10GB. The get latency of Rangemerge (avg: 15.6ms, std: 8.2ms, max: 30.5ms) has lower average value by 51-83% and standard deviation by 2.5-3 times than Geometric (avg: 23.5ms, std: 20.5ms, max: 64.5ms) and SMA (avg: 28.6ms, std: 24.3ms, max: 93.3ms). Also Remerge (not shown) is less responsive and predictable (avg: 21.1ms, std: 10.6ms, max: 35.4ms) than Rangemerge. However, SMA reduces the experiment duration to 90min from 119min required by Rangemerge and 112min by Geometric (see also Figure 7.9). In Figure 7.1b we illustrate the get performance of Geometric, with concurrent compactions as vertical grey lanes. Compactions increase latency by several factors and reduce throughput by 22.5%, from 20req/s to 15.5req/s. The throughput of SMA (not shown) also drops to 10.4req/s, unlike the Rangemerge throughput that remains above 17.4req/s.

In Figure 7.2a we depict the number of files (left y axis) and the average get latency (right y axis) for Geometric. After every compaction, we measure the get latency as average over twenty random requests. From every file, the get operation reads the items of the requested key range (Figure 6.3). Assuming no concurrent compactions, the measured latency varies between 11.9ms and 49.0ms, as the number of files per key varies between 1



(a) Files and get latency in Geom (r=2)

(b) Number of files per key range

Figure 7.2: (a) At the insertion of 10GB with M=512MB using Geometric partitioning (r=2), get latency (at load 10req/s) is closely correlated to the number of files created. (b) We show the number of files maintained per key range for six methods.

and 4. The evident correlation between get latency and the number of files in Geometric explains the variation of get performance in between compactions in Figure 7.1b.

We further explore this issue in Figure 7.2b, where we show the number of maintained files as function of the dataset size. Nomerge increases the number of sorted files up to 20 (only limited by the dataset size), and SMA (k=4) increases the number of created files up to 8. Geometric with r=2 and p=2 varies the number of files up to 4 and 2, respectively. Instead, Remerge always maintains a single file for the entire dataset, while Rangemerge strictly stores on a single file the items of a rangefile range; both methods lead to roughly one random I/O per get operation. Overall Rangemerge leads to more responsive and predictable get operations with concurrent puts.

We also examine the generated I/O activity of compactions. In Figure 7.3 we illustrate the data amount written to and read from disk for 10GB dataset and M=512MB. The plots of the figure are ordered according to the decreasing size of the maximum transferred amount. Remerge merges data from memory to an unbounded disk file with 10GB final size. At the last compaction, the amount of transferred data becomes 20.5GB. Geometric reduces the transferred amount down to 16.5GB for r=2. In SMA, k=4 limits the transferred amount to 4.5GB; k=2 (not shown) leads to 14.5GB maximum compaction transfer

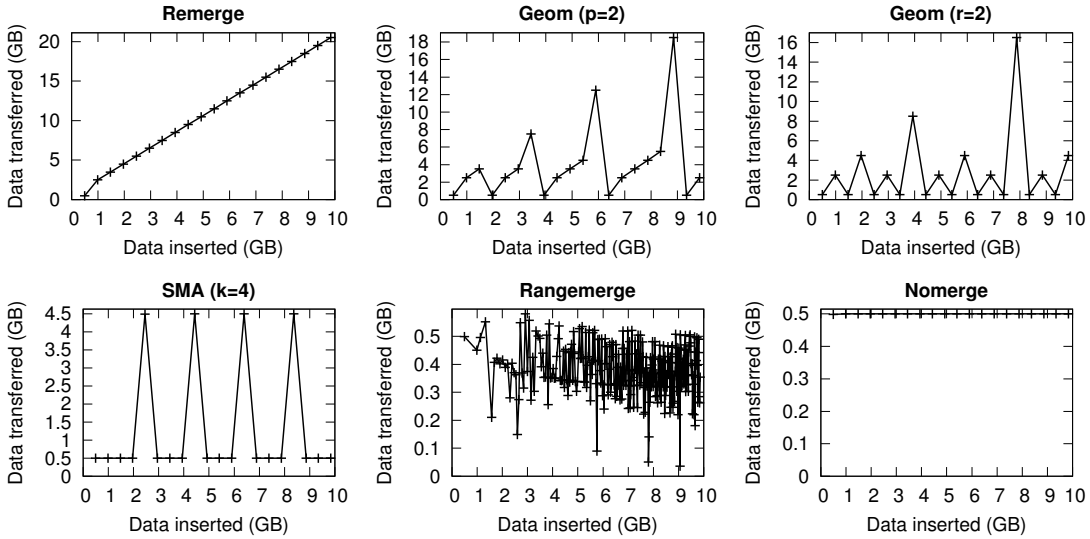


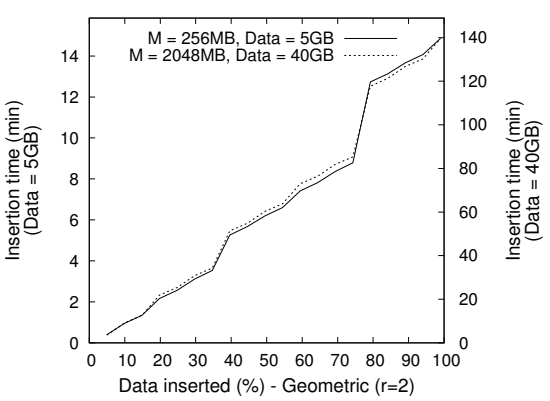
Figure 7.3: I/O intensity of compactions. The disk traffic of compactions in Rangemerge is comparable to that of Nomerge with M=512MB.

with 6 files. It is interesting that Rangemerge reduces to 594MB the maximum transferred amount per compaction bringing it very close to 512MB periodically transferred by Nomerge. Thus Rangemerge makes compactions less I/O aggressive with respect to concurrent gets (Figure 7.1a).

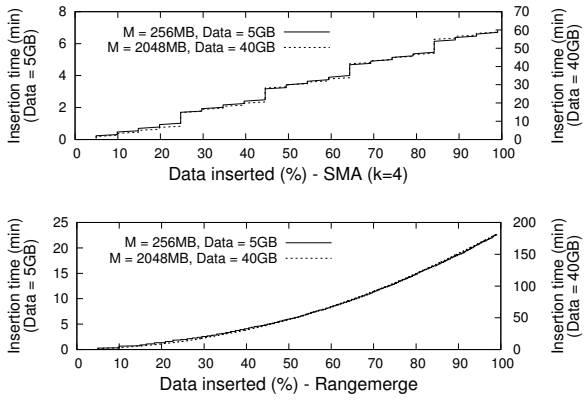
7.3 Insertion Time

Next we study the cumulative latency to insert data items one-by-one into the storage server. Insertion includes some processing to sort the data in memory, but mainly involves I/O to flush data and apply compactions over the disk files. In order to ensure the generality of our results, we measured the total insertion time at different scales of dataset size and memory limit. In Figure 7.4a and Figure 7.4b the cumulative insertion time of Geometric, SMA and Rangemerge forms a similar curve as long as the ratio of dataset size over memory limit is constant (e.g., 5GB/256MB=40GB/2GB=20). We confirmed this behavior across several parameter scales that we examined.

In Figure 7.5a (with log y axis) we examine the time required to insert a dataset using different compaction methods. We already displayed the number of maintained files for different methods in Figure 7.2b. Nomerge takes 9.3min to create 20 files on disk, and



(a) Geom (r=2)

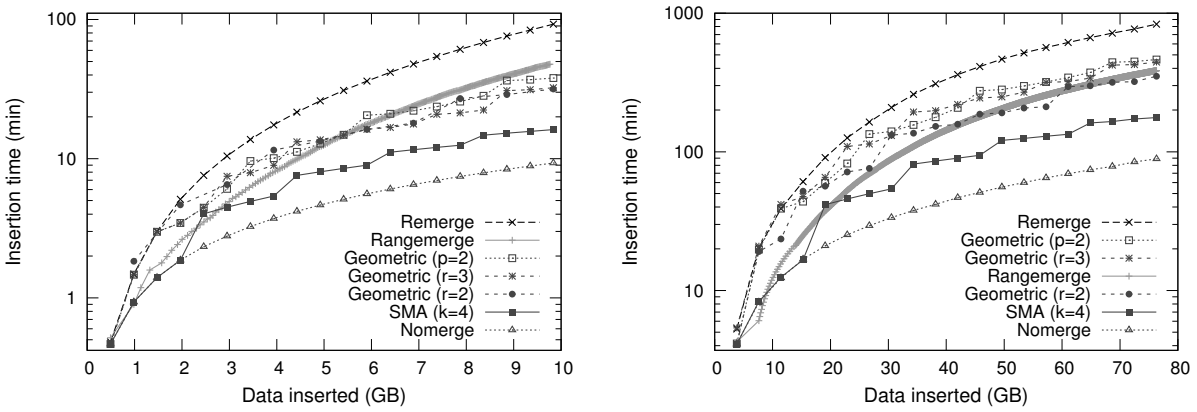


(b) SMA (k=4), Rangemerge

Figure 7.4: Scaling configuration parameters. The insertion progress is similar between the configuration of M=256MB with 5GB dataset (left y-axis) and M=2GB with 40GB (right y-axis) for Geometric (r=2), SMA (k=4) and Rangemerge.

SMA (k=4) spends 16.2min for 8 files. Geometric takes 31.7min with r=2, 32.4min with r=3, and 38.0min with p=2 ($\leq$ 2 files). Remerge requires 92.5min to maintain a single file on disk, and Rangemerge takes 47.8min. As expected, the smaller the number of disk files maintained, the longer it takes to insert the dataset. One exception is Rangemerge that requires about half the insertion time of Remerge to effectively store each key at a single disk location. Geometric reduces the insertion time of Rangemerge by 20.5%–33.6%, but requires 2-4 random I/Os on disk to handle a query (Figure 7.2b) and has greater variability in query latency due to its I/O-intensive background compactions (Figure 7.1a). In addition, under modest concurrent query load the insertion time of Rangemerge is lower than Geometric (p=2) and similar to Geometric (r=3) (Section 7.4).

In Figure 7.5b we repeat the above experiment using a dataset of 80GB with M=4GB over a server with 6GB RAM. Nomerge takes 1.5hr and SMA (k=4) 2.9hr. Geometric takes 5.8hr with r=2, 7.4hr with r=3, and 7.7hr with p=2. Remerge requires 13.9hr and Rangemerge takes 6.4hr. Interestingly, the insertion time of Rangemerge is lower than that of Geometric with p=2 and r=3, even though it stores each disjoint range of keys on a single file on disk. We attribute this behavior to the more efficient use of the available memory by Rangemerge, further explored in Section 7.5.



(a) Insertion time (10GB dataset, M=512MB)

(b) Insertion time (80GB dataset, M=4GB)

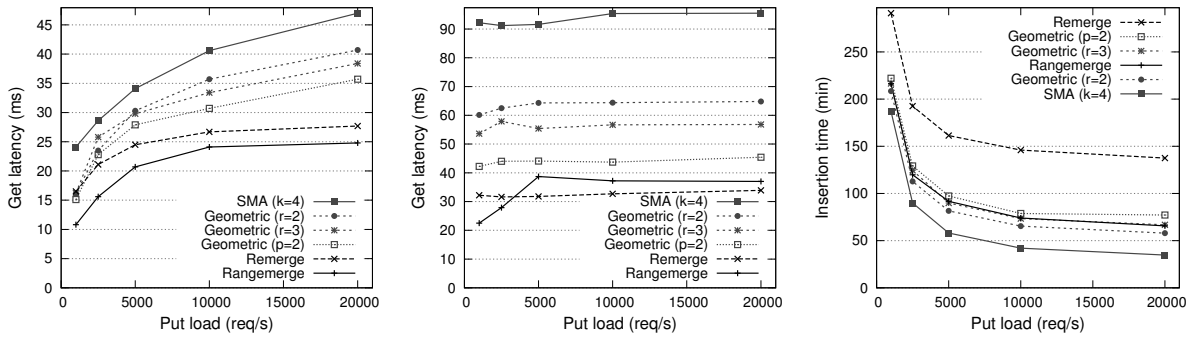
Figure 7.5: (a) The insertion time (log y axis) of Rangemerge is about half the insertion time of Remerge and closely tracks that of Geometric (p=2). (b) With M=4GB and 80GB dataset size Rangemerge has lower insertion time than Geometric (p=2) and (r=3) while storing each key at a single disk location.

