Queries on Compressed Data

by

Anurag Khandelwal

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joseph Hellerstein
Professor Marti Hearst

Fall 2019

Queries on Compressed Data

Abstract

Queries on Compressed Data

by

Anurag Khandelwal

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Low-latency, high-throughput systems for serving interactive queries are crucial to today's web services. Building such systems for today's web services is challenging due to the massive volumes of data they must cater to, and the requirement for supporting sophisticated queries (e.g., searches, filters, aggregations, regular expression matches, graph queries, etc.). Several recent approaches have highlighted the importance of in-memory storage for meeting the low-latency and high-throughput requirements, but these approaches are unable to sustain this performance when the data grows larger than DRAM capacity. Existing systems thus achieve these goals either by assuming large enough DRAM (too expensive) or by supporting only a limited set of queries (e.g., key-value stores).

In this dissertation, we explore algorithmic and data structure-driven solutions to these system design problems. We present Succinct, a distributed data store that addresses these challenges using a fundamentally new approach — executing a wide range of queries (e.g., search, random access, range, wildcard) *directly* on a compressed representation of the input data — thereby enabling efficient execution of queries on data sizes much larger than DRAM capacity. We then describe BlowFish, a system that builds on Succinct to enable a dynamic storage-performance tradeoff in data stores, providing applications the flexibility to modify the storage and performance fractionally, just enough to meet the desired goals. Finally, we explore approaches that enable even richer query semantics on compressed data, including graph queries using ZipG, a memory efficient graph store, and regular expression queries using Sprint, a query rewriting technique.

To my father.

Contents

Contents ii

List of Figures iv

List of Tables vi

1 Introduction 1
- 1.1 Limitations of Existing Approaches . . . 2
- 1.2 Thesis Overview . . . 3
- 1.3 Outline and Previously Published Work . . . 5

2 Enabling Queries on Compressed Data 6
- 2.1 Succinct Interface . . . 8
- 2.2 Querying on Compressed Data . . . 9
- 2.3 Multi-store Design . . . 14
- 2.4 Implementation . . . 17
- 2.5 Evaluation . . . 18
- 2.6 Related Work . . . 24
- 2.7 Summary . . . 25

3 Dynamic Storage-Performance Tradeoff for Compressed Data 26
- 3.1 Applications and summary of results . . . 27
- 3.2 BlowFish Techniques and Contributions . . . 28
- 3.3 BlowFish Overview . . . 28
- 3.4 BlowFish Design . . . 31
- 3.5 Evaluation . . . 37
- 3.6 Related Work . . . 46
- 3.7 Summary . . . 47

4 Interactive Queries on Compressed Graphs 48
- 4.1 Data model and Interface . . . 51
- 4.2 ZipG Design . . . 52
- 4.3 ZipG Implementation . . . 60

4.4 Evaluation . . . 64
4.5 Related Work . . . 75
4.6 Summary . . . 75

5 Executing RegEx Queries on Compressed Data **76**
5.1 Preliminaries . . . 78
5.2 Need for Sprint . . . 79
5.3 Sprint . . . 84
5.4 Related Work . . . 94
5.5 Summary . . . 95

6 Conclusions and Future Work **96**
6.1 Future Work . . . 97

A Succinct Data Structures **99**
A.1 Compression . . . 99
A.2 Query Algorithms . . . 105

Bibliography **108**

List of Figures

1.1 Existing approaches expose a hard tradeoff between scale, performance and functionality. . . . 2

2.1 Succinct interface . . . 8
2.2 Semi-structured data in Succinct . . . 9
2.3 Example for AoS, AoS2Input . . . 10
2.4 Example for Input2AoS . . . 11
2.5 Reducing space usage of AoS . . . 12
2.6 Reducing space usage of AoS2Input . . . 12
2.7 Reducing the space usage of Input2AoS . . . 13
2.8 Two-dimensional NextCharIdx representation . . . 14
2.9 Succinct multi-store architecture . . . 14
2.10 Succinct system architecture . . . 16
2.11 Succinct storage-footprint . . . 20
2.12 Succinct throughput . . . 21
2.13 Succinct latency I . . . 22
2.14 Succinct latency II . . . 22
2.15 Succinct throughput vs. latency . . . 23

3.1 BlowFish architecture . . . 29
3.2 Main idea behind BlowFish . . . 29
3.3 Layered Sampled Array (LSA) example . . . 31
3.4 Different queuing behaviors in BlowFish . . . 37
3.5 BlowFish storage-throughput tradeoff curve . . . 38
3.6 BlowFish storage-throughput tradeoff curve for other workloads . . . 39
3.7 Storage and bandwidth efficient data repair . . . 39
3.8 Handling spatial skew . . . 40
3.9 Handling temporal skew for spiked workloads . . . 42
3.10 Handling temporal skew for gradual workloads . . . 43
3.11 Evaluation of BlowFish query scheduling . . . 45

4.1 NodeFile layout in ZipG . . . 53

4.2 EdgeFile layout in ZipG . . . 53
4.3 Supporting updates using update pointers . . . 55
4.4 Function shipping in ZipG . . . 57
4.5 Data fragmentation due to graph updates . . . 60
4.6 ZipG storage footprint . . . 65
4.7 ZipG single server throughput for TAO workload . . . 65
4.8 ZipG single server throughput for LinkBench workload . . . 67
4.9 ZipG single server throughput for Graph search workload . . . 68
4.10 ZipG distributed cluster throughput . . . 70
4.11 ZipG latency for Regular Path Queries . . . 72
4.12 ZipG latency for BFS traversal . . . 74
4.13 ZipG join performance . . . 75

5.1 Performance for executing RegEx using Black-box approach . . . 77
5.2 RegExTree example . . . 79
5.3 Example for Black-box RegEx execution . . . 80
5.4 Pull-Up Union transformation . . . 83
5.5 Other Sprint transformations . . . 84
5.6 Sprint storage footprint comparison . . . 88
5.7 Sprint performance comparison against existing systems. . . . 88
5.8 Sprint performance on Apache Spark . . . 88
5.9 Sprint vs. Black-box approach for RegEx . . . 90
5.10 Storage footprint for Sprint-supported data structures . . . 91
5.11 Performance for Sprint-supported data structures (Wikipedia Dataset) . . . 92
5.12 Performance for Sprint-supported data structures (Pfam Dataset) . . . 92
5.13 Sprint latency for supported data structures . . . 93
5.14 Why Sprint works . . . 94

A.1 Lookups on AoS2Input and Input2AoS . . . 100
A.2 Sampling AoS2Input and Input2AoS . . . 100
A.3 Rank and select data structures . . . 101
A.4 Skewed wavelet tree . . . 104

List of Tables

2.1 Properties of individual stores in multi-store 15
2.2 Datasets and workloads for Succinct evaluation 19

3.1 Storage & bandwidth required for data repair during failures 28

4.1 ZipG API 50
4.2 TAO and LinkBench workloads 62
4.3 Graph search workload 62
4.4 Datasets used in ZipG evaluation 64
4.5 Graph datasets that fit in memory 66

5.1 Supported operator classes. 79
5.2 Protein signature RegEx queries 85
5.3 Text analysis RegEx queries 86

Acknowledgments

I am greatly indebted to my advisor, Ion Stoica. Whether it came to developing skills required for building large-scale systems, understanding the big picture in order to do impactful research, or working through the many personal problems that came in the way, I could always rely on Ion. I am also very grateful to Rachit Agarwal, who somehow managed to pull-off being both an unofficial co-advisor and a close friend to me. Night or day, I could bug Rachit with a research problem (or a personal one) and he would always be there. Having Ion and Rachit guide me through my PhD has been amazing — I am proud to have a research style that has been shaped by both of them.

I am also also thankful to my dissertation and qualification committee members Joseph Hellerstein, Marti Hearst, and Joseph Gonzalez; their feedback has been invaluable in not only improving my dissertation, but in me becoming a better researcher in the process.

My PhD would not have been possible without my amazing collaborators. I have been extremely lucky to have worked with some of the most motivated and brilliant people at Berkeley: Zongheng Yang, Evan Ye, Lloyd Brown, Ujval Misra, Yupeng Tang, Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, and David Patterson. The open research environment at Berkeley has also allowed me to work with extraordinary external collaborators: Aditya Akella at University of Wisconsin-Maddison, and Thomas Ristenpart, Paul Grubbs, Marie-Sarah Lacharité and Lucy Li at Cornell Tech. I also spent a summer at Microsoft Research Cambridge mentored by Dushyanth Narayanan, Aleksandar Dragojevic and Miguel Castro. All of my collaborators have shaped my research as well as my personality in unique and significant ways, and I am thankful to them for it.

I would also like to thank my mentors prior to starting graduate school: Seny Kamara and Navendu Jain at Microsoft Research Redmond, and Niloy Ganguly and Indanil Sengupta at Indian Institute of Technology, Kharagpur. Seny and Navendu were mentors to me on my first research project during a summer internship: they were the reason I discovered my love for research, and decided to apply for graduate school (with their guidance). Niloy and Indranil, my advisors during undergraduate studies, were a constant source of encouragement and support when I was deciding to pursue higher education.

Besides collaborators and mentors, my friends have had a profound impact on my life during graduate school. Despite being thousands of miles away, Devvrath Bhartia has been my rock, the best friend one could ask for. Thank you, Dev, for always being there. I could also count on Nitesh Mor, Moitrayee Bhattacharyya and Tathagatha Das (TD) to celebrate my joys and drown my sorrows. While they eventually moved on to great places, my life in Berkeley would not have been as enjoyable without Nikunj Bajaj, Sreeta Gorripaty, Neeraja Yadwadkar, Radhika Mittal, Saurabh Gupta, Shubham Tulsiyani, Bharath Hariharan, Vivek Chawda, Gautam Kumar and Qifan Pu. I also relied on a lot of help from my coworkers at Berkeley. I would be remiss if I did not mention the selfless hours Aurojit Panda, Philipp Moritz and TD put in helping me understand inner workings of their systems for my projects. I have also relied on help from the administrative and technical staff at Berkeley — Kattt

Atchley, Boban Zarkovich, Jon Kuroda and Shane Knapp — on innumerable occasions; they have always been patient and kind with all my (often unreasonable) requests.

This thesis is dedicated to my family; they have been unwavering in their support, without which my PhD would not have been possible. My parents, Santosh and Sarla, have not just been supportive of me pursuing my dreams, but have strived to provide the right environment for me to do so, often at their own expense. My brother, Abhishek, has been my role model: I have always aspired to his level of perfectionism. Whether I was a room away or thousands of miles away, my sister, Ritu, always made sure I felt loved. My *bhabhi* Shikha and my *jeeju* Sagar, have helped me with sage advice whenever I have been in a difficult situation. My tiny nephews and nieces, Aarav, Vanya and Vedant, have always managed to bring a smile to my face at any hard moment during graduate school.

Last but not least, words are not enough to express my gratitude for Shromona Ghosh, my constant companion throughout my days at Berkeley. Thank you for being in my life.

Chapter 1

Introduction

The need for interactively querying large volumes of data is ubiquitous across a wide range of cloud applications and web services, ranging from social networks [61, 193, 114], search engines [80, 25, 203], recommender systems [157, 158, 3] to genomics [105] and bioinformatics [132, 75, 150]. These applications have therefore come to rely heavily on an equally diverse set of *distributed data stores*, including document stores [44, 130], key-value stores [125, 63, 154, 142, 112, 111, 50, 55], multi-attribute NoSQL stores [102, 36, 42, 59, 175, 57], relational databases [128, 140, 147, 134], and a varied assortment of specialized stores [14, 188, 10, 189, 136].

While the application requirements may vary with their specific domains, they tend to center around three familiar aspects: *scale*, *performance*, and *functionality*. Scale in data stores typically corresponds to the massive volumes of data that they must serve queries over. For instance, Google's search index contains over 60 trillion webpages and is well over 100s of petabytes in size [87]; Facebook stores over 1.5 petabytes of profile information across more than 1 trillion records in its graph store [30]; Baidu's storage exceeds over 2000 petabytes [51]. At the same time, since these services are *user-facing* services, the backing data stores must meet stringent performance requirements, typically stated as low-latency and high-throughput constraints. For instance, Facebook [16] and Twitter [187] report throughput requirements of over 20 million queries per second, with millisecond-level latency constraints on the 99.9^{th} and 99.99^{th} percentiles even under heavy spikes in query load. Finally, queries to data stores, whether directly issued by a user or generated by the service in response to a user request, are increasingly sophisticated. These queries range from simple accesses to the data (*e.g.*, fetching individual records) to searches [87, 25], filters based on complex constraints [59, 57], regular expression matches [156, 57, 145] and even graph queries [30, 136, 189].