7.4 Sensitivity Study

We did an extensive sensitivity study with respect to the concurrent load. Specifically, we examined how the get latency and the total insertion time is affected when we vary the put and get load of the system.

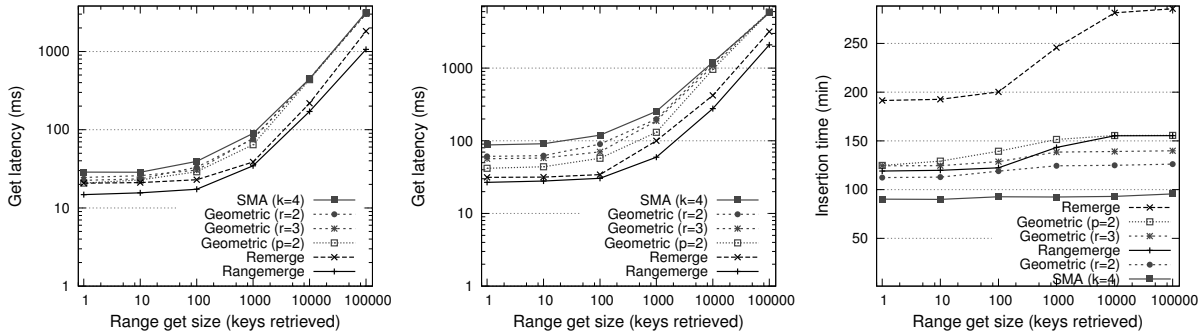
First we evaluate the impact of the put load to the query and insertion time. As we vary the put load between 1000-20000req/s, the average latency of concurrent gets is lowest under Rangemerge (Figure 7.6a). According to the needs of Service Level Agreements [40, 101, 104], we also consider the 99th percentile of get latency in Figure 7.6b. Rangemerge and Remerge are the fastest two methods. Moreover, under concurrent puts and gets, the insertion time of Rangemerge closely tracks that of Geometric (r=3) and lies below that of Remerge and Geometric (p=2) (Figure 7.6c). We omit Nomerge because it leads to excessively long get latency.

We also examine the sensitivity to the get size assuming gets of rate 20req/s concurrently served with puts of rate 2500req/s. In Figures 7.7a and 7.7b we use logarithmic y axis to depict the latency of get requests. Across different get sizes and especially at the larger ones (e.g., 10MB or 100MB), Rangemerge is distinctly faster (up to twice or more) than the other methods both in terms of average get latency and the respective 99th per-



(a) Average get latency (b) 99th percentile of latency (c) Cumulative insertion time

Figure 7.6: Performance sensitivity to put load assuming concurrent get requests at rate 20req/s and scan size 10.



(a) Average get latency (b) 99th percentile of latency (c) Cumulative insertion time

Figure 7.7: Sensitivity to range get size assuming concurrent load of 2500req/s put rate and 20req/s get rate.

centile. From Figure 7.7c it also follows that the concurrent get load has an impact on the insertion time. Remerge takes as high as 285min with larger get sizes, unlike Rangemerge that remains between two instances of the Geometric method (r=2 and p=2).

In Figure 7.8, as the load of concurrent gets varies up to 40req/s, the insertion time of Rangemerge lies at the same level as Geometric and well below Remerge. Under mixed workloads with both puts and gets, from Figures 7.6, 7.7 and 7.8 we conclude that Rangemerge achieves the get latency of Remerge and the insertion time of Geometric.

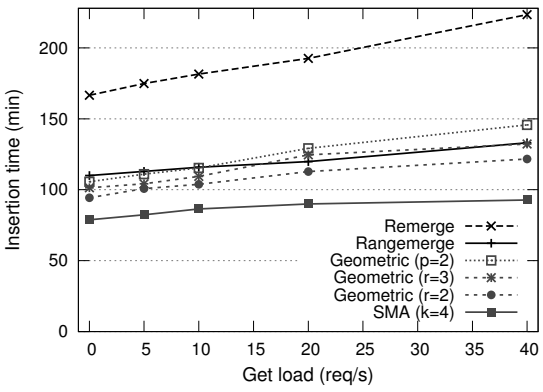


Figure 7.8: Sensitivity of insertion time to get rate of scan size 10 with concurrent put rate set at 2500req/s.

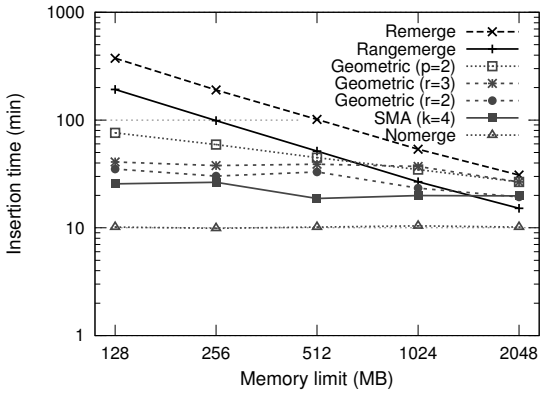


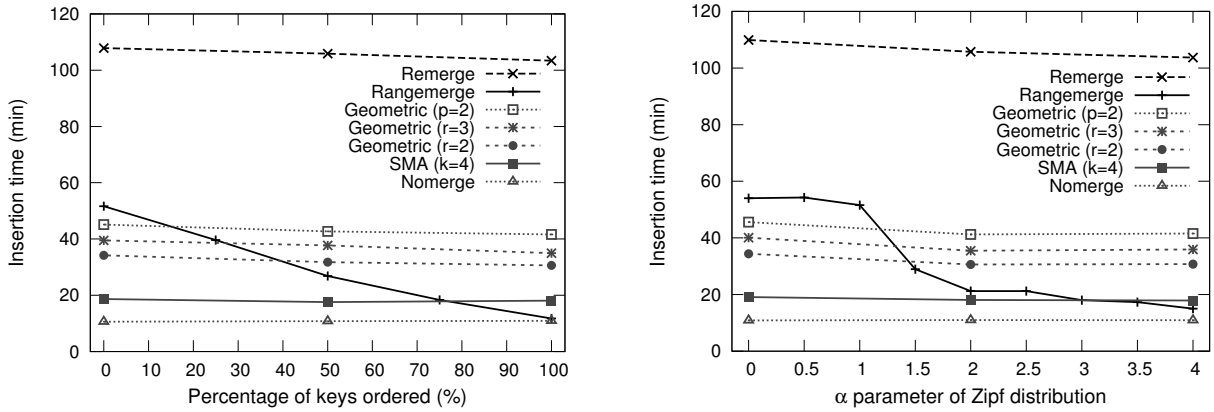
Figure 7.9: Impact of M to insertion time. With M=2GB, Rangemerge approaches Nomerge and stays by at least 21% below the other methods.

7.5 Memory Size

We also evaluate how insertion time depends on the memory limit M (Figure 7.9 with logarithmic y axis). As we increase M from our default value 512MB to 2GB, both Remerge and Rangemerge proportionally reduce the disk I/O time. This behavior is consistent with the respective I/O complexities in Table 2.1 and Section 9.2. At M=2GB, Rangemerge lowers insertion time to 15.2min, which approximates the 10.2min required by Nomerge. The remaining methods require more time, e.g., 19.3min for Geometric (r=2), 19.8min for SMA (k=4) and 30.9min for Remerge. From additional experiments (not shown) we found that a higher M does not substantially reduce the get latency of the remaining methods except for the trivial case that the entire dataset fits in memory. We conclude that the insertion time of Rangemerge approximates that of Nomerge at higher ratio of memory over dataset size.

7.6 Key Distribution

There are many scenarios where the distribution of keys inserted into the datastore is skewed or the keys are already sorted. For example, datastores are commonly used to store timeseries (e.g., system events or user transactions). In these cases the key is usually



(a) Ordered keys

(b) Zipfian keys

Figure 7.10: Sensitivity of insertion time to key distribution, as we generate put requests back-to-back with zero get load.

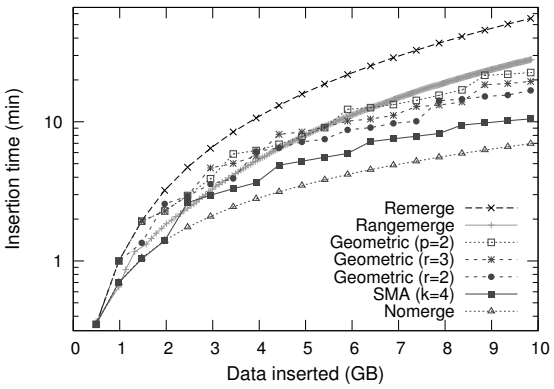
a timestamp so we expect the keys to be ordered —or mostly ordered, if items are collected from multiple sources.

In Figure 7.10a we investigate how insertion time is affected by the percentage of keys inserted in sorted order. Rangemerge approaches Nomerge as the percentage of sorted keys increases from 0% (uniform distribution) to 100% (fully sorted). This behavior is anticipated because the sorted order transforms merges to sequential writes with fewer reads.

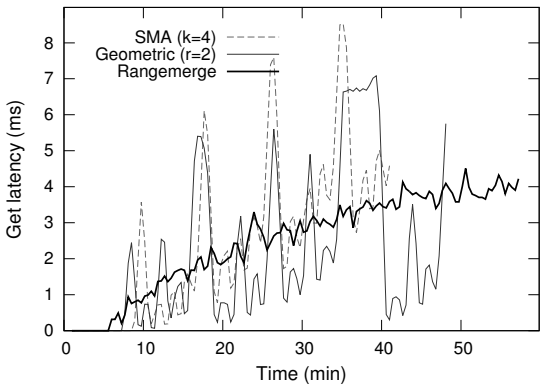
In Figure 7.10b we draw the inserted keys from a Zipfian distribution and study the impact of parameter α to the insertion time. The higher we set the parameter α, the more items appear at the head (popular part) of the distribution. Rangemerge naturally exploits the higher item popularity to again approximate Nomerge.

7.7 Solid-State Drives

Given the enormous technological improvement of solid-state drives (SSD) over the last decade, it is reasonable to consider their behavior as part of the storage hierarchy in a datastore server. Flash SSDs reduce I/O latency at the cost of hardware equipment and system complexity; the limited lifespan and the relatively poor random-write performance



(a) Cumulative insertion time



(b) Get latency

Figure 7.11: (a) Over an SSD, the insertion time of Rangemerge lies halfway between that of Nomerge and Remerge. (b) Rangemerge reduces the variability of get latency in comparison to SMA ($k = 4$) and Geometric ($r = 2$).

have been recognized as problem for the wider deployment of SSDs [77]. In our following experiments we assume that an SSD fully replaces the hard disk drive (HDD) as medium of persistent storage for the written key-value pairs. Our SSD is a SATA2 60GB solid-state drive of max read throughput 285MB/s and max write throughput 275MB/s. In Figure 7.11a we show the cumulative insertion time over an SSD for a 10GB dataset and memory limit 512MB. The compaction methods applied over the SSD reduce by 28%-60% the insertion time measured over the HDD (Figure 7.9). However, the relative performance between the methods remains similar across the two devices. In particular, Rangemerge reduces the insertion time of Remerge by 49% with SSD, and by 53% with HDD.

Next we examine the query latency over the SSD device. From the previous paragraph, the write data throughput of our SSD device is about twice as high as that of the HDD. Therefore we increase the put rate of the background traffic to 5000req/s for the SSD from 2500req/s previously used for the HDD (Section 7.2). In order to estimate the query transaction capacity of the two devices, we use a synthetic benchmark with each request involving a random seek followed by a read of 512B block. Thus we found the read performance of the HDD equal to 76req/s, and that of the SSD 4323req/s. First we tried get load of the SSD at rate 1000req/s in analogy to 20req/s that we used for the HDD

(26% of 76req/s). However the SSD device is saturated (dramatic drop of throughput) with concurrent workload of 5000req/s puts and 1000req/s gets. Thus we reduced the get load to 100req/s, so that we stay below the performance capacity of the SSD (and keep close to 1/100 the operation get/put ratio as with the HDD).