The design of data stores that meet the threefold requirements of scale, performance and functionality is challenging. For instance, several recent approaches [204, 55, 142, 111] have highlighted the importance of in-memory storage for satisfying the stringent low-latency and high-throughput performance constraints. How do we satisfy these constraints when the volume of data being queried grows *larger* than the DRAM capacity? Moreover, several

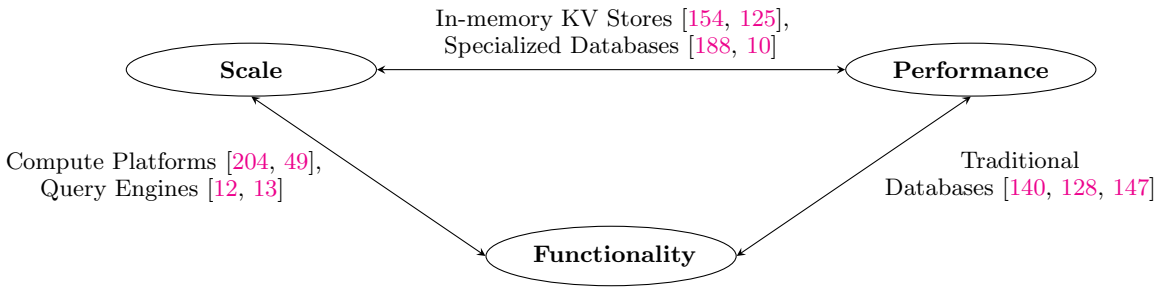


Figure 1.1: Existing approaches expose a hard tradeoff between scale, performance and functionality.

production workloads experience performance degradation due to *skew* in query load distribution even when the data fits completely in DRAM. These issues are amplified when the query load changes with time, leading to high variability in the performance. Finally, the complex access patterns of sophisticated queries such as graph traversals and regular expression matches effectively nullify the advantages of conventional systems approaches like data locality. Figure 1.1 shows that many existing approaches achieve two of the three desirable properties, but not all three.

Our key insight in this dissertation is that it is possible to address the aforementioned challenges *by operating on compressed data in memory*. In particular, we explore the design of practical systems that employ a fundamentally approach: executing a wide range of complex queries *directly* on compressed data to enable rich functionality with performance for data sizes much larger than DRAM capacity. We also show that enabling a *dynamic trade-off curve* between storage footprint and query performance can allow us to handle dynamic, skewed query workloads. Finally, we demonstrate that these techniques can be extended to complex data types (*e.g.*, graphs) and query workloads (*e.g.*, regular expression queries), extending their benefits to a wide range of modern cloud applications. In the remainder of this chapter, we outline the shortcomings of existing approaches, followed by an overview of the main techniques developed in this thesis to address these shortcomings.

1.1 Limitations of Existing Approaches

The need for scale, performance and functionality in distributed data stores has not only driven a long line of research in academia [55, 142, 111, 112, 59, 35, 121, 63], but also received much focus in industry [130, 102, 57, 154, 125, 175]. In this section, we outline two aspects in which existing data stores are unable to meet these three requirements (see Figure 1.1).

1.1.1 Querying data larger than DRAM capacity

The techniques employed by data stores to support queries on large datasets can broadly be grouped into two categories: index-based approaches, and scan-based approaches. Index-

based approaches [102, 130, 57] maintain additional data structures that speed-up queries such as searches and filters, and provide low latency and high throughput when stored in memory. However, the main drawback of indexes is their high memory footprint. Traditionally, index-based systems resort to spilling over data to significantly slower secondary storage when data grows larger than memory capacity, resulting in higher query latency and reduced throughput. On the other hand, scan-based approaches like columnar stores [103, 181] simply scan the data for every query. While this reduces the storage footprint considerably, scan-based approaches are typically associated with low throughput since they require accessing data on all of the machines in a cluster for every query. As a consequence, most existing systems either assume availability of sufficient DRAM capacity to store indexes, or compromise on query functionality by storing few or no indexes (e.g., in-memory key-value stores [154, 55, 142, 111]) to achieve low latency and high throughput.

1.1.2 Handling dynamism in query workloads

Distributed data store scale to large data volumes by partitioning their data across multiple shards, which are then spread across a cluster of servers. Dynamism in query workloads arise from two sources: due to *skew* in query load distribution across these shards, and due to *variation in skew* across the shards over time. In order to handle query skew, the conventional wisdom in data stores is to *replicate* the more popular shards, such that, the number of replicas allocated to a shard is proportional to its query load [9]. Unfortunately, this approach suffers from a number of issues. First, replication-based approaches are coarse-grained in memory allocation, and lead to wastage of DRAM capacity. This, in turn, can lead to worse performance when DRAM capacity is limited and data must now spill over to secondary storage. Second, since creating new replicas and transferring them to a less loaded server takes non-trivial amount time, these approaches do not adapt fast enough to changes in query workloads in real-world scenarios. In contrast, approaches that replicate data at finer-granularities (*e.g.*, per-record or per-object replication [202, 198]) to address these issues, need to maintain complex fine-grained metadata (i.e., at the granularity of replication), which can add both performance (*e.g.*, under rapid changes in query workloads) as well as storage overheads (*e.g.*, for datasets with many small records).

1.2 Thesis Overview

In this dissertation, we attempt to find data structure and algorithm driven solutions to the systems challenges outlined above. We focus on three main approaches to overcome the shortcomings of existing approaches: enabling queries on compressed data, exposing a dynamic storage-performance tradeoff in data-stores, and exploiting the compression structure to enable richer query semantics on compressed data.

1.2.1 Enabling Queries on Compressed Data

We address the problem of querying datasets larger than DRAM capacity using a fundamentally new approach — enabling a wide range of queries *directly* on a compressed representation of the data. We incorporate this approach in a distributed system called Succinct, which allows applications to push as much as an order of magnitude larger volume of data into memory, while supporting low-latency queries by avoiding overheads of data decompression during query execution. Succinct is able to support queries as sophisticated as random access and arbitrary substring search by embedding indexing functionality within its compressed representation.

1.2.2 Dynamic Storage-Performance Tradeoff for Compressed Data

We build on Succinct's compression techniques to address the problem of performance degradation under skew. In contrast to traditional replication-based techniques that expose a coarse-grained tradeoff between storage and performance, our approach, BlowFish, provides applications the flexibility to trade-off storage footprint for performance (and vice versa) in a fine-grained manner, just enough to meet the requirements for data under skew. BlowFish achieves this using novel data structures atop Succinct compressed representation, that enable dynamic, fine-grained adaptation of its compression factor. In fact, this dynamic, fine-grained storage-performance tradeoff in BlowFish provided a new "lens" to revisit several classical systems problems, ranging from adaptation to time-varying workloads to data repair during failures.

1.2.3 Richer Query Semantics on Compressed Data

Finally, we push the boundaries of queries that can be supported directly on compressed data in two specific directions, building on Succinct and BlowFish techniques. The first of these is ZipG, a distributed memory-efficient graph store that enables interactive queries on compressed graphs. ZipG uses a novel graph layout that transforms the input graph data into a flat unstructured file layout, which admits memory-efficient representation using compression techniques from Succinct. In addition, this layout, along with some metadata, supports efficient implementation of expressive graph queries from a wide range of real-world workloads (*e.g.*, Facebook TAO [30], LinkBench [15] and Graph Search [91]).

The second, Sprint, is a query rewriting technique that enabled efficient execution of regular expression queries on compressed data. Sprint exploits the performance characteristics of basic query primitives supported by Succinct, such as search and random access into data, to re-organize query plans for regular expression execution with the goal of latency-optimality. As such, this allows Sprint to achieve both memory-efficiency as well as low-latency for regular expression query execution. While we have implemented Sprint on Succinct, the query

rewriting techniques are general enough to work with other compressed and uncompressed data structures.

1.3 Outline and Previously Published Work

This dissertation is organized as follows. Chapter 2 introduces Succinct, a distributed data store that enables a wide range of queries directly on compressed data. Chapter 3 describes BlowFish, a system that builds on Succinct's compressed representation to admit a smooth tradeoff between storage and performance for point queries. The next two chapters focus on enabling richer query semantics on Succinct: Chapter 4 describes ZipG, a memory efficient graph store atop Succinct, and Chapter 5 presents Sprint, a query rewriting technique for efficiently executing regular expression queries on Succinct data representation. We conclude with our contributions and some possible directions for future work in Chapter 6.

Chapter 2 revises material from [4][1]. Chapter 3 revises material from [95][1]. Chapter 4[2] revises material from [97]. Finally, Chapter 5 includes material from [96][1].

[1]Work done in collaboration with Rachit Agarwal

[2]Work done in collaboration with Zongheng Yang, Evan Ye and Rachit Agarwal

Chapter 2

Enabling Queries on Compressed Data

High-performance data stores, *e.g.* document stores [44, 130], key-value stores [125, 63, 154, 142, 112, 111, 50, 55], relational databases [128, 140, 147, 134] and multi-attribute NoSQL stores [102, 36, 42, 59, 175, 57], are the bedrock of modern cloud services. While existing data stores provide efficient abstractions for storing and retrieving data using primary keys, interactive queries on values (or, secondary attributes) remains a challenge.

To support queries on secondary attributes, existing data stores can use two main techniques. At one extreme, systems such as column oriented stores, simply scan the data [166, 103]. However, data scans incur high latency for large data sizes, and have limited throughput since queries typically touch all machines[1]. At the other extreme, one can construct indexes on queried attributes [130, 102, 57]. When stored in-memory, these indexes are not only fast, but can achieve high throughput since it is possible to execute each query on a single machine. The main disadvantage of indexes is their high memory footprint. Evaluation of popular open-source data stores [130, 102] using real-world datasets (§2.5) shows that indexes can be as much as 8× larger than the input data size. Traditional compression techniques can reduce the memory footprint but suffer from degraded throughput since data needs to be decompressed even for simple queries. Thus, existing data stores either resort to using complex memory management techniques for identifying and caching "hot" data [125, 63, 130, 102] or simply executing queries off-disk or off-SSD [59]. In either case, latency and throughput advantages of indexes drop compared to in-memory query execution.

We present Succinct, a distributed data store that operates at a new point in the design space: memory efficiency close to data scans and latency close to indexes. Succinct queries on secondary attributes, however, touch all machines; thus, Succinct may achieve lower throughput than indexes when the latter fits in memory. However, due to its low memory footprint, Succinct is able to store more data in memory, avoiding latency and throughput

[1]Most data stores shard data by rows, and one needs to scan all rows. Even if data is sharded by columns, one needs to touch multiple machines to construct the row(s) in the query result.

degradation due to off-disk or off-SSD query execution for a much larger range of input sizes than systems that use indexes.

Succinct achieves the above using two key ideas. First, Succinct stores *an entropy-compressed representation of the input data* that allows random access, enabling efficient storage and retrieval of data. Succinct's data representation natively supports count, search, range and wildcard queries *without* storing indexes — all the required information is embedded within this compressed representation. Second, Succinct *executes queries directly on the compressed representation*, avoiding data scans and decompression. What makes Succinct a unique system is that it not only stores a compressed representation of the input data, but also provides functionality similar to systems that use indexes along with input data. Specifically, Succinct makes three contributions:

- Enables efficient queries directly on a compressed representation of the input data. Succinct achieves this using (1) a new data structure, in addition to adapting data structures from theory literature [82, 162, 164, 163], to compress the input data; and (2) a new query algorithm that executes random access, count, search, range and wildcard queries directly on the compressed representation (§2.2). In addition, Succinct provides applications the flexibility to tradeoff memory for faster queries and vice versa (§2.3).
- Efficiently supports data appends by chaining multiple stores, each making a different tradeoff between write, query and memory efficiency (§2.3): (1) a small log-structured store optimized for fine-grained appends; (2) an intermediate store optimized for query efficiency while supporting bulk appends; and (3) an immutable store that stores most of the data, and optimizes memory using Succinct's data representation.
- Exposes a minimal, yet powerful, API that operates on flat unstructured files (§2.1). Using this simple API, we have implemented many powerful abstractions for semi-structured data on top of Succinct including document store (*e.g.*, MongoDB [130]), key-value store (*e.g.*, Dynamo [50]), and multi-attribute NoSQL store (*e.g.*, Cassandra [102]), enabling efficient queries on both primary and secondary attributes.

We evaluate Succinct against a number of popular open-source data stores, including MongoDB [130], Cassandra [102], HyperDex [59] and DB-X, an industrial columnar store that supports queries via data scans. Evaluation results show that Succinct requires $10 - 11\times$ lower memory than data stores that use indexes, while providing similar or stronger functionality. In comparison to traditional compression techniques, Succinct's data representation achieves lower decompression throughput but supports point queries directly on the compressed representation. By pushing more data in memory and by executing queries directly on the compressed representation, Succinct achieves dramatically lower latency and higher throughput (sometimes an order of magnitude or more) compared to above systems even for moderate size datasets.

```
f = compress(file)
append(f, buffer)
buffer = extract(f, offset, len)
cnt = count(f, str)
[offset1, ...] = search(f, str)
[offset1, ...] = rangesearch(f, str1, str2)
[[offset1, len1], ...] = wildcardsearch(f, prefix, suffix, dist)
```

Figure 2.1: Interface exposed by Succinct (see §2.1).

2.1 Succinct Interface

Succinct exposes a simple interface for storing, retrieving and querying flat (unstructured) files; see Figure 2.1. We show in §2.1.1 that this simple interface already allows us to model many powerful abstractions, including the query semantics supported by popular data-stores like MongoDB [130], Cassandra [102] and BigTable [36]. Succinct therefore enables efficient queries on semi-structured data as well.

The application submits and compresses a flat `file` using `compress`; once compressed, it can invoke a set of powerful primitives directly on the compressed file. In particular, the application can append new data using `append`, can perform random access using `extract` that returns an uncompressed buffer starting at an arbitrary offset in original `file`, and count number of occurrences of any arbitrary string using `count`.

Arguably, the most powerful operation provided by Succinct is `search` which takes as an argument an *arbitrary* string (*i.e.*, not necessarily word-based) and returns offsets of all occurrences in the uncompressed `file`. For example, if `file` contains `abbcdeabczabgz`, invoking `search(f, ''ab'')` will return offsets `[0, 6, 10]`. While `search` returns an array of offsets, we provide a convenient iterator interface in our implementation. What makes Succinct unique is that `search` not only runs on the *compressed* representation but is also efficient, that is, does not require scanning the `file`.

Succinct provides two other search functions, again on arbitrary input strings. First, `rangesearch` returns the offsets of all strings between `str1` and `str2` in lexicographical order. Second, `wildcardsearch(f, prefix, suffix, dist)` returns an array of tuples. A tuple contains the offset and the length of a string with the given `prefix` and `suffix`, and whose distance between the prefix and suffix does not exceed `dist`, measured in number of input characters. Suppose again that file `f` contains `abbcdeabczabgz`, then `wildcardsearch(f, ''ab'', ''z'', 2)` will return tuples `[6, 9]` for `abcz`, and `[10, 13]` for `abgz`. Note that we do not return the tuple corresponding to `abbcdeabcz` as the distance between the prefix and suffix of this string is greater than 2.

2.1.1 Extensions for semi-structured data

Consider a logical collection of records of the form `(key, avpList)`, where `key` is a unique identifier, and `avpList` is a list of attribute value pairs, i.e., `avpList = ((attrName1,`

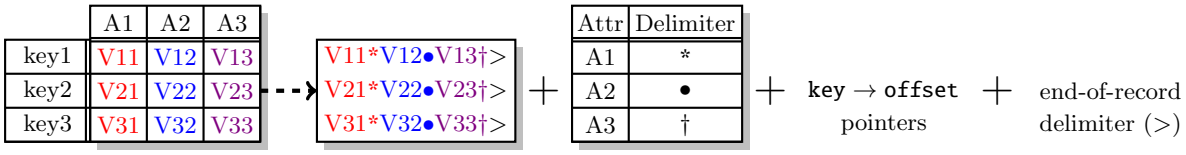


Figure 2.2: Succinct supports queries on semi-structured data by transforming the input data into flat files (see §2.1.1).

`value1),... (attrNameN, valueN))`. To enable queries using Succinct API, we encode `avpList` within Succinct data representation; see Figure 2.2. Specifically, we transform the semi-structured data into a flat file with each attribute value separated by a delimiter unique to that attribute. In addition, Succinct internally stores a mapping from each attribute to the corresponding delimiter, and a mapping from `key` to `offset` into the flat file where corresponding `avpList` is encoded.

Succinct executes `get` queries using `extract` API along with the `key`→`offset` pointers, and `put` queries using the `append` API. The `delete` queries are executed lazily, similar to [166, 148], using one explicit bit per record which is set upon record deletion; subsequent queries ignore records with set bit. Applications can also query individual attributes; for instance, search for string `val` along attribute `A2` is executed as `search(val•)` using the Succinct API, and returns every `key` whose associated attribute `A2` value matches `val`.

Flexible schema, record sizes and data types. Succinct, by mapping semi-structured data into a flat file and by using delimiters, does not impose any restriction on `avpList`. Indeed, Succinct supports single-attribute records (*e.g.*, Dynamo [50]), multiple-attribute records (*e.g.*, BigTable [36]), and even a collection of records with varying number of attributes. Moreover, using its key → offset pointers, Succinct supports the realistic case of records varying from a few bytes to a few kilobytes [16]. Succinct currently supports primitive data types (`strings, integers, floats`), and can be extended to support a variety of data structures and data types including composite types (`arrays, lists, sets`).

2.2 Querying on Compressed Data

We describe the core techniques used in Succinct. We briefly recall techniques from theory literature that Succinct uses, followed by Succinct's entropy-compressed representation (§2.2.1) and a new algorithm that operates directly on the compressed representation (§2.2.2).

Existing techniques. Classical search techniques are usually based on tries or suffix trees [185, 161]. While fast, even their *optimized representations* can require 10–20× more memory than the input size [86, 99]. Burrows-Wheeler Transform (BWT) [32] and Suffix arrays [120, 184] are two memory efficient alternatives, but still require 5× more memory than the input size [86]. FM-indexes [69, 70, 68, 67] and Compressed Suffix Arrays [82, 162, 164, 163, 81] use compressed representation of BWT and suffix arrays, respectively, to further reduce the space requirement. Succinct adapts compressed suffix arrays due to their

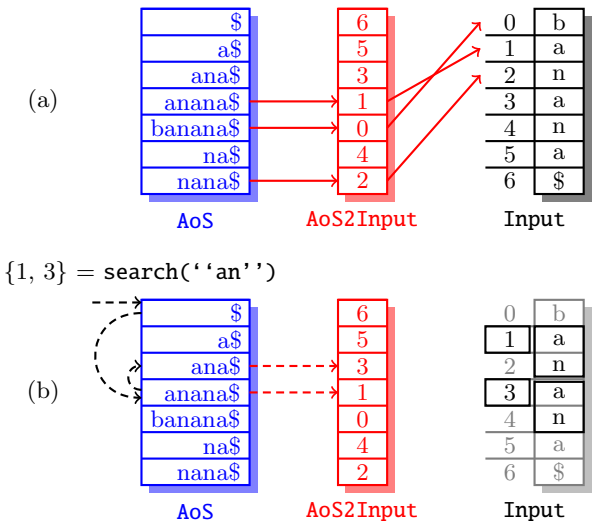


Figure 2.3: An example for input file `banana$`. AoS stores suffixes in the input in lexicographically sorted order. (a) AoS2Input maps each suffix in AoS to its location in the input (solid arrows). (b) Illustration of search using AoS and AoS2Input (dashed arrows). Suffixes being sorted, AoS allows binary search to find the smallest AoS index whose suffix starts with searched string (in this case, the suffix is "ana$" for the searched string "an"); the largest such index is found using another binary search. The result on the original input is showed on the right to aid illustration.

simplicity and relatively better performance for large datasets. We describe the basic idea behind Compressed Suffix Arrays.

Let Array of Suffixes (AoS) be an array containing all suffixes in the input file in lexicographically sorted order. AoS along with two other arrays, AoS2Input and Input2AoS[2], is sufficient to implement the search and the random access functionality without storing the input file. This is illustrated in Figure 2.3 and Figure 2.4.

Note that for a file with n characters, AoS has size $O(n^2)$ bits, while AoS2Input and Input2AoS have size $n\lceil \log n \rceil$ bits since the latter two store integers in range 0 to $n-1$. The space for AoS, AoS2Input and Input2AoS is reduced by storing only a subset of values; the remaining values are computed on the fly using a set of pointers, stored in NextCharIdx array, as illustrated in Figure 2.5, Figure 2.6 and Figure 2.7, respectively.

The NextCharIdx array is compressed using a two-dimensional representation; see Figure 2.8. Specifically, the NextCharIdx values in each column of the two-dimensional representation constitute an *increasing sequence* of integers[3]. Each column can hence be independently compressed using delta encoding [52, 144, 169].

[2]AoS2Input and Input2AoS, in this chapter, are used as convenient names for Suffix array and Inverse Suffix Array, respectively.

[3]Proof: Consider two suffixes `cX`<`cY` in a column (indexed by character `''c''`). By definition, NextCharIdx values corresponding to `cX` and `cY` store AoS indexes corresponding to suffixes `X` and `Y`. Since `cX`<`cY` implies `X`<`Y` and since AoS stores suffixes in sorted order, `NextCharIdx[cX]`<`NextCharIdx[cY]`; hence the proof.

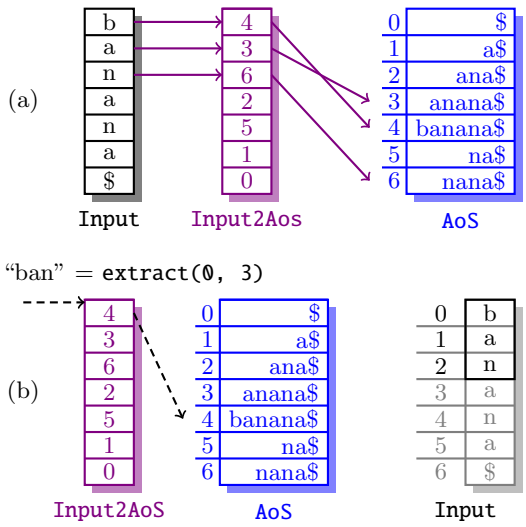


Figure 2.4: (a) The Input2AoS provides the *inverse mapping* of AoS2Input, from each index in the input to the index of the corresponding suffix in AoS (solid arrows). (b) Illustration of extract using AoS and Input2AoS (dashed arrows). The result on the original input is showed on the right to aid illustration.

2.2.1 Succinct data representation

Succinct uses the above data representation with three main differences. We give a high-level description of these differences; see Appendix A for a detailed discussion.

First, Succinct uses a more space-efficient representation of AoS2Input and Input2AoS by using a sampling by "value" strategy. In particular, for sampling rate α, rather than storing values at "indexes" $\{0, \alpha, 2\alpha, \dots\}$ as in Figure 2.6 and Figure 2.7, Succinct stores all AoS2Input values that are a multiple of α. This allows storing each sampled value `val` as `val`/α, leading to a more space-efficient representation. Using $\alpha = 2$ for example of Figure 2.6, for instance, the sampled AoS2Input values are {`6,0,4,2`}, which can be stored as {`3,0,2,1`}. Sampled Input2AoS then becomes {`1,3,2,0`} with i-th value being the index into sampled AoS2Input where i is stored. Succinct stores a small amount of additional information to locate sampled AoS2Input indexes.

Second, Succinct achieves a more space-efficient representation for NextCharIdx using the fact that values in each row of the two-dimensional representation constitute a *contiguous sequence* of integers[4]. Succinct uses its own *Skewed Wavelet Tree* data structure, based on Wavelet Trees [82, 162], to compress each row independently. Skewed Wavelet Trees allow looking up NextCharIdx value at any index without any decompression. The data structure and lookup algorithm are described in detail in Appendix A. These ideas allow Succinct to achieve 1.25–3× more space-efficient representation compared to existing techniques [81, 144, 169].

Finally, for semi-structured data, Succinct supports dictionary encoding along each at-

[4]Intuitively, any row indexed by `rowID` contains NextCharIdx values that are pointers into suffixes starting with the string `rowID`; since suffixes are sorted, these must be contiguous set of integers.

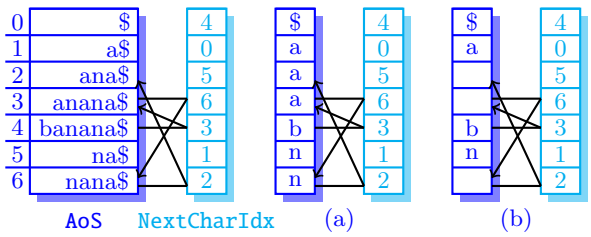


Figure 2.5: Reducing the space usage of AoS: NextCharIdx stores pointers from each suffix S to the suffix S' after removing the first character from S. (a) for each suffix in AoS, only the first character is stored. NextCharIdx pointers allow one to reconstruct suffix at any AoS index. For instance, starting from AoS[4] and following pointers, we get the original AoS entry "banana$". (b) Since suffixes are sorted, only the first AoS index at which each character occurs (*e.g.*, {`($,0),(a,1),(b,4),(n,5)`}) need be stored; a binary search can be used to locate character at any index.

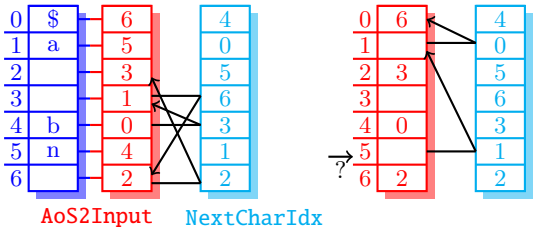


Figure 2.6: Reducing the space usage of AoS2Input. (left) Since AoS2Input stores locations of suffixes in AoS, NextCharIdx maps AoS2Input values to next larger value. That is, NextCharIdx[idx] stores the AoS2Input index that stores AoS2Input[idx]+1[5]; (right) only a few sampled values need be stored; unsampled values can be computed on the fly. For instance, starting AoS2Input[5] and following pointers twice, we get the next larger sampled value 6. Since each pointer increases value by 1, the desired value is $6 - 2 = 4$.

tribute to further reduce the memory footprint. This is essentially orthogonal to Succinct's own compression; in particular, Succinct's dictionary encodes the data along each attribute before constructing its own data structures.