In Figure 7.11b we compare the get latencies of SMA ($k = 4$), Geometric ($r = 2$) and Rangemerge. We terminate the experiment after we insert 10GB into the system concurrently with the get load. In comparison to the get latency over the HDD (Figure 7.1a), the measured latencies of the SSD are about an order of magnitude lower. However the curves of the three methods look similar across the two devices. In fact the maximum get latency of Rangemerge reaches 4.5ms, while that of Geometric ($r = 2$) gets as high as 7.1ms and that of SMA ($k = 4$) 8.6ms. We conclude that the relative insertion and query performance of the compaction methods remains similar across the two different types of storage devices that we experimented with.

7.8 Discussion

In this section, we discuss about practical issues that we considered in our design and potential limitations resulting from our assumptions.

7.8.1 Compaction I/O Intensity

Motivated from the highly variable query latency in several existing datastores, we propose the Rangemerge method to reduce the I/O intensity of file merging in several ways: (i) We only flush a single range from memory rather the entire buffer space, and keep the amount of I/O during a flush independent of the memory limit. (ii) We combine flushing and compaction into a single operation to avoid extra disk reads during merging. (iii) We keep the size of disk files bounded in order to avoid I/O spikes during file creation. The configurable size of the rangefile provides direct control of the I/O involved in a range flush.

In Table 7.1 we consider loading 10GB to a datastore at unthrottled insertion rate. From the transferred data and the compaction time we estimate every compaction to require 32.7-36.2MB/s, which is about half of the sequential disk bandwidth. If we reduce

Table 7.1: Amount of flushed and totally transferred data per compaction, delay per compaction, and total insertion time for different rangefile sizes of Rangemerge.

Delays and Transferred Data over a Hard Disk

Rangefile (MB)	Flushed (MB)	Transferred (MB)	Compaction Time (s)	Insertion Total (min)
32	4.9	49.1	1.5	54.2
64	9.6	97.8	2.7	51.6
128	19.1	196.0	5.5	51.6
256	37.1	386.8	11.0	53.8

the rangefile size from F=256MB to F=32MB, the average duration of a compaction drops from 11.0s to 1.5s, but the respective total insertion time varies in the range 51.6min to 54.2min. It is not surprising that M=32MB raises insertion time to 54.2min, because a smaller rangefile causes more frequent and less efficient data flushes. In practice we can configure the rangefile size according to the insertion and query requirements of the application.

Previous research has already explored ways to continue accepting insertions during memory flushing. When the memory limit M is reached, it is possible to allocate additional memory space of size M to store new inserts, and let previously buffered data be flushed to disk in the background [28]. Alternatively, a low and high watermark can be set for the used fraction of memory space. The system slows down application inserts when the high watermark is exceeded, and it stops merges when the occupied memory drops below the low watermark [93]. Depending on the rate of incoming inserts, such approaches can defer the pause of inserts. However they do not eliminate the interference of compaction I/O with query requests that we focus on in our present study. Essentially, the above approaches can be applied orthogonally to the Rangemerge compaction mechanism that allows queries to gracefully coexist with inserts.

7.8.2 Queries

Range queries are supported by most datastores that use range partitioning (e.g., Bigtable, Cassandra) and are used by data serving and analytics applications (Section 6.2). We do not consider Bloom filters because they are not applicable to range queries, and their

effectiveness in point queries has been extensively explored previously; in fact, support for range queries can orthogonally coexist with Bloom filters [28].

We recognize that query performance is hard to optimize for the following reasons: (i) Service-level objectives are usually specified in terms of upper-percentile latency [40, 104]. (ii) Query performance is correlated with the number of files at each server [15, 81, 98]. (iii) The amortization of disk writes may lead to intense device usage that causes intermittent delay (or disruption) of normal operation [71, 69, 70, 101]. (iv) The diversity of supported applications requires acceptable operation across different distributions of the input data keys [33].

7.8.3 Updates

Incoming updates are inserted to the itemtable, and queries are directed to both the itemtable and the rangefiles (Figure 6.4). Although the itemtable supports concurrent updates at high rate, the rangeindex along with the rangefiles and their chunkindexes remain immutable between range merges. Every few seconds that Rangemerge splits a range and resizes the rangeindex, we protect the rangeindex with a coarse-grain lock. We find this approach acceptable because the rangeindex has relatively small size (in the order of thousands entries) and only takes limited time to insert a new range.

The enormous amount of I/O in write-intensive workloads has led to data structures that involve infrequent but demanding sequential transfers [79, 63]. Excessive consumption of disk bandwidth in maintenance tasks can limit interactive performance. Deamortization is a known way to enforce an upper bound to the amount of consecutive I/O operations at the cost of extra complexity to handle interrupted reorganizations [11]. Instead, Rangemerge naturally avoids to monopolize disk I/O by applying flush operations at granularity of a single range rather than the entire memory buffer and configuring the range size through the rangefile parameter F.

7.8.4 Availability and Recovery

Availability over multiple machines is generally achieved through data replication by the datastore itself or an underlying distributed filesystem [28, 40, 32]. Durability requirements depend on the semantics and performance characteristics of applications, while data

consistency can be enforced with a quorum algorithm across the available servers [40]. We consider important the freshness of accessed data due to the typical semantics of online data serving [30]. For instance, a shopping cart should be almost instantly updated in electronic commerce, and a message should be made accessible almost immediately after it arrives in a mailbox.

At permanent server failure, a datastore recovers the lost state from redundant replicas at other servers. After transient failures, the server rebuilds index structures in volatile memory from the rangefiles and the write-ahead log. We normally log records about incoming updates and ranges that we flush to disk. Thus we recover the itemtable by replaying the log records and omitting items already flushed to rangefiles. Holding a copy of the chunkindex in the respective rangefile makes it easy to recover chunkindexes from disk. We also rebuild the rangeindex from the contents of the itemtable and the rangefiles.

7.8.5 Caching

It is possible to improve the query performance with data caching applied at the level of blocks read from disk or data items requested by users [28, 38]. We currently rely on the default page caching of the system without any sophistication related to file mapping or item caching. Prior research suggested the significance of data compaction regardless of caching [98]. We leave for future work the study of multi-level caching and dynamic memory allocation for the competing tasks of update batching and query data reuse.

7.9 Summary

After consideration of existing solutions in storage management of datastores, we point out several weaknesses related to high query latency, interference between queries and updates, and excessive reservation of storage space. To address these issues, we propose and analyze the simple yet efficient Rangemerge method and Rangetable structure. We implement our method in a prototype storage system and experimentally demonstrate that Rangemerge minimizes range query time, keeps low its sensitivity to compaction I/O, and removes the need for reserved unutilized storage space. Furthermore, under various moderate conditions Rangemerge exceeds the insertion performance of practical

write-optimized methods, and naturally exploits the key skewness of the inserted dataset.

Chapter 8

Implementation of Rangemerge in a Production System

8.1 LevelDB Implementation

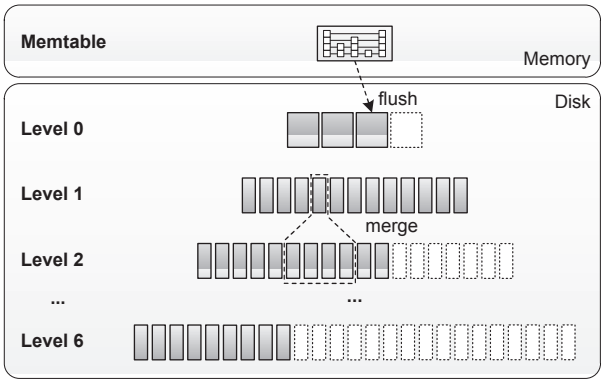
8.2 Performance Evaluation

8.3 Summary

In this section, we describe the implementation of Rangemerge in a production system. We present the design of our logging and recovery mechanisms, and go over the details of implementing Rangemerge and other compaction methods in LevelDB. Finally, we evaluate the efficiency of our logging approach and compare the performance of Rangemerge with those of related methods.

8.1 LevelDB Implementation

To study the applicability and the benefits of Rangemere on a production system, we implemented Rangemerge in LevelDB [70, 93]. LevelDB is a storage library written by Google that provides an ordered mapping from string keys to string values. It has the



Level	Files overlap	File size	Merge trigger
0	yes	4MB	4 files
1	no	2MB	10MB
2	no	2MB	100MB
3	no	2MB	1GB
4	no	2MB	10GB
5	no	2MB	100GB
6	no	2MB	1TB

Figure 8.1: Files are hierarchically organized in LevelDB. When memtable is full, it is flushed into an SSTable at level 0. Thus, level-0 files may contain overlapping ranges of keys. When the size of a level L exceeds its threshold, a single file from level L (or all level files, if $L = 0$) along with all overlapping files from level $L+1$ are merged and stored as a number of 2MB files at level $L + 1$. The maximum size of a level is expressed either as maximum number of files ($L = 0$) or as maximum total size ($L > 0$).

same general design as the BigTable tablet stack (Section 2.2.1). However, it was written from scratch in order to have no dependencies on internal Google libraries. It supports the typical `put(k,v)`, `get(k)`, `delete(k)` key-value API to modify and query the database, along with the forward and backward iterators that provide a functionality similar to range queries. In comparison to our prototype, LevelDB includes some additional useful features such as batch updates that are committed atomically, snapshots that provide a consistent read-only view of the database, logging and block checksums for durability and consistency, and compression. It is currently used as backend database for the Riak distributed datastore (Section 2.2.1) and the Google Chrome browser, but it has also been used as file system backend [88].

Incoming updates in the form of key-value pairs are inserted into a memory buffer called *memtable*, which is implemented as a skip list. When the memtable size reaches a predefined threshold (4MB by default), the memory contents are sorted, indexed and then written to disk as an *SSTable* [28]. An SSTable is an immutable file storing a sequence of key-value entries sorted by key. The set of SSTables is organized into a sequence of levels (Figure 8.1). SSTables generated from the memtable are placed into level 0. When the number of files in level 0 exceeds a certain threshold (currently 4), all level-0 files are

merged together with all overlapping level-1 files to produce a sequence of new level-1 files (a new file is created for every 2MB of data). When the total size of files at level L ($L > 0$) exceeds 10^L MB, one file from level L and all of the overlapping files from level $L + 1$ are merged to form a set of new files for level $L + 1$.

In particular, each newly received update is first appended to a log file on disk for durability and then inserted into the memtable. When the memtable is full, the system: (i) blocks incoming updates, (ii) makes the memtable read-only, (iii) creates a new memtable and a new log file for the new updates, and (iv) resumes updates. A background thread is then scheduled to compact the old memtable into a level-0 SSTable, free the memtable, delete the old log file (since its entries have been successfully persisted to disk), and perform any file merges required. If there is a system failure before a memtable is written to disk, its log can be used to recover all its entries after a system restart.

We refactored the LevelDB code so that the file merging algorithm is pluggable, and implemented the merging patterns of Nomerge, Stepped Merge Array, Geometric Partitioning, Remerge and Rangemerge. Since Rangemerge only *partially* flushes some items from memory to disk, we had to modify accordingly the memory management, logging and recovery subsystems of LevelDB.

8.1.1 Memory Management

Our implementation of Rangemerge in LevelDB maintains for each range a memtable in memory and an SSTable on disk. When the cumulative size of all memtables exceeds the memory threshold M, we select for flushing the range with the largest memtable. Updates are then temporarily blocked, until we mark the memtable as read-only and create a new memtable for this range, at which point updates are resumed. A background thread subsequently flushes the old memtable to an SSTable, frees the memory of the old memtable, and finally merges the new SSTable with the existing SSTable for that range.