2.2.2 Queries on compressed data

Succinct executes queries directly on the compressed representation from §2.2.1. We describe the query algorithm assuming access to uncompressed data structures; as discussed earlier, any value not stored in the compressed representation can be computed on the fly.

Succinct executes an **`extract`** query as illustrated in Figure 2.7 on Input2AoS representation from §2.2.1. A strawman algorithm for **`search`** would be to perform two binary searches as in Figure 2.3. However, this algorithm suffers from two inefficiencies. First, it executes binary searches on the entire AoS2Input array; and second, each step of the binary search requires computing the suffix at corresponding AoS index for comparison purposes. Succinct uses a query algorithm that overcomes these inefficiencies by aggressively exploiting the two-dimensional NextCharIdx representation.

[5]Proof: Let S be a suffix and S' be the suffix after removing first character from S. If S starts at location `loc`, then S' starts at `loc+1`. NextCharIdx stores pointers from S to S'. Since AoS2Input stores locations of

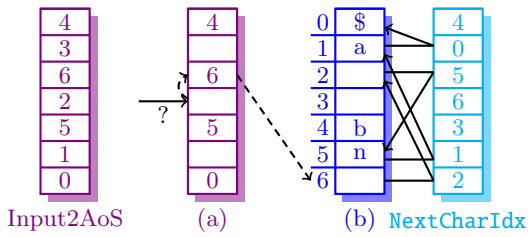


Figure 2.7: Reducing the space usage of Input2AoS. (a) only a few sampled values need be stored; (b) `extract` functionality of Figure 2.4 is achieved using sampled values and NextCharIdx. For instance, to execute `extract(3, 3)`, we find the next smaller sampled index (Input2AoS[2]) and corresponding suffix (AoS[2]="nana$"). We then remove the first character since the difference between the desired index and the closest sampled index was 1; hence the result "ana$".

Recall that the cell `(colID, rowID)` in two-dimensional NextCharIdx representation corresponds to suffixes that have `colID` as the first character and `rowID` as the following t characters. Succinct uses this to perform binary search in cells rather than the entire AoS2Input array. For instance, consider the query `search(''anan'')`; all occurrences of string `''nan''` are contained in the cell ⟨`n,an`⟩.

To find all occurrences of string `anan`, our algorithm performs a binary search only in the cell ⟨`a,na`⟩ in the next step. Intuitively, after this step, the algorithm has the indexes for which suffixes start with `''a''` and are followed by `''nan''`, the desired string. For a string of length m, the above algorithm performs $2(m-t-1)$ binary searches, two per NextCharIdx cell (see Appendix A, which is far more efficient than executing two binary searches along the entire AoS2Input array for practical values of m. In addition, the algorithm does not require computing any of the AoS suffixes during the binary searches. For a 16GB file, Succinct's query algorithm achieves a 2.3× speed-up on an average and 19× speed-up in the best case compared to the strawman algorithm.

Range and Wildcard Queries. Succinct implements `rangesearch` and `wildcardsearch` using the `search` algorithm. To implement `rangesearch(f, str1, str2)`, we find the smallest AoS index whose suffix starts with string `str1` and and the largest AoS index whose suffix starts with string `str2`. Since suffixes are sorted, the returned range of indices necessarily contain all strings that are lexicographically contained between `str1` and `str2`. To implement `wildcardsearch(f, prefix, suffix, dist)`, we first find the offsets of all prefix and suffix occurrences, and return all possible combinations such that the difference between the suffix and prefix offsets is positive and no larger than `dist` (after accounting for the prefix length).

suffixes in input, NextCharIdx maps value `loc` in AoS2Input to AoS2Input index that stores the next larger value (`loc+1`).

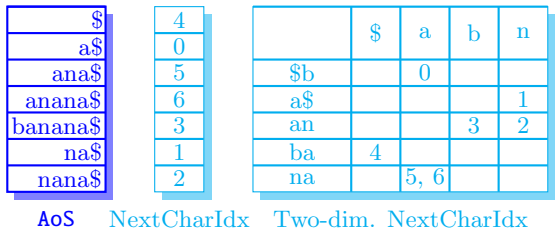


Figure 2.8: Two-dimensional NextCharIdx representation. Columns are indexed by all unique characters and rows are indexed by all unique $t-$length strings in input file, both in sorted order. A value belongs to a cell `(colID, rowID)` if corresponding suffix has `colID` as first character and `rowID` as following t characters. For instance, `NextCharIdx[3]=5` and `NextCharIdx[4]=6` are contained in cell `(a, na)`, since both start with ``a'' and have ``na'' as following two characters.

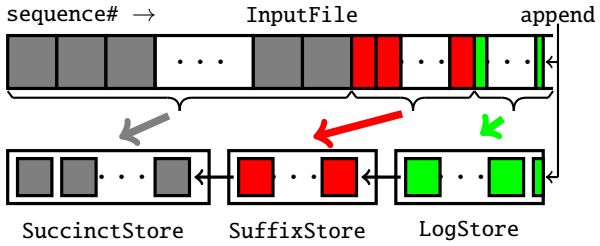


Figure 2.9: Succinct uses a write-optimized LogStore that supports fine-grained appends, a query-optimized SuffixStore that supports bulk appends, and a memory-optimized SuccinctStore. New data is appended to the end of LogStore. The entire data in LogStore and SuffixStore constitutes a single partition of SuccinctStore. The properties of each of the stores are summarized in Table 2.1.

2.3 Multi-store Design

Succinct incorporates its core techniques into a write-friendly multi-store design that chains multiple individual stores each making a different tradeoff between write, query and memory efficiency. This section describes the design and implementation of the individual stores and their synthesis to build Succinct.

Succinct design overview. Succinct chains *three* individual stores as shown in Figure 2.9; Table 2.1 summarizes the properties of the individual stores. New data is appended into a write-optimized *LogStore*, that executes queries via in-memory data scans; the queries are further sped up using an inverted index that supports fast fine-grained updates. An intermediate store, *SuffixStore*, supports bulk appends and aggregates larger amounts of data before compression is initiated. Scans at this scale are simply inefficient. SuffixStore thus supports fast queries using *uncompressed* data structures from §2.2; techniques in place ensure that these data structures do not need to be updated upon bulk appends. SuffixStore raw data is periodically transformed into an immutable entropy-compressed store *SuccinctStore* that supports queries directly on the compressed representation. The average memory footprint of Succinct remains low since most of data is contained in the memory-optimized SuccinctStore.

Table 2.1: Properties of individual stores. Data size estimated for 1TB original uncompressed data on a 10 machine 64GB RAM cluster. Memory estimated based on evaluation (§2.5).

	SuccinctStore	**SuffixStore**	**LogStore**
Stores	Comp. Data (§2.2.1)	Data + AoS2Input	Data + Inv. Index
Appends	-	Bulk	Fine
Queries	§2.2.2	Index	Scans + Inv. Index
#Machines	$n-2$	1	1
Memory	$\approx 0.4\times$	$\approx 5\times$	$\approx 9\times$

2.3.1 LogStore

LogStore is a write-optimized store that executes data `append` via main memory writes, and other queries via data scans. Memory efficiency is not a goal for LogStore since it contains a small fraction of entire dataset.

One choice for LogStore design is to let cores concurrently execute read and write requests on a single shared partition and exploit parallelism by assigning each query to one of the cores. However, concurrent writes scale poorly and require complex techniques for data structure integrity [121, 126, 111]. Succinct uses an alternative design, partitioning LogStore data into multiple partitions based on the order of writes, each containing a small amount of data. However, straightforward partitioning may lead to incorrect results if the query searches for a string that spans two partitions. LogStore thus uses *overlapping partitions*, each annotated with the starting and the ending offset corresponding to the data "owned" by the partition. The overlap size can be configured to expected string search length (default is 1MB). New data is always appended to the most recent partition. We note that although this approach effectively serializes write operations, read-only queries (*e.g.*, `search` and `extract`) can scale independent of write operations.

LogStore executes an `extract` request by reading the data starting at the offset specified in the request. While this is fast, executing `search` via data scans can still be slow, requiring tens of milliseconds even for 250MB partition sizes. Succinct avoids scanning the entire partition using an "inverted index" per partition that supports fast updates. This index maps short length (default is three character) strings to their locations in the partition; queries then need to scan characters starting only at these locations. The index is memory inefficient, requiring roughly 8× the size of LogStore data, but has little affect on Succinct's average memory since LogStore itself contains a small fraction of the entire data. The speed-up is significant allowing Succinct to scan, in practice, up to 1GB of data within a millisecond. The index supports fast updates since, upon each write, only locations of short strings in the new data need to be appended to corresponding entries in the index.

2.3.2 SuffixStore

SuffixStore is an intermediate store between LogStore and entropy-compressed SuccinctStore that serves two goals. First, to achieve good compression, SuffixStore accumulates and

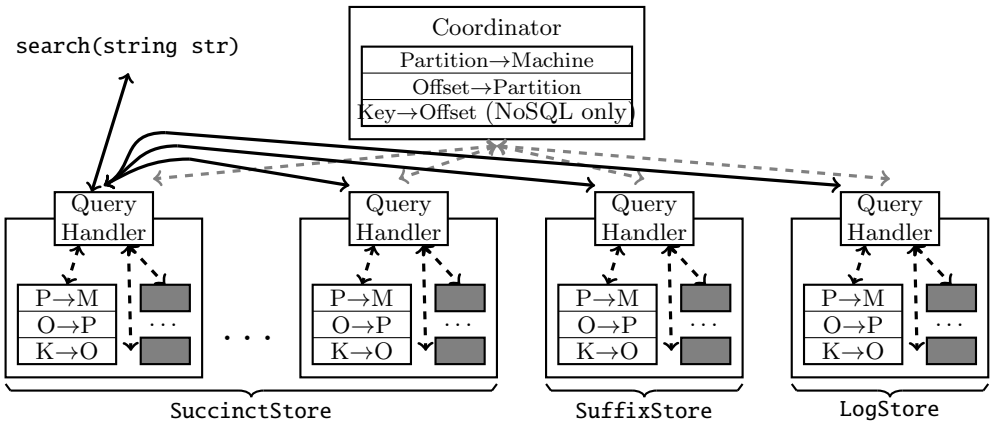


Figure 2.10: **Succinct system architecture**. Server and coordinator functionalities are described in §2.4. Each server uses a light-weight Query Handler interface to (1) interact with coordinator; (2) redirect queries to appropriate partitions and/or servers; and (3) local and global result aggregation. P→M, O→P and K→O are the same pointers as stored at the coordinator.

queries much more data than LogStore before initiating compression. Second, to ensure that LogStore size remains small, SuffixStore supports bulk data appends without updating any existing data.

Unfortunately, the LogStore approach of fast data scans with support of inverted index does not scale to data sizes in SuffixStore due to high memory footprint and data scan latency. SuffixStore thus stores an *uncompressed* AoS2Input array (§2.2); note that since the AoS2Input array stores locations of sorted suffixes in the input, the SuffixStore can execute search queries via binary search on the sorted suffixes. SuffixStore avoids explicitly storing the sorted suffixes in the AoS, by storing the uncompressed input — the sorted suffixes can be obtained using AoS2Input pointers into the input. Similarly, `extract` can be directly supported using the uncompressed input. SuffixStore achieves the second goal using a combination of partitioning and overlapping partitions, similar to LogStore. Bulk appends from LogStore are executed at partition granularity, with the entire LogStore data constituting a single partition of SuffixStore. AoS2Input is constructed per partition to ensure that bulk appends do not require updating any existing data.

2.3.3 SuccinctStore

SuccinctStore is an immutable store that contains most of the data, and is thus designed for memory efficiency. SuccinctStore uses the entropy-compressed representation from §2.2.1 and executes queries directly on the compressed representation as described in §2.2.2. SuccinctStore's design had to resolve two additional challenges.

First, Succinct's memory footprint and query latency depends on multiple tunable parameters (*e.g.*, AoS2Input and Input2AoS sampling rate and string lengths for indexing NextCharIdx rows). While default parameters in SuccinctStore are chosen to operate on a sweet spot between memory and latency, Succinct will lose its advantages if input data is

too large to fit in memory even after compression using default parameters. Second, LogStore being extremely small and SuffixStore being latency-optimized makes SuccinctStore a latency bottleneck. Hence, Succinct performance may deteriorate for workloads that are skewed towards particular SuccinctStore partitions.