8.1.2 Logging

For the logging of the updates in Rangemerge we had two options. The first one was to keep a separate log for each memtable, and delete it after we flush the memtable to disk. This logging approach simplifies log management, but causes a large number

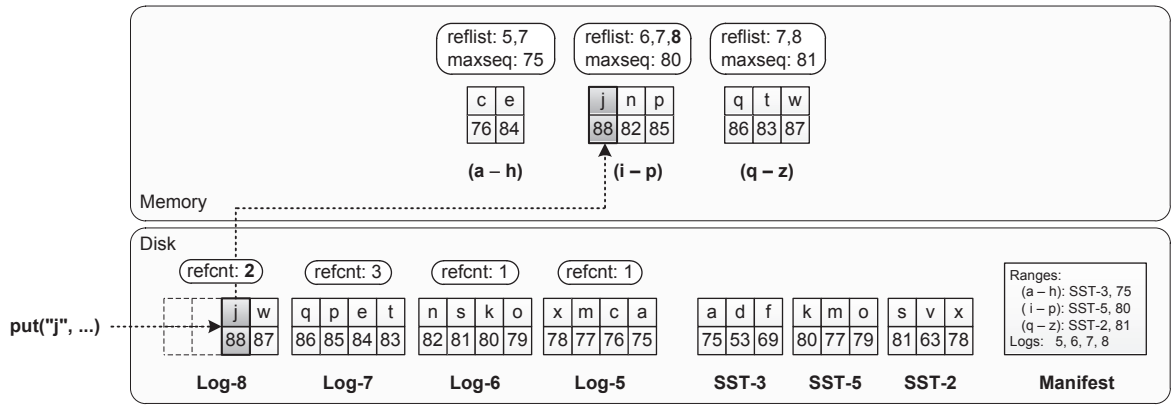


Figure 8.2: Rangemerge logging in LevelDB.

of disk seeks since each new update must be appended to the log file of the respective memtable. The second approach was to use a single log file for all memtables. This provides significant performance benefits, as the updates can be handled at sequential disk bandwidth. However, it complicates log cleaning and recovery as we have to keep track of which log entries are valid (i.e., have not been written to an SSTable) at each time. Additionally, to keep the log size bounded, the system must periodically read the log and remove any obsolete entries, a process that can easily become a performance bottleneck [90].

To combine the logging throughput of the single-log approach with the simplicity of the per-range-log approach, and additionally avoid the costs associated with garbage collection, we came up with a new logging strategy. Similar to the single-log case, updates for all memtables are append to a single file on disk. When this file reaches a predefined size (8MB), it is *sealed* and a new log file is created. For each log file we keep a reference counter (*refcounter*) that indicates the number of memtables which have entries in it (Figure 8.2). Additionally, for each memtable we maintain a list of all logs into which it has entries (*reflist*). When a new update is appended to log ℓ and added to a memtable m, we check the reflist of m; if ℓ is not present, we append ℓ to the reflist and increase the refcounter of ℓ by 1. When a memtable is flushed, we decrease by 1 the refcounters of all logs in its reflist and clear the reflist. When a refcounter for a sealed log drops to zero, we delete the log.

8.1.3 Recovery

Every time the disk state of the database changes (e.g., a file is added or deleted), LevelDB atomically updates a special *Manifest* file to reflect the new state. The Manifest file lists the set of SSTables that make up each level, the log file, and other important information. During recovery the system reads the Manifest file, deletes all files not included in it, and then recovers the memtable from the log file. We also include in the Manifest file the ranges and the respective SSTable for each range.

As shown in Figure 8.2, the log files contain all entries that are accumulated in memtables (*live* entries), as well as a number of entries that have already been stored in SSTables (*obsolete* entries). Apparently, we need a mechanism to distinguish between live and obsolete entries during log recovery. A naive approach would find for each $\langle k, v \rangle$ log entry the range R that k belongs to, and then check if the entry is stored in the SSTable of R. Since logs are not sorted by key, this approach would require at least one random I/O for each entry read, leading to unacceptably long recovery times. What we need is an efficient mechanism that quickly checks whether a log entry is stored on an SSTable or not. Additionally, the mechanism should have small memory footprint and low maintenance overhead.

LevelDB assigns a monotonically increasing sequence number to each update inserted in the system, which is stored along with the entry. Every time we flush a range to an SSTable on disk, we update a counter that stores the maximum sequence number flushed to disk for this range (*max_sequence*). This information is also persisted in the Manifest file on every range flush (Figure 8.2). After a system restart, we first recover from the Manifest file the ranges and their max sequence numbers, and create an empty memtable for each range. Then, for each entry e that is read from a log file, we find the range R it belongs to, and we compare its sequence number with the max sequence number of R. If $e.sequence \leq R.max_sequence$ then this entry was previously flushed to the SSTable of R and can thus be safely discarded. Otherwise, the entry was part of the memtable before the crash and is inserted into the memtable of R. When a range R is split into a number of ranges (typically two), we initialize the *max_sequence* for the memtables of the new ranges to the *max_sequence* of the memtable of the parent range R.

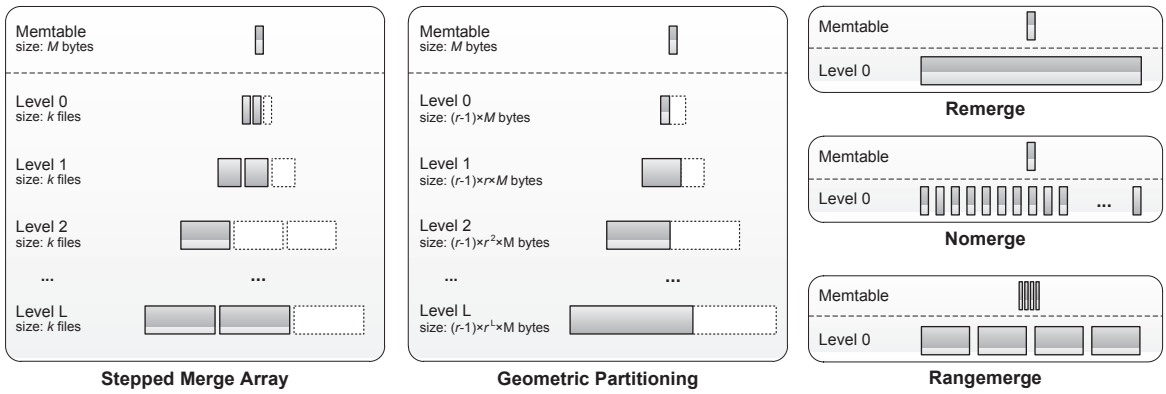


Figure 8.3: Various merging strategies, as we implemented them in LevelDB.

8.1.4 Other Merging Strategies

We have also implemented in LevelDB all the compaction managers of our prototype. All methods keep a single memtable in memory for the accumulation of new updates, except for Rangemerge which maintains a separate memtable for each range. When the total byte size of all items in memory exceeds the memory threshold M, Rangemerge compacts only the largest memtable into an SSTable at level 0; all the other methods compact their single memtable. Depending on the method, the memtable compaction may cause a cascade of merges. In case of Stepped Merge Array, if level ℓ has more than k files then all these files are merged into a new file at level $\ell + 1$. In Geometric Partitioning, the new file is first merged with the existing level-0 file. Then, if the file at level ℓ has size greater than $(r-1)r^l M$, it is merged with the respective file from level $\ell + 1$ and stored at level $\ell + 1$. Remerge always merges the new file with the single file at level 0, while Nomerge just places the file created at level 0. Rangemerge merges the new level-0 file with the existing file that corresponds to the same range, and splits the file (and the range) if its size becomes greater than F (Algorithm 6.3). In Figure 8.3, we present a high level view of the various file merging patterns that we implemented in LevelDB.

8.2 Performance Evaluation

In this section, we experimentally evaluate the efficiency of Rangemerge logging approach and study how the various compaction managers compare to each other on a full-featured

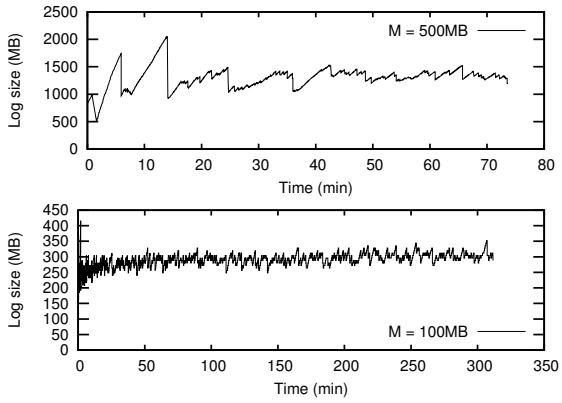
production system. The experimental environment is identical to the one described in Section 7.1. All methods use a total of $M = 512$MB for the accumulation of items in memory. We insert key-value pairs of 100 bytes keys and 1KB values, until a total of 10GB has been inserted into the system. For the experiments in which we measure the interference between inserts and queries, we issue put requests at 2500req/s and get requests at 20req/s, according to the analysis in Section 7.2. Depending on the experiment, a get request is either a range query with scan size of 10 entries, or a point query. Range queries are implemented by initializing an iterator over the LevelDB database at a specific key and reading a number of subsequent entries. Point queries (i.e. `get(k)`) return the value associated with a given key, or return "not found" if the key is not stored in the system. All methods are implemented in LevelDB v1.9, in which we disable compression for a more direct comparison to our prototype.

8.2.1 Logging Performance

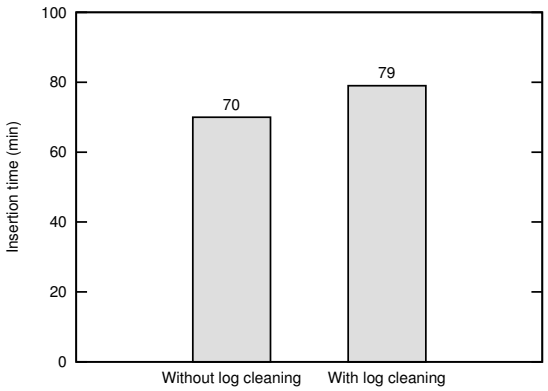
As described in Section 8.1.2, our log cleaning approach behaves lazily, in the sense that it splits the log in multiple 8MB-files and waits until a file contains no valid entries (i.e., its refcounter drops to 0) before it deletes it. This means that even in the case only a small portion of a log file corresponds to live entries, the file will still remain in the system. On the other hand, this lazy strategy enables updates to be logged at sequential disk throughput, and it totally avoids fetching log files in memory for cleaning. We now study the efficiency of our approach in terms of disk space consumed and logging throughput.

Since all memory entries must be kept in the log for durability until they are flushed to disk, the size of the log on disk is at least equal to the cumulative size M of the memtables in memory. From Figure 8.4a, the log files maintained by Rangemerge on disk take up roughly 2-3 times the memory size M, independently of the memory size. We believe that dedicating a few GB from a TB hard drive[1] for logging is a reasonable compromise for the performance we achieve, studied next. Nonetheless, in case the available disk space is scarce, we could prioritize the flushing of ranges that reference many logs or logs with low refcounters, in order to reclaim space more aggressively. We could also increase or decrease this priority depending on the size of the log on disk. We have not implemented

[1] Today, someone can can buy a 3TB disk with $120 [74].



(a) Size of log on disk for Rangemerge



(b) Insertion time for Rangemerge

Figure 8.4: (a) We show the total disk space consumed by log files in our Rangemerge implementation within LevelDB. Log size is at least equal to the memory size M, and normally between $2M$ and $3M$. (b) There is a small overhead involved in tracking the log files referenced by each range and deleting the unreferenced ones.

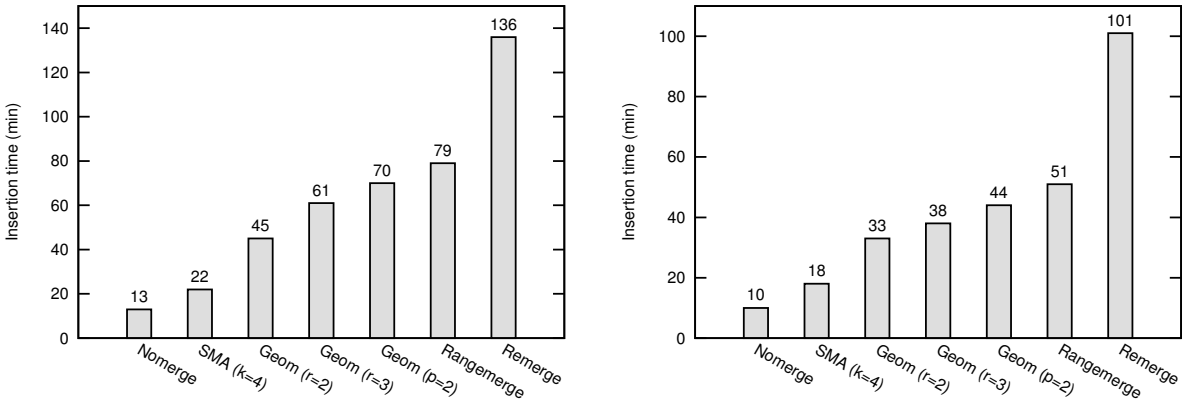
this optimization.

Figure 8.4b shows the amount of time Rangemerge requires to index 10GB of data in LevelDB. On the left bar ("Without log cleaning"), Rangemerge simply appends each incoming update on an unbounded log file, which is never garbage collected. On the right bar ("With log cleaning"), the system splits the disk log over multiple 8MB files, keeps track of the number of memtables that reference each file, and deletes a file when it is not referenced by any memtable. This increases insertion time only by 12.8%.

Overall, our log maintenance strategy requires disk space that is low considering the capacities of hard disks today, and only adds a modest overhead on insertion time compared to the case of no log cleaning at all.

8.2.2 Insertion Time

In Figure 8.5a we measure the amount of time each method requires to ingest 10GB of data in LevelDB. Nomerge requires 13min to flush the memory 20 times on disk into an equal number of files, and SMA with $k = 4$ takes 22min for 8 files. Geometric ($p = 2$) ingests the 10GB in 70min, storing each key in at most 2 files on disk during the experiment. The insertion time can be reduced by 12.8% with $r = 3$ and 35.7% with $r = 2$, at the



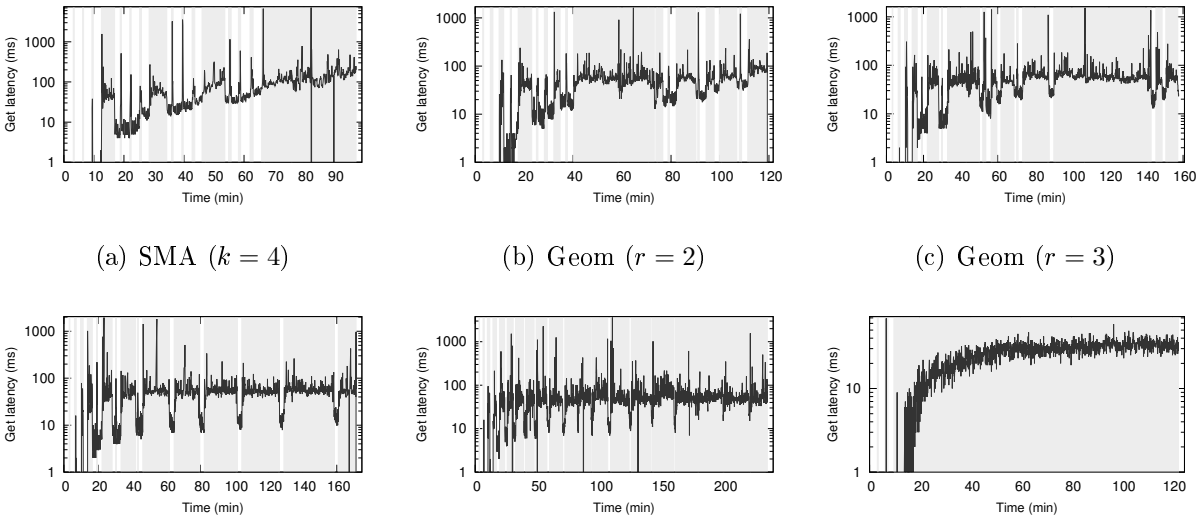
(a) LevelDB implementation

(b) Prototype storage framework

Figure 8.5: Comparison of the insertion time of various methods implemented in LevelDB and in our prototype system.

cost of increased get latency and variability due to fragmentation of keys in multiple disk files (Figures 7.1b, 7.2). Compared to Geometric ($p = 2$), Rangemerge increases insertion time by only 12.8% (79min), but keeps each key range strictly in 1 disk file. To achieve the same storage contiguity as Rangemerge, Remerge requires 136min, i.e., 72.5% more time.

In Figure 8.5b we repeat the same experiment in our prototype storage framework. From Figures 8.5a and 8.5b, we observe that the methods take longer to index the same amount of data in LevelDB than the time that they need in our prototype. This is mainly due to the fact that LevelDB involves a CPU-intensive task to compute a checksum for each block written to disk. Nevertheless, the relative differences between the methods remain similar across the two systems. One exception is Remerge, which has a smaller increase in its insertion time on LevelDB compared to the remaining methods. This is explained by the fact that Remerge (and the rest methods) simply does not complete its last compaction, as LevelDB performs the compactions asynchronously: when the last entry is inserted into the system and the memtable fills for the last time, LevelDB marks the memtable as read-only, creates a new memtable, schedules a new background thread to flush the memtable, and returns. After this write returns, our benchmark immediately exits. Consequently, the last flush of the 500MB-memtable and the merge of this file with the existing 9.5GB-file in Remerge is never performed. If we wait —as our prototype



(a) SMA ($k = 4$) (b) Geom ($r = 2$) (c) Geom ($r = 3$)

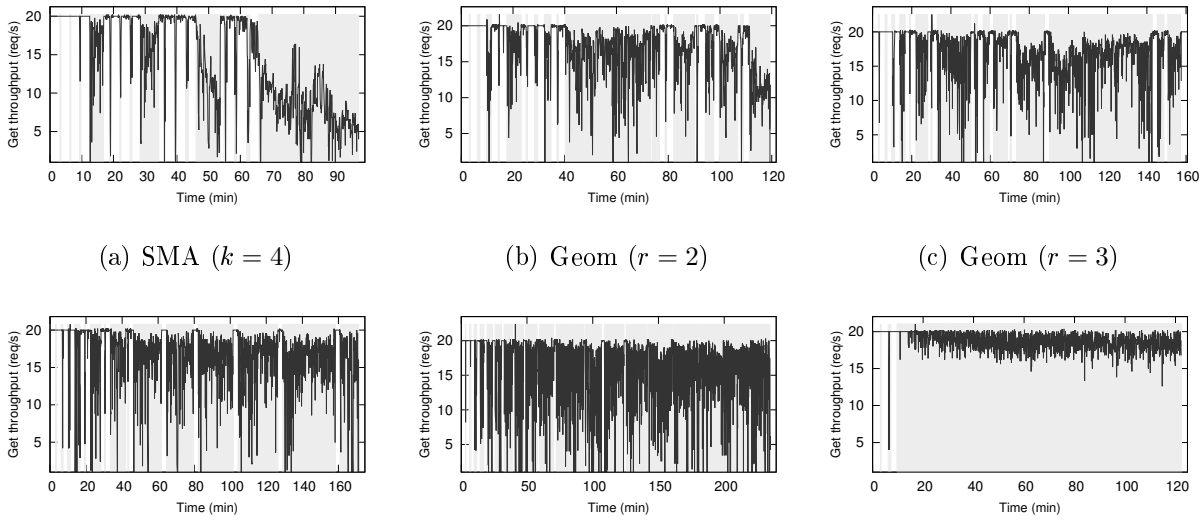
(d) Geom ($p = 2$) (e) Remerge (f) Rangemerge

Figure 8.6: Get latency in various compaction methods implemented in LevelDB, assuming a concurrent load of 2500put/s and 20get/s of scan size 10. Background compactions (gray background) severely affect queries in all methods except for Rangemerge.

does— for the completion of this last compaction, which reads a total of 10GB and writes a total 10.5GB, the insertion time of Remerge increases to 148min.

8.2.3 Interference of Queries and Inserts

We now measure the interference between the background compactions that insertions cause and the serving of the queries. Following the analysis of Section 7.2, we issue put requests at a rate of 2500req/s and range gets of scan size 10 at 20req/s. Figure 8.6 illustrates the effect that memtable flushes and file merges performed in background have on the latency of queries. As shown, these system operations (depicted as gray background) seriously impact the performance of concurrent queries in all methods except for Rangemerge, increasing range get latency to several hundred or even thousand milliseconds. In contrast, the frequent but less intensive compactions of Rangemerge allow the get latency to be kept below 50ms, improving query responsiveness by up to two orders of magnitude. Interestingly, even though Remerge stores the entries on disk contiguously in a single disk file to improve retrieval times (similar to Rangemerge), in the face of concurrent compactions it achieves no better query performance than the methods which fragment the



(a) SMA ($k = 4$) (b) Geom ($r = 2$) (c) Geom ($r = 3$)

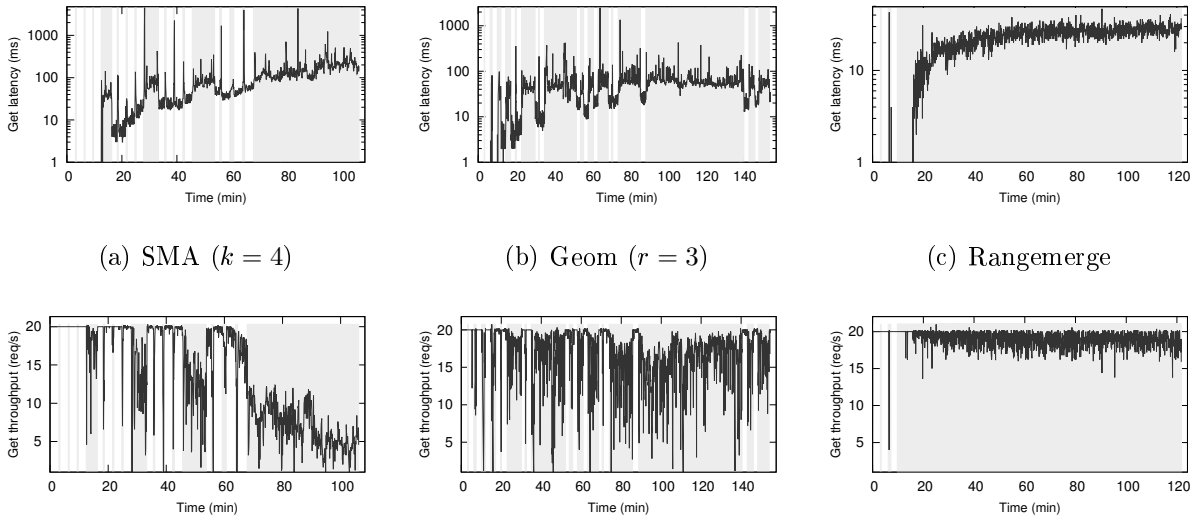
(d) Geom ($p = 2$) (e) Remerge (f) Rangemerge

Figure 8.7: Get throughput in various compaction methods implemented in LevelDB, assuming a concurrent load of 2500put/s and 20get/s of scan size 10. Rangemerge manages to keep the rate at which queries are served above 15req/s. In all remaining methods the get throughput is seriously affected during the background compactions (gray background).

entries on disk.

In Figure 8.7 we demonstrate the impact that background compactions have on the query throughput. Similar to the Figure 8.6, there is an evident correlation between these operations and the rate at which queries are served: besides Rangemerge, compactions greatly affect the query throughput in all the remaining methods, frequently yielding the system completely unresponsive with respect to query serving. Instead, Rangemerge is always responsive, keeping the query throughput above 15req/s.