Succinct resolves both these challenges by enabling applications to tradeoff memory for query latency. Specifically, Succinct enables applications to select AoS2Input and Input2AoS sampling rate; by storing fewer sampled values, lower memory footprint can be achieved at the cost of higher latency (and vice versa). This resolves the first challenge above by reducing the memory footprint of Succinct to avoid answering queries off-disk[6]. This also helps resolving the second challenge by increasing the memory footprint of overloaded partitions, thus disproportionately speeding up these partitions for skewed workloads.
We discuss data transformation from LogStore to SuffixStore and from SuffixStore to SuccinctStore in §2.4.

2.4 Implementation

We have implemented three Succinct prototypes along with extensions for semi-structured data (§2.1.1) — in Java running atop Tachyon [108], in Scala running atop Spark [204], and in C++. We discuss implementation details of the C++ prototype that uses roughly $5,200$ lines of code. The high-level architecture of our Succinct prototype is shown in Figure 2.10. The system consists of a central coordinator and a set of storage servers, one server each for LogStore and SuffixStore, and the remaining servers for SuccinctStore. All servers share a similar architecture modulo the differences in the storage format and query execution, as described in §2.2.

The coordinator performs two tasks. The first task is *membership management*, which includes maintaining a list of active servers in the system by having each server send periodic heartbeats. The second task is *data management*, which includes maintaining an up-to-date collection of pointers to quickly locate the desired data during query execution. Specifically, the coordinator maintains two set of pointers: one that maps file offsets to partitions that contain the data corresponding to the offsets, and the other one that maps partitions to machines that store those partitions. As discussed in §2.1.1, an additional set of key → offset pointers are also maintained for supporting queries on semi-structured data.

Clients connect to one of the servers via a light-weight *Query Handler* (QH) interface; the same interface is also used by the server to connect to the coordinator and to other servers in the system. Upon receiving a query from a client, the QH parses the query and identifies whether the query needs to be forwarded to a single server (for `extract` and `append` queries) or to all the other servers (for `count` and `search` queries).

In the case of an `extract` or `append` query, QH needs to identify the server to which the query needs to be forwarded. One way to do this is to forward the query to the coordinator,

[6]Empirically, Succinct can achieve a memory footprint comparable to GZip. When even the GZip-compressed data does not fit in memory, the only option for any system is to answer queries off disk.

which can then lookup its sets of pointers and forward the query to the appropriate server. However, this leads to the coordinator becoming a bottleneck. To avoid this, the pointers are cached at each server. Since the number of pointers scales only in the number of partitions and servers, this has minimal impact on Succinct's memory footprint. The coordinator ensures that pointer updates are immediately pushed to each of the servers. Using these pointers, an `extract` query is redirected to the QH of the appropriate machine, which then locates the appropriate partition and extracts the desired data.

In the case of a `search` query, the QH that receives the query from the client forwards the query to all the other QHs in the system. In turn, each QH runs multiple tasks to search all local partitions in parallel, then aggregates the results, and sends these results back to the initiator, that is, to the QH that initiated the query (see Figure 2.10). Finally, the initiator returns the aggregated result to the client. While redirecting queries using QHs reduces the coordinator load, QHs connecting to all other QHs may raise some scalability concerns. However, as discussed earlier, due to its efficient use of memory, Succinct requires many fewer servers than other in-memory data stores, which helps scalability.

Data transformation between stores. LogStore aggregates data across multiple partitions before transforming it into a single SuffixStore partition. LogStore is neither memory nor latency constrained; we expect each LogStore partition to be smaller than 250MB even for clusters of machines with 128GB RAM. Thus, AoS2Input for LogStore data can be constructed at LogStore server itself, using an efficient linear-time, linear-memory algorithm [194]. Transforming SuffixStore data into a SuccinctStore partition requires a merge sort of AoS2Input for each of the SuffixStore partitions, scanning the merged array once to construct Input2AoS and NextCharIdx, sampling AoS2Input and Input2AoS, and finally compressing each row of NextCharIdx. Succinct could use a single over-provisioned server for SuffixStore to perform this transformation at the SuffixStore server itself but currently does this in the background.

Failure tolerance and recovery. The current Succinct prototype requires manually handling: (1) coordinator failure; (2) data failure and recovery; and (3) adding new servers to an existing cluster. Succinct could use traditional solutions for maintaining multiple coordinator replicas with a consistent view. Data failure and recovery can be achieved using standard replication-based techniques. Finally, since each SuccinctStore contains multiple partitions, adding a new server simply requires moving some partitions from existing servers to the new server and updating pointers at servers. We aim to incorporate these techniques and evaluate their overheads in future work.

2.5 Evaluation

We now perform an end-to-end evaluation of Succinct's memory footprint (§2.5.1), throughput (§2.5.2) and latency (§2.5.3).

Compared systems. We evaluate Succinct using the NoSQL interface extension (§2.1.1),

Table 2.2: (left) Datasets used in our evaluation; (right) Workloads used in our evaluation. All workloads use a query popularity that follows a Zipf distribution with skewness 0.99, similar to YCSB [41].

	Size (Bytes)		#Attributes	#Records (Millions)
	Key	Value		
`smallVal`	8	≈ 140	15	123–1393
`LargeVal`	8	≈ 1300	98	19–200

	Workload	Remarks
A	100% Reads	YCSB workload C
B	95% Reads, 5% appends	YCSB workload D
C	100% Search	-
D	95% Search, 5% appends	YCSB workload E

since it requires strictly more space and operations than the unstructured file interface. We compare Succinct against several open-source and industrial systems that support search queries: MongoDB [130] and Cassandra [102] using secondary indexes; HyperDex [59] using hyperspace hashing; and an industrial columnar-store DB-X, using in-memory data scans[7].

We configured each of the system for no-failure scenario. For HyperDex, we use the dimensionality as recommended in [59]. For MongoDB and Cassandra, we used the most memory-efficient indexes. These indexes do not support substring searches and wildcard searches. HyperDex and DB-X do not support wildcard searches. Thus, the evaluated systems provide slightly weaker functionality than Succinct. Finally, for Succinct, we disabled dictionary encoding to evaluate the performance of Succinct techniques in isolation.

Datasets, Workloads and Cluster. We use two multi-attribute record datasets, one `smallVal` and one `largeVal` from Conviva customers as shown in Table 2.2. The workloads used in our evaluation are also summarized in Table 2.2. Our workloads closely follow YCSB workloads; in particular, we used YCSB to generate query keys and corresponding query frequencies, which were then mapped to the queries in our datasets (for each of read, write, and search queries). All our experiments were performed on Amazon EC2 m1.xlarge machines with 15GB RAM and 4 cores, except for DB-X where we used pre-installed r2.2xlarge instances. Each of the system was warmed up for 5 minutes to maximize the amount of data cached in available memory.

2.5.1 Memory Footprint

Figure 2.11 shows the amount of input data (without indexes) that each system fits across a distributed cluster with 150GB main memory. Succinct supports in-memory queries on data sizes larger than the system RAM; note that Succinct results do *not* use dictionary encoding and also include pointers required for NoSQL interface extensions (§2.1.1, §2.4). MongoDB and Cassandra fit roughly 10–11× less data than Succinct due to storing secondary indexes along with the input data. HyperDex not only stores large metadata but also avoids touching multiple machines by storing a copy of the entire record with each subspace, thus fitting up to 126× less data than Succinct.

[7]For HyperDex, we encountered a previously known bug [89] that crashes the system during query execution when inter-machine latencies are highly variable. For DB-X, distributed experiments require access to the industrial version. To that end, we only perform micro-benchmarks for HyperDex and DB-X for Workloads A and C.

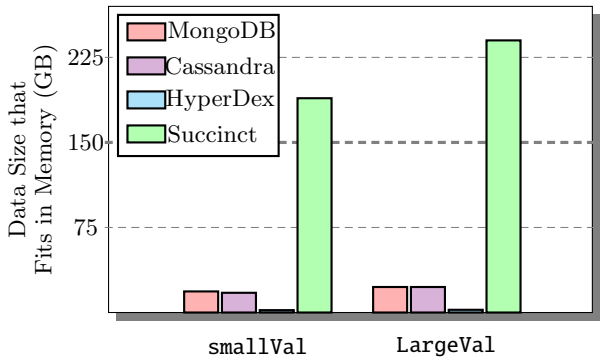


Figure 2.11: Input data size that each system fits in-memory on a distributed cluster with 150GB main memory (thick horizontal line). Succinct pushes 10-11× larger amount of data in memory compared to popular open-source data stores, while providing similar or stronger functionality.

2.5.2 Throughput

We now evaluate system throughput using a distributed 10 machine Amazon EC2 cluster. Figure 2.12 shows throughput results for `smallVal` and `LargeVal` datasets across the four workloads from Table 2.2.

Workload A. When MongoDB and Cassandra can fit datasets in memory (17GB for `smallVal` and 23GB for `LargeVal` across a 150GB RAM cluster), Succinct's relative performance depends on record size. For small record sizes, Succinct achieves higher throughput than MongoDB and Cassandra. For MongoDB, the routing server becomes a throughput bottleneck; for Cassandra, the throughput is lower because more queries are executed off-disk. However, when record sizes are large, Succinct achieves slightly lower throughput than MongoDB due to increase in Succinct's `extract` latency.

When MongoDB and Cassandra data does not fit in memory, Succinct achieves better throughput since it performs in-memory operations while MongoDB and Cassandra have to execute some queries off-disk. Moreover, we observe that Succinct achieves consistent performance across data sizes varying from tens of GB to hundreds of GB.

Workload B. MongoDB and Succinct observe reduced throughput when a small fraction of queries are append queries. MongoDB throughput reduces since indexes need to be updated upon each write; for Succinct, LogStore writes become a throughput bottleneck. Cassandra being write-optimized observes minimal reduction in throughput. We observe again that, as we increase the data sizes from 17GB to 192GB (for `SmallVal`) and from 23GB to 242GB (for `LargeVal`), Succinct's throughput remains essentially unchanged.

Workload C. For search workloads, we expect MongoDB and Cassandra to achieve high throughput due to storing indexes. However, Cassandra requires scanning indexes for search queries leading to low throughput. The case of MongoDB is more interesting. For datasets with fewer number of attributes (`SmallVal` dataset), MongoDB achieves high throughput due to caching being more effective; for `LargeVal` dataset, MongoDB search throughput

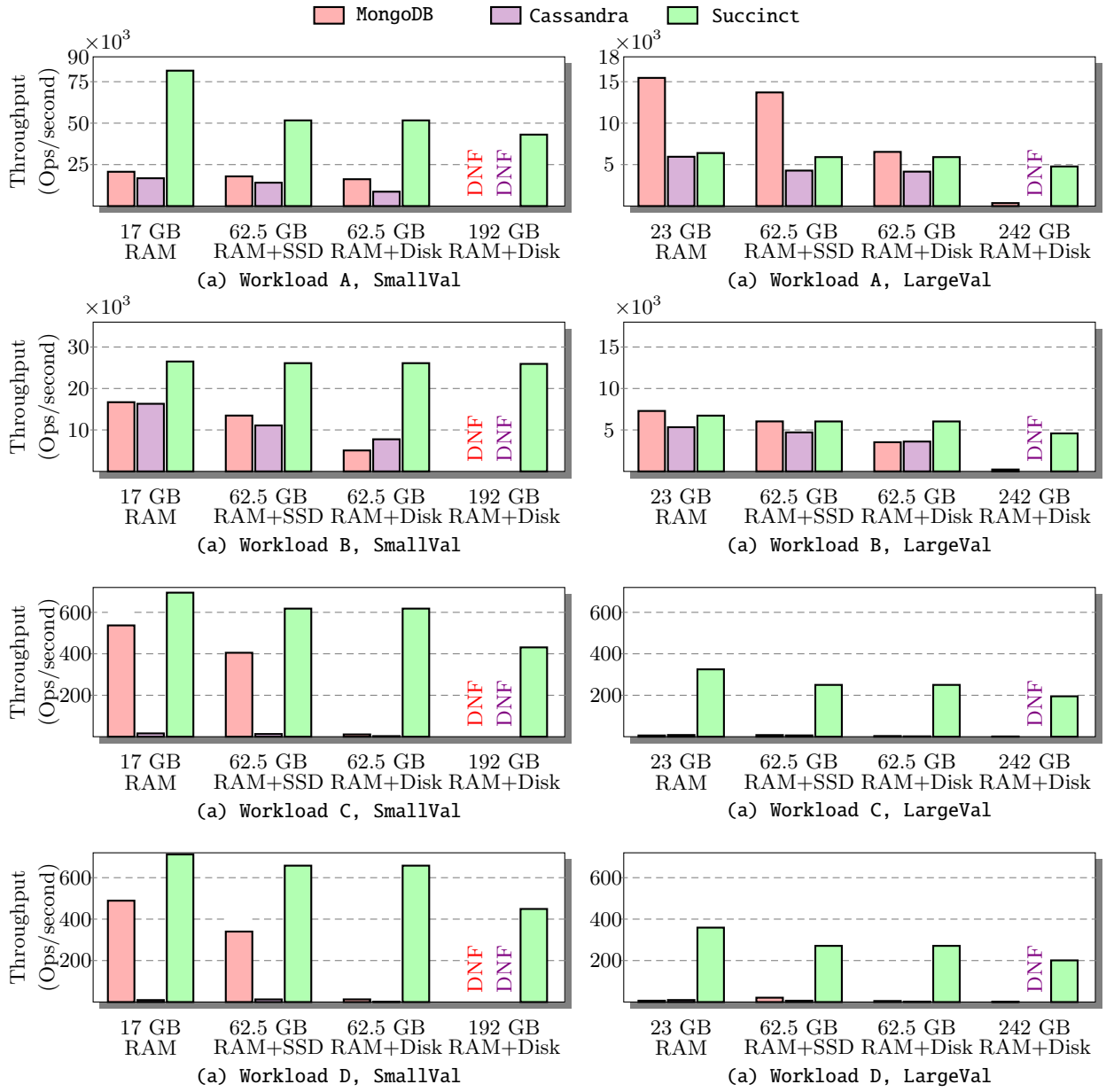


Figure 2.12: Succinct throughput against MongoDB and Cassandra for varying datasets, data sizes and workloads. MongoDB and Cassandra fit 17GB of `SmallVal` dataset and 23GB of `LargeVal` dataset in memory; Succinct fits 192GB and 242GB, respectively. `DNF` denote the experiment did not finish after 100 hours of data loading, mostly due to index construction time. Note that top four figures have different y-scales.

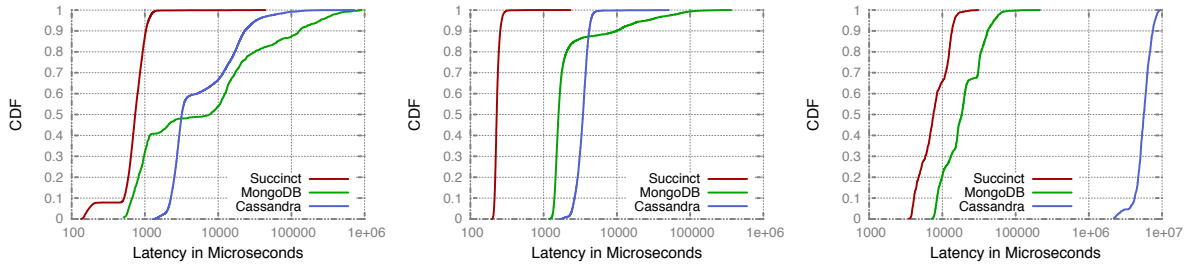


Figure 2.13: Succinct's latency for `get` (left), `put` (center) and `search` (right) against MongoDB and Cassandra for `smallVal` dataset when data and index fits in memory (best case for MongoDB and Cassandra). Discussion in §2.5.3.

Figure 2.14: Succinct's latency for `get` (left) and `search` (right) against HyperDex and DB-X for `smallVal` 10GB dataset on a single machine. HyperDex uses subspace hashing and DB-X uses in-memory data scans for search. Discussion in §2.5.3.

reduces significantly even when the entire index fits in memory. When MongoDB indexes do not fit in memory, Succinct achieves 13–134× higher throughput since queries are executed in-memory.

As earlier, even with 10× increase in data size (for both `smallVal` and `LargeVal`), Succinct throughput reduces minimally. As a result, Succinct's performance for large datasets is comparable to the performance of MongoDB and Cassandra for much smaller datasets.

Workload D. The search throughput for MongoDB and Cassandra becomes even worse as we introduce 5% appends, precisely due to the fact that indexes need to be updated upon each append. Unlike Workload B, Succinct search throughput does not reduce with appends, since writes are no more a bottleneck. As earlier, Succinct's throughput scales well with data size.

Note that the above discussion holds even when MongoDB and Cassandra use SSDs to store the data that does not fit in memory. When such is the case, throughput reduction is lower compared to the case when data is stored on disk; nevertheless, the trends remain unchanged. Specifically, Succinct is able to achieve better or comparable performance than SSD based systems for a much larger range of input values.

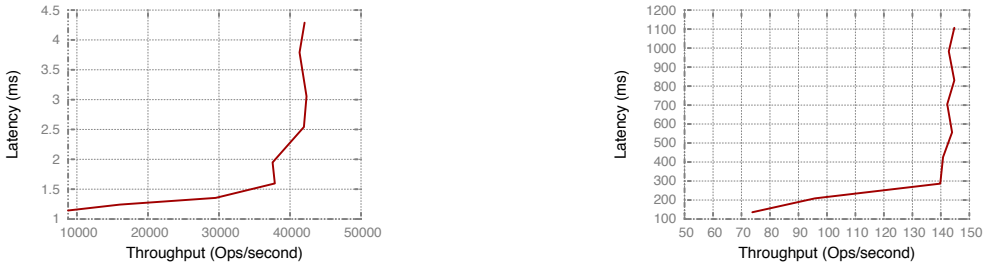


Figure 2.15: Throughput versus latency for Succinct, for `get` (left) and for `search` (right).

2.5.3 Latency

We now compare Succinct's latency against two sets of systems: (1) systems that use indexes to support queries (MongoDB and Cassandra) on a distributed 10 node Amazon EC2 cluster; and (2) systems that perform data scans along with metadata to support queries (HyperDex and DB-X) using a single-machine system. To maintain consistency across all latency experiments, we only evaluate cases where all systems (except for HyperDex) fit the entire data in memory.

Succinct against Indexes. Figure 2.13 shows that Succinct achieves comparable or better latency than MongoDB and Cassandra even when all data fits in memory. Indeed, Succinct's latency will get worse if record sizes are larger. For writes, we note that both MongoDB and Cassandra need to update indexes upon each write, leading to higher latency. For search, MongoDB achieves good latency since MongoDB performs a binary search over an in-memory index, which is similar in complexity to Succinct's search algorithm. Cassandra requires high latencies for search queries due to much less efficient utilization of available memory.

Succinct against data scans. Succinct's latency against systems that do not store indexes is compared in Figure 2.14. HyperDex achieves comparable latency for `get` queries; `search` latencies are higher since due to its high memory footprint, HyperDex is forced to answer most queries off-disk. DB-X being a columnar store is not optimized for `get` queries, thus leading to high latencies. For `search` queries, DB-X despite optimized in-memory data scans is around 10× slower at high percentiles because data scans are inherently slow.

2.5.4 Throughput versus Latency

Figure 2.15 shows the throughput versus latency results for Succinct, for both `get` and `search` queries for a fully loaded 10 machine cluster with `smallVal` 192GB dataset. The plot shows that Succinct latency and throughput results above are for the case of a fully loaded system.

2.5.5 Sequential Throughput

Our evaluation results for workload A and B used records of sizes at most 1300bytes per query. We now discuss Succinct's performance in terms of throughput for long sequential reads. We ran a simple micro-benchmark to evaluate the performance of Succinct over a single `extract` request for varying sizes of reads. Succinct achieves a constant throughput of 13Mbps using a single core single thread implementation, irrespective of the read size; the throughput increases linearly with number of threads and/or cores. This is essentially a tradeoff that Succinct makes for achieving high throughput for short reads and for search queries using a small memory footprint. For applications that require large number of sequential reads, Succinct can overcome this limitation by keeping the original uncompressed data to support sequential reads, of course at the cost of halving the amount of data that Succinct pushes into main memory. The results from Figure 2.11 show that Succinct will still push 5-5.5× more data than popular open-source systems with similar functionality.

2.6 Related Work

Succinct's goals are related to three key research areas:

Queries using secondary indexes. To support point queries, many existing data stores store indexes/metadata [130, 102, 57, 59] in addition to the original data. While indexes achieve low latency and high throughput when they fit in memory, their performance deteriorates significantly when queries are executed off-disk. Succinct requires more than 10× lower memory than systems that store indexes, thus achieving higher throughput and lower latency for a much larger range of input sizes than systems that store indexes.

Queries using data scans. Point queries can also be supported using data scans. These are memory efficient but suffer from low latency and throughput for large data sizes. Most related to Succinct is this space are columnar stores [1, 181, 48, 103, 166]. The most advanced of these [166] execute queries either by scanning data or by decompressing the data on the fly (if data compressed [195]). As shown in §2.5, Succinct achieves better latency and throughput by avoiding expensive data scans and decompression.

Theory techniques. Compressed indexes has been an active area of research in theoretical computer science since late 90s [82, 162, 164, 163, 69, 70, 68, 67]. Succinct adapts data structures from above works, but improves both the memory and the latency by using new techniques (§2.2). Succinct further resolves several challenges to realize these techniques into a practical data store: (1) efficiently handling updates using a multi-store design; (2) achieving better scalability by carefully exploiting parallelism within and across machines; and (3) enabling queries on semi-structured data by encoding the structure within a flat file.

2.7 Summary

In this chapter, we have presented Succinct, a distributed data store that supports a wide range of queries while operating at a new point in the design space between data scans (memory-efficient, but high latency and low throughput) and indexes (memory-inefficient, low latency, high throughput). Succinct achieves memory footprint close to that of data scans by storing the input data in an entropy-compressed representation that supports random access, as well as a wide range of analytical queries. When indexes fit in memory, Succinct achieves comparable latency, but lower throughput. However, due to its low memory footprint, Succinct is able to store more data in memory, avoiding latency and throughput reduction due to off-disk or off-SSD query execution for a much larger range of input sizes than systems that use indexes.

Chapter 3

Dynamic Storage-Performance Tradeoff for Compressed Data

In the previous chapter, we showed that enabling a wide range of queries on compressed data can lead to significant performance gains in data stores, due to two main reasons. First, serving queries directly on compressed data avoids the traditional performance overheads of decompressing data. Second, compression allows larger volumes of data to be resident in faster storage, *e.g.*, SSD or main memory.

While the former is unique to Succinct, the latter, i.e., compression, is often employed across a number of data stores [102, 130, 57], with the goal of executing as many queries in faster storage as possible. Often, the system architecture employed to incorporate compression is also similar to Succinct — most data stores partition the data across multiple *shards* (partitions) which are then compressed, with each server potentially storing multiple compressed shards. Shards may be replicated and cached across multiple servers and the queries are load balanced across shard replicas.

Unfortunately, compression leads to a hard tradeoff between throughput and storage for the cached shards — when stored uncompressed, a shard can support high throughput but takes a larger fraction of available cache size; and, when compressed, takes smaller cache space but also supports lower throughput. Furthermore, switching between these two extreme points on the storage-performance tradeoff space cannot be done at fine-grained time scales since it requires compression or decompression of the entire shard. Note that Succinct, as described in the previous chapter, suffers from these issues as well. Such a hard storage-performance tradeoff severely limits the ability of existing data stores in many real-world scenarios when the underlying infrastructure [151, 167], workload[21, 16, 143, 46, 199], or both changes over time. We discuss several such scenarios from real-world production clusters below (§3.1).

In this chapter, we present BlowFish, a distributed data store that builds on Succinct to enable a *smooth* storage-performance tradeoff between the two extremes (uncompressed, high throughput and compressed, low throughput), allowing fine-grained changes in storage and performance. What makes BlowFish unique is that applications can navigate from one

operating point to another along this tradeoff curve *dynamically* over fine-grained time scales. We show that, in many cases, navigating this smooth tradeoff has higher system-wide utility (*e.g.*, throughput per unit of storage) than existing techniques. Intuitively, this is because BlowFish allows shards to increase/decrease the storage "fractionally", just enough to meet the performance goals.

3.1 Applications and summary of results

BlowFish, by enabling a dynamic and smooth storage-performance tradeoff, allows us to explore several problems from real-world production clusters from a different "lens". We apply BlowFish to three such problems:

Storage and bandwidth efficient data repair during failures. Existing techniques either require high storage (replication) or high bandwidth (erasure codes) for data repair, as shown in Table 3.1. By storing multiple replicas at different points on a tradeoff curve, BlowFish can achieve the best of the two worlds — in practice, BlowFish requires storage close to erasure codes while requiring repair bandwidth close to replication. System state is restored by copying one of the replicas and navigating along the tradeoff curve. We explore the corresponding storage-bandwidth-throughput tradeoffs in §3.5.2.

Skewed workloads. Existing data stores can benefit significantly using compression [36, 102, 57, 4, 130]. However, these systems lose their performance advantages in case of dynamic workloads where (i) the set of hot objects changes rapidly over time [16, 143, 46, 199], and (ii) a single copy is not enough to efficiently serve a hot object. Studies from production clusters have shown that such workloads are a norm [21, 16, 143, 46, 199]. Selective caching [9], that caches additional replicas for hot objects, only provides coarse-grained support to handle dynamic workloads — each replica increases the throughput by 2× while incurring an additional storage overhead of 1×.