To evaluate a range query, the system must create an iterator over each file that *may* contain the first key of the range, and then merge the results from these iterators. This means that the more files with overlapping keys a method maintains on disk, the greater the overhead per query will be. This is not true however for point queries, in which bloom filters can eliminate (with high probability) the need to access files that do not contain the key searched. In Figure 8.8 we use the same put and get load as in previous experiments, but use point queries instead of range queries. This means that in the vast majority of



(a) SMA ($k = 4$) (b) Geom ($r = 3$) (c) Rangemerge

(d) SMA ($k = 4$) (e) Geom ($r = 3$) (f) Rangemerge

Figure 8.8: Get latency (above) and throughput (below) for point queries in three methods, assuming puts at 2500req/s and point gets at 20req/s.

lookups we can avoid all files except from the one that contains the key. Therefore, the number of disk files is irrelevant to the query performance. Nevertheless, the results are similar to the case of range queries: background compactions cause a great increase in latency and a serious drop in throughput in all methods apart from Rangemerge.

The insertion times in Figure 8.6 are greater than those shown in Figures 8.5a mainly because in these experiments we throttle put requests at 2500req/s. Additionally, the serving of queries causes a number of random I/Os that negatively affect the concurrent sequential I/O performed by the compactions. This leads to further increase in data ingestion times. Surprisingly, the insertion time of Rangemerge (122min) is similar to Geometric with $r = 2$ (119min), and even lower than those of Geometric with $r = 3$ and $p = 2$ (158min and 171min respectively), despite the fact that these methods store each key into multiple disk files to improve write performance. This can be attributed to the fact that Rangemerge stores the entries contiguously on disk; as a result, Rangemerge minimizes the random seeks required to serve each range query and the impact of queries to concurrent compactions.

8.3 Summary

To study both the engineering effort required to implement Rangemerge in a production system and its performance benefits, we port Rangemerge and a number of representative compaction methods in Google's LevelDB key-value store. We describe the design and implementation of our logging and recovery components and experimentally evaluate its performance. We show that the implementation of Rangemerge over LevelDB is both practical and efficient: we can achieve logging at sequential disk bandwidth at the cost of a modest increase in data ingestion time and storage requirements. We then compare Rangemerge to related compaction methods that offer different tradeoffs in read and write performance. We demonstrate that Rangemerge has low sensitivity to background compactions, achieves minimal query latency, and maintains comparable or even better insertion performance than other write-optimized methods.

Chapter 9

Theoretical Analysis

9.1 I/O Complexity of Unified Range Flush

9.2 I/O Complexity of Rangemerge

9.3 Summary

In this section, we study the asymptotic behavior of the Unified Range Flush and Rangemerge methods by analyzing their I/O cost. Since these methods transfer data to and from disk in large sequential I/Os and therefore their disk access cost is negligible, we are mainly interested in estimating the amount of bytes that they transfer in the worst case when ingesting a dataset of a given size.

9.1 I/O Complexity of Unified Range Flush

For complexity comparison with existing methods of index building, we estimate the worst-case asymptotic I/O cost of our approach. We focus on the URF method because the simple flushing of the largest ranges makes the analysis more tractable. For simplicity, we assume that a termblock is not relocated when overflown.

During index building, URF allows a term list to be split across the in-place and the merge-based indices. This approach is also followed by the *non-contiguous* methods of hybrid index maintenance [24]. Accordingly, if the size of a short list during a merge exceeds the threshold value T, Büttcher et al. move the postings of the term that participate in the merge to the in-place index. They define as $\hat{L}(N,T)$ the number of postings accumulated in the in-place index, and $\hat{P}(N,T)$ the number of postings in the merge-based index for a collection of N postings. Next, they provide the following asymptotic estimates:

$$\begin{aligned} \hat{L}(N,T) &= N - c \cdot T^{(1-1/a)} \cdot N^{1/a}, \\ \hat{P}(N,T) &= c \cdot T^{(1-1/a)} \cdot N^{1/a}, \\ c &= \frac{1}{(a-1)(\gamma + \frac{1}{a-1})^{1/a}}. \end{aligned} \tag{9.1}$$

The parameter $\gamma \approx 0.577216$ is the Euler-Mascheroni constant, while a is the parameter of Zipfian distribution that models the frequency of term occurrences (e.g., $a = 1.2$).

In Equation 9.1, the counts of short and long postings result from the terms distribution rather than the method used to maintain each part of the index on disk. Therefore, if we replace T with the append threshold T_a, the above estimates also apply to the number of postings stored in the rangeblocks and termblocks of URF. In order to indicate the intuition of URF in our analysis, we use the symbols $P_{append}(N)$ and $P_{merge}(N)$ instead of the respective $\hat{L}(N,T)$ and $\hat{P}(N,T)$.

For a collection of N postings, the total I/O cost C_{total} to build the index with URF is the sum of costs for appends, C_{append}, and merges, C_{merge}:

$$\begin{aligned} C_{total}(N) &= C_{append}(N) + C_{merge}(N) \\ &= k_{append}(N) \cdot c_{append}(N) + k_{merge}(N) \cdot c_{merge}(N), \end{aligned} \tag{9.2}$$

where $k_{append}()$ and $k_{merge}()$ are the respective numbers of appends and merges, while $c_{append}()$ and $c_{merge}()$ are the respective costs per append and merge.

If a list participates in a range merge and has size greater than T_a, we append the postings of the list to a termblock on disk. After N postings have been processed, we assume that each append takes a fixed amount of time that only depends on the disk geometry and the threshold T_a:

$$c_{append}(N) \approx c_{write}(T_a) = c_{append}, \tag{9.3}$$

where $c_{write}()$ approximates the delay of a disk write. For a collection of N postings, each append flushes at least T_a postings to a termblock, so the total number of appends does not exceed $\lfloor P_{append}(N)/T_a \rfloor$:

$$k_{append}(N) \leq \left\lfloor \frac{P_{append}(N)}{T_a} \right\rfloor = \left\lfloor N \cdot \frac{1}{T_a} - N^{1/a} \cdot \frac{c}{T_a^{1/a}} \right\rfloor \in O(N). \tag{9.4}$$

Instead, a range merge involves the following steps: (i) read the rangeblock to memory, (ii) merge the disk postings with new postings in memory, and (iii) write the merged postings back to the rangeblock on disk. If the rangeblock overflows, we split it into two half-filled rangeblocks. Since a rangeblock begins 50% filled and splits when it is 100% full, we assume that a rangeblock is 75% full on average. Thus, in a posting collection of size N, the cost of a range merge can be estimated as: $c_{read}(0.75B_r) + c_{merge}(0.75B_r + p) + c_{write}(0.75B_r + p)$, where p is the number of new postings accumulated in memory for the range. The $c_{merge}()$ refers to processor activity mainly for string comparisons and memory copies; we do not consider it further because we focus on disk operations. From the merged postings of amount $0.75B_r + p$ some will be moved to termblocks because they exceed the threshold T_a. Since additionally the number p of new postings is usually small relatively to the amount of merged postings, we can also omit p and approximate $c_{merge}(N)$ with a constant:

$$c_{merge}(N) \approx c_{read}(0.75B_r) + c_{write}(0.75B_r) = c_{merge}. \tag{9.5}$$

To process a collection of N postings, we do $\lceil N/M_f \rceil$ flushes. During the i-th flush, we perform m_i range merge operations to flush a total of M_f postings. We first estimate an upper bound for m_i, before we derive an upper bound for the total number of merge operations $k_{merge}(N)$.

Suppose the posting memory is exhausted for i-th time, and we need to flush M_f postings. The URF method flushes the minimum number of ranges m_i needed to transfer M_f postings to disk. That is, it transfers the largest ranges until a total of M_f postings are flushed. In the worst-case analysis, we aim to maximize m_i. For M_p postings and R_i ranges currently in memory, m_i is maximized if the postings in memory are equally distributed across all ranges. Then, before a range is flushed to disk, the respective number of new postings accumulated in memory for the range is $p_i = M_p/R_i$. Accordingly, the number of ranges m_i flushed during the i-th flush operation is equal to: $m_i = \frac{M_f}{p_i} = \frac{M_f \cdot R_i}{M_p}$.

Just before the i-th flush, a total of $(i-1)M_f$ postings were written to disk. From them, $P_{merge}((i-1)M_f)$ postings are stored over rangeblocks. Since each rangeblock stores an average of $0.75B_r$ postings, the number of rangeblocks on disk is $P_{merge}((i-1)M_f)/0.75B_r$. The number of ranges in the rangetable just before the i-th flush will be equal to the number of rangeblocks on disk, because each range is associated with exactly one rangeblock on disk: $R_i = \frac{P_{merge}((i-1)\cdot M_f)}{0.75B_r}$.

Based on the above equations of m_i and R_i, for a collection of N postings we can derive an upper bound for the total number of range merges:

$$\begin{aligned}
k_{merge}(N) &= \sum_{i=1}^{(\#\text{ flushes})} (\#\text{ merges during } i\text{-th flush}) = \sum_{i=1}^{\lceil N/M_f \rceil} m_i \\
&= \sum_{i=1}^{\lceil N/M_f \rceil} \frac{(i-1)^{1/a} \cdot T_a^{(1-1/a)} \cdot {M_f}^{1+1/a} \cdot c}{0.75 \cdot M_p \cdot B_r} \\
&\leq \frac{T_a^{(1-1/a)} \cdot {M_f}^{1+1/a} \cdot c}{0.75 \cdot M_p \cdot B_r} \cdot \sum_{i=1}^{\lceil N/M_f \rceil} i^{1/a} \qquad (9.6) \\
&\leq \frac{T_a^{(1-1/a)} \cdot {M_f}^{1+1/a} \cdot c}{0.75 \cdot M_p \cdot B_r} \cdot \left\lceil \frac{N}{M_f} \right\rceil^{1+1/a} \\
&\approx \frac{T_a^{(1-1/a)} \cdot c}{0.75 \cdot M_p \cdot B_r} \cdot N^{1+1/a} \in O(N^{1+1/a}). \qquad (9.7)
\end{aligned}$$

According to Equations 9.2, 9.3, 9.4, 9.5 and 9.7, the total I/O cost of index building has the following upper bound:

$$C_{total}(N) \in O(N^{1+1/a}). \qquad (9.8)$$

From Table 3.1, the upper-bound index building cost of Equation 9.8 makes URF asymptotically comparable to HIM [24]. Additionally, the approach of URF to store the postings of each term across up to two sub-indices makes constant the I/O cost of term retrieval.

Special Case To cross-validate our result, we use a special case of URF to emulate the behavior of HIM [24]. We set $M_{flush} = M_{total}$ to force a full flush when we run out of memory. We also append to termblocks any list with more than T_a postings, and choose a large B_r value for URF to approximate the sequential merging of HIM. Each range merge transfers $0.75 \cdot B_r$ postings to disk. For collection size N, the total amount of postings

written to disk across $k_{merge}(N)$ merges follows from Equation 9.6:

$$\begin{aligned} P_{merge_written}(N) &= k_{merge}(N) \cdot (0.75 \cdot B_r) \\ &= \frac{T_a^{1-1/a} \cdot {M_p}^{1+1/a} \cdot c \cdot 0.75 \cdot B_r}{0.75 \cdot M_p \cdot B_r} \cdot \sum_{i=1}^{\lceil N/M_p \rceil} i^{1/a} \\ &= \sum_{i=1}^{\lceil N/M_p \rceil} c \cdot T_a^{1-1/a} (i \cdot M_p)^{1/a} \\ &\leq c \cdot T_a^{1-1/a} \cdot \frac{N^{1+1/a}}{M_p}. \end{aligned} \tag{9.9}$$

After we add the linear I/O cost from appends (Equations 9.3 and 9.4) and replace T_a with T at the right part of inequality 9.9, we estimate the worst-case cost of HIM to be that of Equation (6) by Büttcher et al. [24]. Thus we asymptotically confirm that the behavior of HIM is approximated as special case of the URF method.

9.2 I/O Complexity of Rangemerge

We aim to estimate the total amount of bytes transferred between memory and disk during the insertion of N items to the Rangetable with Rangemerge. For simplicity each item is assumed to occupy one byte. Since the rangefile size is roughly $0.5F$ after a split and cannot exceed F by design, on average it is equal to $0.75F$. Accordingly each merge operation transfers on average a total of $c_{merge} = 1.5F$ bytes, as it reads a rangefile, updates it, and writes it back to disk. For the insertion of N items, the total amount of transferred bytes is equal to $C_{total} = K \cdot c_{merge} = K \cdot \frac{3F}{2}$, where K is the number of merges. In order to estimate an upper bound on C_{total} we assume an insertion workload that maximizes K.