BlowFish not only provides a finer-grained tradeoff (increasing the storage overhead fractionally, just enough to meet the performance goals), but also achieves a better tradeoff between storage and throughput than selective caching of compressed objects. We show in §3.5.3 that BlowFish achieves 2.7–4.9× lower storage (for comparable throughput) and 1.5× higher throughput (for fixed storage) compared to selective caching.

Time-varying workloads. In some scenarios, production clusters delay additional replica creation to avoid unnecessary traffic (*e.g.*, for 15 minutes during transient failures [151, 167]). Such failures contribute to 90% of the failures [151, 167] and create high temporal load across remaining replicas. We show that BlowFish can adapt to such time-varying workloads even for spiked variations (as much as by 3×) by navigating along the storage-performance tradeoff in less than 5 minutes (§3.5.4).

Table 3.1: Storage and bandwidth required for erasure codes, replication and BlowFish for data repair during failure.

	Erasure (RS) Code	**Replication**	**BlowFish**
Storage	1.2×	3×	1.9×
Repair Bandwidth	10×	1×	1×

3.2 BlowFish Techniques and Contributions

BlowFish builds upon Succinct (§2), which supports queries on *compressed data*[1]. Recall that Succinct stores two *sampled* arrays, whose sampling rate acts as a proxy for the compression factor in Succinct. BlowFish introduces *Layered Sampled Array* (LSA), a new data structure that stores sampled arrays using multiple layers of sampled values. Each combination of layers in LSA correspond to a static configuration of Succinct. Layers in LSA can be added or deleted transparently, independent of existing layers and query execution, thus enabling dynamic navigation along the tradeoff curve.

Each shard in BlowFish can operate on a different point on the storage-performance tradeoff curve. This leads to several interesting problems: how should shards (within and across servers) share the available cache? How should shard replicas share requests? BlowFish adopts techniques from scheduling theory, namely the back-pressure style Join-the-shortest-queue [83] mechanism, to resolve these challenges in a unified and near-optimal manner. Shards maintain request queues that are used both to load balance queries as well as to manage shard sizes within and across servers.

In summary, this chapter makes three contributions:

- Presents the design and implementation of BlowFish, a distributed data store that enables a smooth storage-performance tradeoff, allowing fine-grained changes in storage and performance for each individual shard.
- Enables dynamic adaptation to changing workloads by navigating along the smooth tradeoff curve at fine-grained time scales.
- Uses techniques from scheduling theory to perform load balancing and shard management within and across servers.

3.3 BlowFish Overview

We briefly describe how BlowFish transforms Succinct data structures to enable the desired storage-performance tradeoff in §3.3.1. We then discuss the storage model and target workloads for BlowFish (§3.3.2). Finally, we provide a high-level overview of BlowFish design (§3.3.3).

[1]Unlike Succinct, BlowFish does *not* enforce compression; some points on the tradeoff curve may have storage comparable to systems that store indexes along with input data.

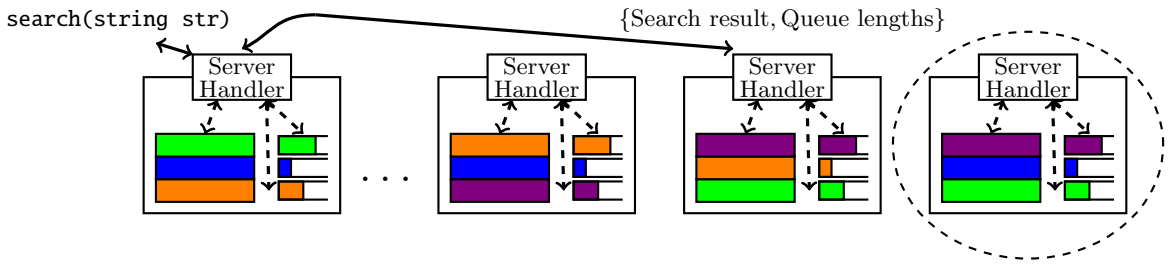


Figure 3.1: **Overall BlowFish architecture**. Each server has an architecture similar to the one shown in Figure 3.2. Queries are forwarded by Server Handlers to appropriate servers, and query responses encapsulate both results and queue lengths at that server.

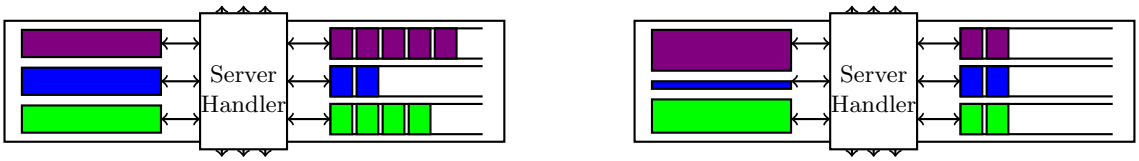


Figure 3.2: **Main idea behind BlowFish:** (left) the state of the system at some time t; (right) the state of the shards after BlowFish adapts — the shards that have longer outstanding queue lengths at time t adapt their storage footprint to a larger one, thus serving larger number of queries per second than at time t; the shards that have smaller outstanding queues, on the other hand, adapt their storage footprint to a smaller one thus matching the respective load.

3.3.1 Sampled Arrays: Storage versus Performance

Recall that Succinct reduces the space requirements of Input2AoS and AoS2Input using *sampling* — only a few sampled values (*e.g.*, for sampling rate α, value at indexes `0`, α, `2`α, `..`) from these two arrays are stored. NextCharIdx allows computing unsampled values during query execution.

The tradeoff is that for a sampling rate of α, the storage requirement for Input2AoS and AoS2Input is $2n\lceil \log n \rceil / \alpha$ and the number of operations required for computing each unsampled value is α.

Succinct thus has a fixed small storage cost for AoS and NextCharIdx, and the sampling rate α acts as a proxy for overall storage and performance in Succinct.

3.3.2 BlowFish data model and assumptions

BlowFish enables the same functionality as Succinct (§3.3.1) — support for random access and search queries on flat unstructured files, with extensions for key-value stores and NoSQL stores.

Assumptions. BlowFish makes two assumptions. First, *systems are limited by the capacity of faster storage*, that is, they operate on data sizes that do not fit entirely into the fastest storage. Indeed, indexes to support search queries along with the input data makes it hard to fit the entire data in fastest storage especially for purely in-memory data stores (*e.g.*,

Redis [154], MICA [111], RAMCloud [142]). Second, BlowFish assumes that data can be sharded in a manner that a query does not require touching each server in the system. Most real-world datasets and query workloads admit such sharding schemes [143, 46, 199].

3.3.3 BlowFish Design Overview

BlowFish uses a system architecture similar to existing data stores, *e.g.*, Cassandra [102] and ElasticSearch [57]. Specifically, BlowFish comprises of a set of servers that store the data as well as execute queries (see Figure 3.1). Each server shares a similar design, comprised of multiple data shards (§3.4.1), a *request queue* per shard that keeps track of outstanding queries, and a special module called the *server handler* that triggers navigation along the storage-performance curve and schedules queries (§3.4.2).

Each shard admits the desired storage-performance tradeoff using *Layered Sampled Array* (LSA), a new data structure that allows transparently changing the sampling factor α for Input2AoS and AoS2Input over fine-grained time scales. Smaller values of α indicate higher storage requirements, but also lower latency (and vice versa). Layers can be added and deleted without affecting existing layers or query execution thus enabling dynamic navigation along the tradeoff curve. We describe LSA and the layer addition-deletion process in LSA in §3.4.1.

BlowFish allows each shard to operate at a different operating point on the storage-performance tradeoff curve (see Figure 3.2). Such a flexibility comes at the cost of increased dynamism and heterogeneity in system state. Shards on a server can have varying storage footprint and as a result, varying throughput. Moreover, storage footprint and throughput may vary across shard replicas. How should shards (within and across servers) share the available cache? How should shard replicas share requests? When should a shard trigger navigation along the storage-performance tradeoff curve?

BlowFish adopts a technique from scheduling theory, the Join-the-shortest-queue [83] mechanism, to resolve the above questions in a unified manner. BlowFish servers maintain a *request queue* per shard, that stores outstanding requests for the respective shard. A server handler module periodically monitors request queues for local shards, maintains information about request queues across the system, schedules queries and triggers navigation along the storage-performance tradeoff curve.

Upon receiving a query from a client for a particular shard, the server handler forwards the query to the shard replica with shortest request queue length. All incoming queries are enqueued in the request queue for the respective shard. When the load on a particular shard is no more than its throughput at the current operating point on the storage-performance curve, the queue length remains minimal. On the other hand, when the load on the shard increases beyond the supported throughput, the request queue length for this shard increases (see Figure 3.2 (left)). Once the request queue length crosses a certain threshold, the navigation along the tradeoff curve is triggered either using the remaining storage on the server or by reducing the storage overhead of a relatively lower loaded shard. BlowFish internally implements a number of optimizations for selecting navigation triggers, maintaining request

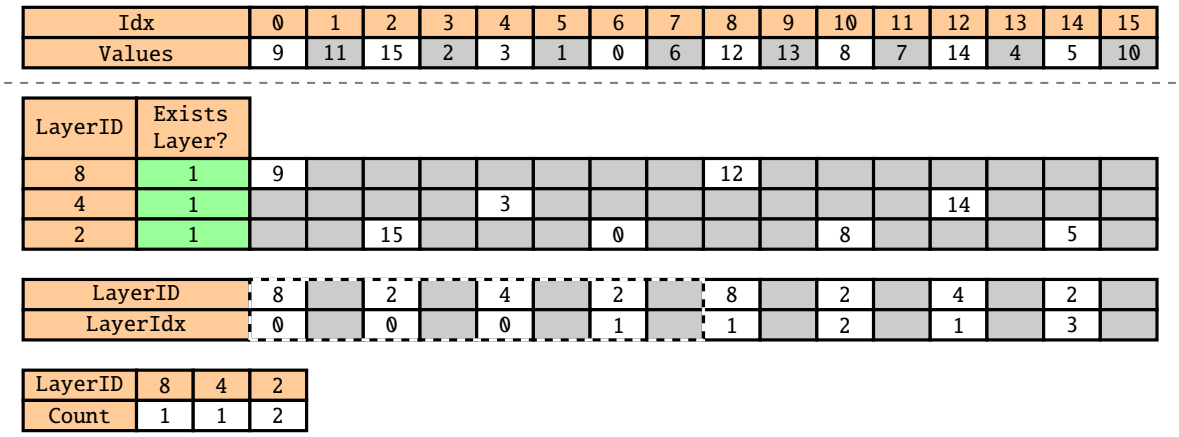


Figure 3.3: Illustration of *Layered Sampled Array* (LSA). The original unsampled array is shown above the dashed line (gray values indicate unsampled values). In LSA, each layer stores values for sampling rate given by `LayerID`, modulo values that are already stored in upper layers (in this example, sampling rates $8, 4, 2$). Layers are added and deleted at the bottom; that is, `LayerID=2` will be added if and only if all layers with sampling rate $4, 8, 16, ..$ exist. Similarly, `LayerID=2` will be the first layer to be deleted. The `ExistsLayer` bitmap indicates whether a particular layer exists (1) or not (0). `LayerID` and `ExistsLayer` allow checking whether or not value at any index `idx` is stored in LSA — we find the largest existing `LayerID` that is a proper divisor of `idx`. Note that among every consecutive 8 values in original array, 1 is stored in topmost layer, 1 in the next layer and 2 in the bottommost layer. This observation allows us to find the index into any layer `LayerIdx` where the corresponding sampled value is stored.

hysteresis to avoid unnecessary oscillations along the tradeoff curve, storage management during navigation and ensuring correctness in query execution during the navigation. We discuss these design details in §3.4.2.

3.4 BlowFish Design

We start with the description of Layered Sampled Array (§3.4.1) and then discuss the system details (§3.4.2).

3.4.1 Layered Sampled Array

BlowFish enables a smooth storage-performance tradeoff using a new data structure, Layered Sampled Array (LSA), that allows dynamically changing the sampling factor in the two sampled arrays — Input2AoS and AoS2Input. We describe LSA below.

Consider an array `A`, and let `SA` be another array that stores a set of *sampled-by-index* values from `A`. That is, for *sampling rate* α, `SA[idx]=A[`$\alpha\times$`idx]`. For instance, if `A = {6, 4, 3, 8, 9, 2}`, the sampled-by-index array with sampling rate 4 and 2 are $\texttt{SA}_4$ `= {6, 9}` and $\texttt{SA}_2$ `= {6, 3, 9}`, respectively.

LSA emulates the functionality of SA, but stores the sampled values in multiple *layers*, together with a few auxiliary structures (Figure 3.3). Layers in LSA can be added or deleted transparently without affecting the existing layers. Addition of layers results in higher storage (lower sampling rate α) and lower query latency; layer deletion, on the other hand, reduces

the storage but also increases the query latency. Furthermore, looking up a value in LSA is *agnostic* to the existing layers, independent of how many and which layers exist (see Algorithm 1). This allows BlowFish to navigate along the storage-performance curve without any change in query execution semantics compared to Succinct.