We call *epoch* a time period during which no range split occurs, leaving unmodified the number of ranges (and rangefiles). Let E be the number of epochs involved in the insertion of N items, and k_i be the number of merges during epoch e_i, $i = 1, ..., E$. Then the total number of merges becomes equal to $K = \sum_{i=1}^{E} k_i$.

When memory fills up for the first time, there is a single range in memory and no rangefile on disk. The first merge operation transfers all memory items to $r_1 = M/0.5F$ half-filled rangefiles, where r_i is the number of rangefiles (or ranges) during the ith epoch.

The next time memory fills up, we pick to merge the largest range in memory. In order to maximize the number of merges, we minimize the number of items in the largest range through the assumption of uniformly distributed incoming data. Then the largest range has size $s_1 = M/r_1$ items. During the ith epoch, it follows that each merge transfers to disk a range of size $s_i = M/r_i$ items.

A split initiates a new epoch, therefore a new epoch increments the number of rangefiles by one: $r_i = r_{i-1} + 1 = r_1 + i - 1$. Due to the uniform item distribution, a larger number of ranges reduces the amount s_i of items transferred to disk per merge and increases the number k_i of merges for N inserted items. If we shorten the duration of the epochs, then the number of merges will increase as a result of the higher number of rangefiles.

At a minimum, a half-filled rangefile needs to receive $0.5F$ new items before it splits. Therefore the minimum number of merges during the epoch e_i is $k_i = 0.5F/s_i$. Since an epoch flushes $0.5F$ items to disk before a split occurs, it takes $E = N/0.5F$ epochs to insert N items. From C_{total}, K, E, s_i, r_i and r_1 we find:

$$\begin{aligned} C_{total} &= \frac{3F}{2} \cdot K = \frac{3F}{2} \cdot \sum_{i=1}^{E} k_i = \frac{3F^2}{4} \cdot \sum_{i=1}^{E} \frac{1}{s_i} \\ &= \frac{3F^2}{4M} \cdot \sum_{i=1}^{E} r_i = \frac{3F^2}{4M} \sum_{i=1}^{E} (r_1 + i - 1) \\ &= \frac{3F^2}{4M} \left(E \cdot r_1 + \frac{1}{2} E(E+1) - E \right) \\ &= N^2 \frac{6}{4M} + N \left(3 - \frac{3F}{4M} \right) \in O(\frac{N^2}{M}) \end{aligned}$$

If we divide $O(\frac{N^2}{M})$ by the amount of inserted items N and the block size B, the above result becomes the $O(\frac{N}{MB})$ per-item insertion I/O complexity of the Remerge method (Table 2.1).

The above analysis of Rangemerge estimates the number of I/O operations involved in the worst case during index building. However it does not account for the cost of an individual I/O operation or the interaction of insertion I/O operations with concurrent queries. Through extensive experimentation in Chapters 7 and 8 we show that Rangemerge combines high performance in both queries and insertions because it achieves search latency comparable to or below that of the read-optimized Remerge and insertion performance comparable to that of the write-optimized methods (e.g., Geometric, Nomerge) under various conditions.

9.3 Summary

To compare the I/O complexity of our methods with related methods from literature, we estimate the worst-case asymptotic I/O cost of URF and Rangemerge. We show that the asymptotic cost of our methods matches those of existing methods with similar query performance, but as demonstrated in the previous chapters, in practice URF and Rangemerge outperform these methods.

Chapter 10

Conclusions and Future Work

10.1 Contributions

10.2 Future work

10.1 Conclusions

Motivated by the current needs of processing enormous amounts of both structured and unstructured data under stringent latency and throughput requirements, we study the related problems of text indexing and storage management at large scale. To cope with the increasing requirements in data ingestion throughput, current solutions tend to adopt a write-optimized approach that sacrifices query responsiveness for improved insertion rates. It is our thesis that these systems can achieve low query latency while maintaining high insertion throughput.

For the problem of incremental maintenance of the disk-based inverted index, we propose a simple yet innovative disk organization which groups the inverted lists on disk into disjoint lexicographical ranges and subsequently stores them in separate blocks. The lists are categorized as short or long depending on their size, and a different update policy is used for each category. We introduce two new methods, the Selective Range Flush (SRF) and the Unified Range Flush (URF), to efficiently schedule the merges of the new lists from memory with the lists on disk.

The Proteus is a prototype search engine that we develop to examine the efficiency and performance of our methods. We also propose and implement a number of optimizations for the disk and memory management. Using real-world datasets and query workloads, we show that our methods offer search latency that matches or reduces up to half the lowest achieved by existing disk-based methods and systems. In comparison to a related method of similar search latency on the same system, our methods reduce by a factor of 2.0–2.4 the I/O part of the indexing process, and by 21–24% the total indexing time.

For the storage management of datastores, we survey existing solutions from various research fields. We point out several weaknesses related to low query performance due fragmentation of entries on disk, increased variation in query latency caused by background compactions, and excessive reservation of storage space. To address these issues, we propose the Rangemerge method that replaces the periodic and intensive compactions that existing methods incur with more frequent but less intensive ones, while maintaining the storage contiguity of entries on disk.

A number of related methods along with Rangemerge are implemented in a prototype storage framework that we develop. To evaluate the practicality of Rangemerge and the generality of our results, the methods are also implemented in Google's LevelDB key-value store. Our results from both storage systems demonstrate the superior performance of Rangemerge: (i) it enables serving range queries with low latency and high rate by storing the entries contiguously on disk, and minimizes their sensitivity to background I/O by using less aggressive compactions; (ii) it maintains high insertion throughput, which is similar to or even better than those of other write-optimized methods, by selectively flushing entries from memory to disk based on their merge efficiency; (iii) it removes the need for excessive storage reservation.

10.2 Future Work

Recent reports show that the size of the structured and unstructured data accumulated increases exponentially [54]. This means that the problem of big data management that we studied will remain relevant and important at least in the following years. Here we discuss some interesting problems and research directions for future work.

Large-scale storage and indexing systems usually adopt a multi-tier architecture. Nodes from the upper tiers receive client requests and forward them to the worker nodes of the lower tiers for serving. As the performance of the worker servers is critical for the overall system performance, in this dissertation we mainly studied the efficiency of the storage management in the worker nodes. Nevertheless, orthogonal issues on the upper layers such as load balancing, replication and caching are equally important in large-scale deployments and require further investigation.

Disk capacity today is both cheap and large, so that many organizations afford to keep multiple versions of their data. Even though most users are primarily interested in the latest version the data, there are many cases where search or access over all or some previous versions would also be of interest. Examples include the Internet Archive[1] that collects, stores and offers access to historical versions of web pages from the last ten years, and a large number of companies that archive their data and analyze them to extract useful information and patterns. Designing and evaluating methods and systems to handle multi-versioned data is an interesting problem that we plan to examine.

Text indexing primarily involves parsing documents into memory postings and merging these postings with existing inverted lists on disk. We demonstrated that our methods combined with a carefully optimized implementation can reduce the merging cost of indexing by a factor of 2.0–2.9 in comparison to other methods and systems. Furthermore, increasing the available memory leads to a proportional decrease of the merging time. Nevertheless, the process of parsing does not benefit from our approach or the extra memory available on the system, as the time spent on it remains roughly the same. We are interested in studying and improving the performance of document parsing, using efficient methods that potentially exploit the multi-core CPUs and the powerful graphics processing units (GPUs) commonly found in systems today. We also plan to investigate issues related to concurrency control and handling of document modifications and deletions.

Flash SSDs have low-latency random reads and provide high throughput for sequential reads and writes. Combined with low power consumption and their declining cost, they have drawn attention to various datastore designers and developers from both the academic community and the industry. Since storing the entire dataset on SSD is usually

[1] https://archive.org/

infeasible due to impractically large costs, SSDs are typically used as an intermediate layer between RAM and HDD. We are interested in adapting the Rangemerge method and the Rangetable structure into a multi-tier storage architecture where SSDs are used complementary to HDDs as either read or write caches, and study the implications of such a design.

Bibliography

[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *ACM SOSP Symp.*, pages 1–14, Big Sky, MO, October 2009.

[2] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *ACM SIGIR Conference*, pages 372–379, Seattle, WA, August 2006.

[3] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 1(1):2–43, August 2001.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Conf.*, pages 53–64, London, UK, June 2012.

[5] Ricardo Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. Challenges on distributed web retrieval. In *IEEE Intl Conf on Data Engineering*, pages 6–20, Instanbul, Turkey, April 2007.

[6] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *ACM SIGIR Conference*, pages 183–190, Amsterdam, The Netherlands, 2007.

[7] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore : Providing scalable, highly available storage for interactive services. In *CIDR Conf.*, pages 223–234, Asilomar, CA, January 2011.

[8] Luiz Andre Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, mar/apr 2003.

[9] Alexandros Batsakis and Randal Burns. Awol: An adaptive write optimizations layer. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 67–80, San Jose, CA, February 2008.

[10] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *USENIX OSDI Symp.*, pages 47–60, Vancouver, Canada, October 2010.

[11] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *ACM SPAA Symp.*, pages 81–92, San Diego, CA, June 2007.

[12] Jon Louis Bentley and James B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1:301–358, 1980.

[13] Truls A. Bjorklund, Michaela Gotz, and Johannes Gerhke. Search in social networks with access control. In *Intl Workshop on Keyword Search on Structured Data*, Indianapolis, IN, June 2010. ACM.

[14] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, Cambridge, UK, 1998.

[15] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache Hadoop goes realtime at facebook. In *ACM SIGMOD Conf.*, pages 1071–1080, Athens, Greece, June 2011.

[16] Eric A. Brewer. Combining systems and databases: A search engine retrospective. In Joseph M. Hellerstein and Michael Stonebraker, editor, *Readings in Database Systems*, Cambridge, MA, 2005. MIT Press. Fourth Edition.

[17] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. Cache-oblivious dynamic dictionaries with up-

date/query tradeoff. In *ACM-SIAM Symp. Discrete Algorithms*, pages 1448–1456, Austin, TX, January 2010.

[18] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *ACM Conference on Information and Knowledge Management*, pages 426–434, New Orleans, LA, November 2003.

[19] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *VLDB Conference*, pages 192–202, September 1994.

[20] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. Earlybird: Real-time search at twitter. In *IEEE Intl Conference on Data Engineering*, pages 1360–1369, Washington, D.C., April 2012.

[21] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: tradeoffs in dynamic information retrieval systems. In *Proc. 14th ACM Intl. Conf. on Information and Knowledge Management (CIKM*, pages 317–318, 2005.

[22] Stefan Büttcher and Charles L. A. Clarke. Hybrid index maintenance for contiguous inverted lists. *Information Retrieval*, 11:197–207, June 2008.

[23] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. A hybrid approach to index maintenance in dynamic text retrieval systems. In *European Conference on IR Research (ECIR)*, pages 229–240, London, UK, April 2006. BCS-IRSG.

[24] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *ACM SIGIR Conference*, pages 356–363, Seattle, WA, August 2006.

[25] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, and Arild Skjolsvold et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *ACM SOSP Symp.*, pages 143–157, Cascais, Portugal, October 2011.

[26] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. ES^2: A cloud data storage system for supporting

both OLTP and OLAP. In *IEEE ICDE*, pages 291–302, Hannover, Germany, April 2011.

[27] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, December 2010.

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, Seattle, WA, November 2006.

[29] Tzicker Chiueh and Lan Huang. Efficient real-time index updates in text retrieval systems. Technical Report 66, ECSL, Stony Brook University, Stony Brook, NY, April 1999.

[30] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *ACM EuroSys Conf.*, pages 169–182, Bern, Switzerland, April 2012.

[31] The ClueWeb09 dataset, 2009. http://boston.lti.cs.cmu.edu/Data/clueweb09/.

[32] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB Conf.*, pages 1277–1288, Auckland, New Zealand, August 2008.

[33] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM SOCC Symp.*, pages 143–154, Indianapolis, IN, June 2010.

[34] Doug Cutting. Open source search. `http://www.scribd.com/doc/18004805/Lucene-Algorithm-Paper`, 2005.

[35] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *ACM SIGIR*, pages 405–411, Brussels, Belgium, September 1990.

[36] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *ACM SOCC Symp.*, pages 163–174, Indianapolis, Indiana, USA, June 2010.

[37] Savvio 10k.5 data sheet: The optimal balance of capacity, performance and power in a 10k, 2.5 inch enterprise drive, 2012.

[38] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[40] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SOSP Symp.*, pages 205–220, Stevenson, WA, October 2007.

[41] Thibault Dory, Boris Mejias, Peter Van Roy, and Nam-Luc Tran. Measuring elasticity for cloud databases. In *IARIA Intl Conf Cloud Computing, GRIDs, and Virtualization*, pages 154–160, Rome, Italy, September 2011.

[42] Bruno Dumon. Visualizing HBase flushes and compactions. `http://www.ngdata.com/site/blog/74-ng.html`, February 2011.

[43] Jonathan Ellis. Leveled compaction in Apache Cassandra. `http://www.datastax.com/dev/blog/`, June 2011.

[44] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. In *ACM SIGCOMM Conf.*, pages 25–36, Helsinki, Finland, August 2012.

[45] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, Lei Wang, Zhiguo Li, Jianfeng Zhan, Yong Qi, Yongqiang He, Shiming Gong, et al. Bigdatabench: a big data benchmark suite from web search engines. *arXiv preprint arXiv:1307.0320*, 2013.

[46] David Geer. Is it really time for real-time search? *Computer*, pages 16–19, March 2010.

[47] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 international conference on Management of data*, pages 1197–1208. ACM, 2013.

[48] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SOSP*, pages 29–43, Bolton Landing, NY, October 2003.

[49] Ruijie Guo, Xueqi Cheng, Hongbo Xu, and Bin Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Conference on Information and Knowledge Management (CIKM)*, pages 751–759, Lisboa, Portugal, November 2007.

[50] S. Gurajada and S. Sreenivasa Kumar. On-line index maintenance using horizontal partitioning. In *ACM Conference on Information and Knowledge Management*, pages 435–444, Hong Kong, China, November 2009.

[51] Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *Journal of the Americal Society for Information Science and Technology*, 54(8):713–729, 2003.

[52] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2011.

[53] `http://hypertable.org`.

[54] IDC. 2011 digital universe study: Extracting value from chaos, June 2011. http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

[55] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, pages 16–25, Athens, Greece, August 1997.

[56] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, NY, 1991.

[57] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16:417–437, October 2007.

[58] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *ACM SIGMOD Conf.*, pages 579–590, Indianapolis, IN, June 2010.

[59] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44:35–40, April 2010.

[60] Florian Leibert, Jake Mannix, Jimmy Lin, and Babak Hamadani. Automatic management of partitioned, replicated search services. In *ACM Symposium on Cloud Computing*, pages 27:1–27:8, Cascais, Portugal, October 2011.

[61] Ronnu Lempel, Yosi Mass, Shila Ofek-Koifman, Yael Petruschka, Dafna Sheinwald, and Ron Sivan. Just in time indexing for up to the second search. In *Conference on Information and Knowledge Management (CIKM)*, pages 97–106, Lisboa, Portugal, 2007.

[62] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *Conference on Information and Knowledge Management (CIKM)*, pages 776–783, Bremen, Germany, October 2005.

[63] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Trans. Database Systems (TODS)*, 33(3):1–33, August 2008.

[64] Nicholas Lester, Justin Zobel, and Hugh Williams. Efficient online index maintenance for contiguous inverted lists. *Information Processing Management*, 42(4):916–933, 2006.

[65] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In *Australasian Computer Science Conference*, pages 15–23, Dunedin, New Zeland, January 2004.

[66] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *ACM SOSP Symp.*, pages 1–13, Cascais, Portugal, October 2011.

[67] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. Dynamic maintenance of web indexes using landmarks. In *World Wide Web Conference*, pages 102–111, Budapest, Hungary, May 2003.

[68] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Keb Yocum. Stateful bulk processing for incremental analytics. In *ACM Symposium on Cloud Computing*, pages 51–62, Indianapolis, IN, June 2010.

[69] Richard Low. Cassandra under heavy write load. `http://www.acunu.com/blogs/richard-low/`, March 2011.

[70] Leveldb: A fast and lightweight key/value database library by google. `http://code.google.com/p/leveldb/`, May 2011.

[71] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with Anvil. In *ACM Symposium on Operating Systems Principles*, pages 147–160, Big Sky, MO, October 2009.

[72] Yandong Mao, Eddie Kohler, and Robert Morris. Cache craftiness for fast multicore key-value storage. In *ACM EuroSys Conf.*, pages 183–196, April 2012.

[73] Giorgos Margaritis and Stergios V. Anastasiadis. Low-cost management of inverted files for online full-text search. In *ACM CIKM*, pages 455–464, Hong Kong, China, November 2009.

[74] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, history, and grafting in the ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 151–166, New York, NY, USA, 2013. ACM.

[75] Michael McCandless, Erik Hatcher, and Otis Gospodnetić. *Lucene in action*. Manning Publications Co., Stamford, CT, 2010.

[76] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Transactions on Information Systems*, 19(3):217–241, July 2001.

[77] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *USENIX FAST Conf.*, pages 139–154, San Jose, CA, February 2012.

[78] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *ACM SIGIR Conference*, pages 191–198, Amsterdam, Netherlands, July 2007.

[79] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33:351–385, June 1996.

[80] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *ACM SOSP Symp.*, pages 29–41, Cascais, Portugal, October 2011.

[81] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *ACM SOCC Symp.*, pages 1–14, Cascais, Portugal, 2011.

[82] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[83] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, October 2010.

[84] Pouria Pirzadeh, Junichi Tatemura, Oliver Po, and Hakan Hacigümüs. Performance evaluation of range queries in key value stores. *J. Grid Computing*, 10(1):109–132, 2012.

[85] M.F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[86] Tilmann Rabl, Mohammad Sadoghi, Hans-Arno Jacobsen, Sergio Goméz-Villamor, Victor Muntés-Mulero, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. In *VLDB Conf.*, pages 1724–1735, Instanbul, Turkey, August 2012.

[87] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, New York, NY, 3 edition, 2003.

[88] Kai Ren and Garth Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 145–156, Berkeley, CA, USA, 2013. USENIX Association.

[89] Berthier Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *ACM SIGIR*, pages 105–112, Berkeley, CA, August 1999.

[90] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Computer Systems (TOCS)*, 10(1):26–52, February 1992.

[91] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A survey of large scale data management approaches in cloud environments. *IEEE Communications Surveys & Tutorials*, 2011.

[92] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *ACM European Conference on Computer Systems*, pages 267–280, Bern, Switzerland, April 2012.

[93] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *ACM SIGMOD Conf.*, pages 217–228, Scottsdale, AZ, May 2012.

[94] Sam Shah, Craig A. N. Soules, Gregory R. Ganger, and Brian D. Noble. Using provenance to aid in personal file search. In *USENIX Annual Technical Conference*, pages 171–184, Santa Clara, CA, June 2007.

[95] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[96] Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *ACM SIGMOD Conf.*, pages 765–778, Vancouver, Canada, June 2008.

[97] Adam Silberstein, Russel Sears, Wenchao Zhou, and Brian Cooper. A batch of pnuts: Experiences connecting cloud batch and serving systems. In *ACM SIGMOD Conf.*, pages 1101–1112, Athens, Greece, June 2011.

[98] Richard P. Spillane, Pradeep J. Shetty, Erez Zadok, Sagar Dixit, and Shrikar Archak. An efficient multi-tier tablet server storage architecture. In *ACM SOCC Symp.*, pages 1–14, Cascais, Portugal, October 2011.

[99] Michael Stonebraker and Rick Cattell. 10 rules for scalable performance in "simple operation" datastores. *Commun. ACM*, 54(6):72–80, June 2011.

[100] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *ACM SIGIR Conference*, pages 175–182, Amsterdam, Netherlands, July 2007.

[101] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with Project Voldemort. In *USENIX FAST*, pages 223–236, San Jose, CA, February 2012.

[102] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *ACM SIGMOD Conf.*, pages 289–300, Minneapolis, Minnesota, May 1994.

[103] TREC terabyte track, 2006. National Institute of Standards and Technology,http://trec.nist.gov/data/terabyte.html.

[104] Beth Trushkowsky, Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *USENIX FAST Conf.*, pages 163–176, San Jose, CA, February 2011.

[105] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified B-trees and versioned dictionaries. In *USENIX Hotstorage Workshop*, Portland, OR, June 2011.

[106] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Not.*, 19(5):157–167, April 1984.

[107] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Trans. Knowl. Data Eng.*, 9(3):391–409, may/jun 1997.

[108] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[109] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. In *VLDB Conf*, pages 506–514, Singapore, September 2010.

[110] Zheng Wei and Joseph JaJa. An optimized high-throughput strategy for constructing inverted files. *IEEE Transactions on Parallel and Distributed Systems*, 2012. Digital Object Identifier 10.1109/TPDS.2012.43.

[111] The Wikipedia dataset, 2008. http://static.wikipedia.org/downloads/2008-06/en/.

[112] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems*, 22(4):573–594, October 2004.

[113] Wumpus search engine (nov 10th, 2011), November 2011. http://www.wumpus-search.org.

[114] Ke Yi. Dynamic indexability and lower bounds for dynamic one-dimensional range query indexes. In *ACM PODS Symp.*, pages 187–196, Providence, RI, July 2009.

[115] The Zettair search engine, 2009. RMIT University, http://www.seg.rmit.edu.au/zettair/.

[116] M. Zhu, S. Shi, N. Yu, and J. Wen. Can phrase indexing help to process non-phrase queries. In *ACM Conference on Information and Knowledge Management*, pages 679–688, Napa Valley, CA, November 2008.

[117] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data.* McGraw-Hill Osborne Media, 2011.

[118] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.

[119] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Storage management for files of dynamic records. In *Australian Database Conference*, pages 26–38, Brisbane, Australia, 1993.

Author's Publications

Related publications:

1. **Giorgos Margaritis**, Stergios V. Anastasiadis, *Incremental Text Indexing for Fast Disk-Based Search*, ACM Transactions on the Web (TWEB), December 2013 (to appear).

2. **Giorgos Margaritis**, Stergios V. Anastasiadis, *Efficient Range-Based Storage Management for Scalable Datastores*, IEEE Transactions on Parallel and Distributed Systems (TPDS), November 2013 (to appear).

3. **Giorgos Margaritis**, Stergios V. Anastasiadis, *Low-cost Management of Inverted Files for Online Full-text Search*, ACM Conference on Information and Knowledge Management (CIKM), pages 455-464, Hong Kong, China, November 2009.

Other publications:

1. Eirini C. Micheli, **Giorgos Margaritis**, Stergios V. Anastasiadis, *Lethe: Cluster-based Indexing for Secure Multi-User Search*, IEEE International Congress on Big Data (BigData), Anchorage, Alaska, USA, June 2014 (to appear).

2. Eirini C. Micheli, **Giorgos Margaritis**, Stergios V. Anastasiadis, *Efficient Multi-User Indexing for Secure Keyword Search*, International Workshop on Privacy and Anonymity in the Information Society (PAIS) (held in conjunction with EDBT/ICDT), Athens, Greece, March 2014.

3. **Giorgos Margaritis**, Andromachi Hatzieleftheriou, Stergios V. Anastasiadis, *Nephele: Scalable Access Control for Federated File Services*, Journal of Grid Computing, pub. Springer, Volume 11, Issue 1, pp 83-102, March 2013.

Short Vita

Giorgos Margaritis was born in Thessaloniki, Greece, in 1983. He received the B.Sc in Computer Science in 2005 and the M.Sc. in Computer Science (Computer Systems) in 2008, from the Department of Computer Science, University of Ioannina, Greece. Since the end of 2008 he has been a Ph.D. candidate in the same Department under the supervision of Prof. Stergios Anastasiadis. He has been involved in two research projects and has published 3 papers in peer-review scientific journals and 2 papers in refereed conference proceedings. His research interests are in the areas of text retrieval and storage systems.