Algorithm 1 `LookupLSA`

```
1: procedure GetLayerID (idx) ▷ Get the layer ID given the index into the sampled array; α is the sampling rate.
2:     return LayerID[idx % α]

3: procedure GetLayerIdx(idx)  ▷ Get the index into LayerID given the index into the sampled array; α is the sampling rate.
4:     count ← Count[LayerID(idx)]
5:     return count × (idx / α) + LayerIdx[idx % α]

6: procedure LookupLSA (idx)                                  ▷ Performs lookup on the LSA.
7:     if IsSampled(idx) then
8:         l_id ← GetLayerID(idx)                             ▷ Get layer ID.
9:         l_idx ← GetLayerIdx(idx)                           ▷ Get index into layer.
10:        return SampledArray[l_id][l_idx]
```

Layer Addition. The design of LSA as such allows arbitrary layers (in terms of sampling rates) to coexist; furthermore, layers can be added or deleted in arbitrary order. However, our implementation of LSA makes two simplifications. First, layers store sampled values for indexes that are *powers of two*. Second, new layers are always added in a manner that ensures the sampling rate of the new layer is double the sampling rate of the last added layer. The rationale is that these two simplifications induce a certain structure in LSA, that makes the increase in storage footprint as well as time taken to add the layer very predictable. In particular, under the assumption that the unsampled array is of length $n = 2^k$ for some integer k, the number of sampled values stored at any layer is equal to the cumulative number of sampled values stored in the layers added before it (see Figure 3.3). If the sampling rate for the new layer is α, then this layer stores precisely $n/2\alpha$ sampled values; thus, the increase in storage becomes predictable. Moreover, since the upper layers constitute sampling rate 2α, computing each value in the new layer requires 2α operations (§3.3.1). Hence, adding a layer takes a fixed amount of time independent of the sampling rate of layer being added.

BlowFish supports two modes for creating new layers. In *dedicated layer construction*, the space is allocated for a new layer[2] and dedicated threads populate values in the layer; once all the values are populated the `ExistsLayer` bit is set to 1. The additional compute resources required in dedicated layer construction may be justified if the time spent in populating the new layer is smaller than the period of increased throughput experienced by the shard(s). However, such may not be the case for many scenarios.

[2]using free unused cache or by deleting layers from relatively lower loaded shards, as described in §3.4.2.

Algorithm 2 `CreateLayerOpportunistic`

1: **procedure** CreateLayerOpportunistic(l_{id}) ▷ Marks layer l_{id} for creation, and initializes bitmap marking layer's sampled values; α is the sampling rate.
2: Mark layer l_{id} for creation.
3: LayerSize ← InputSize/2α
4: **for** l_{idx} in (0, LayerSize − 1) **do**
5: `IsLayerValueSampled`[l_{id}][l_{idx}] ← `0`

6: **procedure** OpportunisticPopulate(`val, idx`) ▷ Exploit query execution to populate layers opportunistically; `val` is the unsampled values computed during query execution, and `idx` is its index into the unsampled array.
7: l_{id} ← `GetLayerID(idx)` ▷ Get layer ID.
8: **if** layer l_{id} is marked for creation **then**
9: l_{idx} ← `GetLayerIdx(idx)` ▷ Get index into layer.
10: `SampledArray`[l_{id}][l_{idx}] ← val
11: `IsLayerValueSampled`[l_{id}][l_{idx}] ← 1

The second mode for layer creation in BlowFish is *opportunistic layer construction*. This mode exploits the fact that the unsampled values for the two arrays are computed on the fly during query execution. A subset of the these values are the ones to be computed for populating the new layer. Hence, the query execution phase can be used to populate the new layer without using dedicated threads. The challenge in this mode is when to update the `ExistsLayer` flag — if set during the layer creation, the queries may incorrectly access values that have not yet been populated; on the other hand, the layer may remain unused if the flag is set after all the values are populated. BlowFish handles this situation by using a bitmap that stores a bit per sampled value for that layer. A set bit indicates that the value has already been populated and vice versa. The algorithm for opportunistic layer construction is outlined in Algorithm 2.

It turns out that opportunistic layer construction performs very well for real-world workloads that typically follow a zipf-like distribution (repeated queries on certain objects). Indeed, the required unsampled values are computed during the first execution of a query and are thus available for all subsequent executions of the same query. Interestingly, this is akin to caching the query results without any explicit query result caching implementation.

Layer Deletion. Deleting layers is straightforward in BlowFish. To maintain consistency with layer additions, layer deletion proceeds from the most recently added layer. Layer deletions are computationally inexpensive, and do not require any special strategy. Upon the request for layer deletion, the `ExistsLayer` bitmap is updated to indicate that the corresponding layer is no longer available. Subsequent queries, thus, stop accessing the deleted layer. In order to maintain safety, we delay the memory deallocation for a short period of time after updating the `ExistsLayer` flag.

3.4.2 BlowFish Servers

We now provide details on the design and implementation of BlowFish servers.

Server Components

Each BlowFish server has three main components (see Figure 3.1 and Figure 3.2):

Data shards. Each server stores multiple data shards (employing the same partitioning strategy as Succinct), typically one per CPU core. The shards are replicated for fault-tolerance and load-balancing (§3.4.2). Each shard stores the two sampled arrays — Input2AoS and AoS2Input — using LSA, along with other data structures in Succinct. This enables a smooth storage-performance tradeoff, as described in §3.4.1. The aggregate storage overhead of the shards may be larger than available main memory. Each shard is memory mapped; thus, only the most accessed shards may be paged into main memory.

Request Queues. BlowFish servers maintain a queue of outstanding queries per shard, referred to as *request queues*. The length of request queues provide a rough approximation to the load on the shard — larger request queue lengths indicate a larger number of outstanding requests for the shard, implying that the shard is observing more queries than it is able to serve (and vice versa).

Server Handler. Each server in BlowFish has a server handler module that acts as an interface to clients as well as other server handlers in the system. Each client connects to one of the server handlers that handles the client query (similar to Cassandra [102]). The server handler interacts with other server handlers to execute queries and to maintain the necessary system state. BlowFish server handlers are also responsible for query scheduling and load balancing, and for making decisions on how shards share the cache available at the *local* server. We discuss these functionalities below.

Query execution

Similar to existing data stores [102, 130, 57], an incoming query in BlowFish may touch one or more shards depending on the sharding scheme. The server handler handling the query is responsible for forwarding the query to the server handler(s) of the corresponding shard(s); we discuss query scheduling across shard replicas below. Whenever possible, the query results from multiple shards on the same server are aggregated by the server handler.

Random access and search. BlowFish does *not* require changes in Succinct algorithms for executing queries at each shard, with the exception of looking up values in sampled arrays. In particular, since the two sampled arrays in Succinct — Input2AoS and AoS2Input — are replaced by LSA, the corresponding lookup algorithms are replaced by lookup algorithms for LSA (§3.3.3, Figure 3.3). We note that, by using `ExistsLayer` flag, BlowFish makes LSA lookup algorithms transparent to existing layers and query execution.

Updates. BlowFish implements data appends exactly as Succinct does. Specifically, BlowFish uses a multi-store architecture with a write-optimized LogStore that supports fine-grained appends, a query-optimized SuffixStore that supports bulk appends and a memory-optimized SuccinctStore. LogStore and SuffixStore, for typical cluster configurations, store less than 0.1% of the entire dataset (the most recently added data). BlowFish does not require changes in LogStore and SuffixStore implementation, and enables the storage- performance tradeoff for data only in SuccinctStore. Since the storage and the performance of the system is dominated by SuccinctStore, the storage-performance tradeoff curve of BlowFish is not impacted by update operations for most workloads.

Scheduling and Load Balancing

BlowFish server handlers maintain the request queue lengths for each shard in the system. Each server handler periodically monitors and records the request queue lengths for *local* shards. For non-local shards, the request queue lengths are collected during the query phase — server handlers encapsulate the request queue lengths for their local shards in the query responses. Upon receiving a query response, a server handler unpacks the request queue lengths and updates its local metadata to record the new lengths for the corresponding shards.

Each shard (and shard replica) in BlowFish may operate on a different point on the storage-performance curve (Figure 3.2). Thus, different replicas of the same shard may have different query execution time for the same query. To efficiently schedule queries across such a heterogeneous system, BlowFish adopts techniques from scheduling theory literature — a back-pressure scheduling style Join-the-shortest-queue [83] mechanism. An incoming query for a shard is forwarded to the replica with the smallest request queue length. By conceptually modeling this problem as replicas having the same speed but varying job sizes (for the same query), the analysis for Join-the-shortest-queue [83] applies to BlowFish, implying close to optimal load balancing.

Dynamically Navigating the Tradeoff

BlowFish uses the request queues not only for scheduling and load balancing, but also to trigger navigation along the storage-performance tradeoff curve for each individual shard. We discuss below the details on tradeoff navigation, and how this enables efficient cache sharing among shards within and across servers.

One challenge in using request queue lengths as an approximation to load on the shard is to differentiate short-term spikes from persistent overloading of shards (Figure 3.4). To achieve this, BlowFish server handlers also maintain exponentially averaged queue lengths for each local shard — the queue lengths are monitored every δ time units, and the exponentially averaged queue length at time t is computed as:

$$Q_t^{avg} = \beta \times Q_t + (1 - \beta) \times Q_{t-\delta}^{avg} \tag{3.1}$$

The parameters β and δ provide two knobs for approximating the load on a shard based on its request queue length. β is a fraction ($\beta < 1$) that determines the contribution of more recent queue length values to the average — larger β assigns higher weight to more recent values in the average. δ is the periodicity at which queue lengths are averaged — smaller values of δ (i.e., more frequent averaging) results in higher sensitivity to bursts in queue length. Note that a small exponentially average queue length implies a persistently underloaded shard.

We now describe how shards share the available cache within and across servers by dynamically navigating along the storage-performance tradeoff curve. We start with the relatively simpler case of shards on the same server, and then describe the case of shards across servers.

Shards on the same server. Recall that BlowFish implementation adds and deletes layers in a bottom-up fashion, with each layer storing sampled values for powers of two. Thus, at any instant, the sampling rate of LSA is a power of two $(2, 4, 8, \dots)$. For each of these sampling rates, BlowFish stores two *threshold* values. The *upper threshold* value is used to trigger storage increase for any particular shard — when the exponentially averaged queue length of a shard S crosses the upper threshold value, S must be consistently overloaded and must increase its throughput.

However, the server may not have extra cache to sustain the increased storage for S. For such scenarios, BlowFish stores a *lower threshold* value which is used to trigger storage reduction. In particular, if the exponentially averaged queue length *and* the instantaneous request queue length for one of the other shards S' on the same server is below the lower threshold, BlowFish reduces the storage for S' before triggering the storage increase for S. If there is no such S', the server must already be throughput bottlenecked and the navigation for S is not triggered.

We make two observations. First, the goals of exponentially averaged queue lengths and two threshold values are rather different: the former makes BlowFish stable against temporary spikes in load, while the latter against "flap damping" of load on the shards. Second, under stable loads, the above technique for triggering navigation along the tradeoff curve allows each shard on the same server to share cache proportional to its throughput requirements.

Shard replicas across servers. At the outset, it may seem like shards (and shard replicas) across servers need to coordinate among themselves to efficiently share the total system cache. It turns out that local cache sharing, as described above, combined with BlowFish's scheduling technique implicitly provides such a coordination.

Consider a shard S with two replicas R1 and R2, both operating at the same point on the tradeoff curve and having equal queue lengths. The incoming queries are thus equally distributed across R1 and R2. If the load on S increases gradually, both R1 and R2 will eventually experience load higher than the throughput they can support. At this point, the request queue lengths at R1 and R2 start building up at the same rate. Suppose R2 shares the server with other heavily loaded shards (that is, R2 can not navigate up the tradeoff

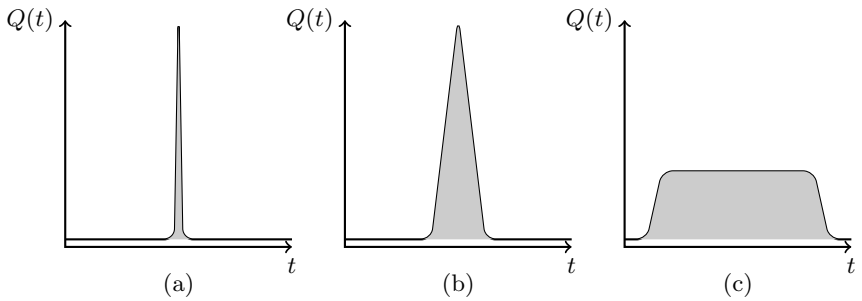


Figure 3.4: Three different scenarios of queue length ($Q(t)$) variation with time (t). (a) shows a very short-lasting "spike", (b) shows a longer lasting spike while (c) shows a persistent "plateau" in queue-length values. BlowFish should ideally ignore spikes as in (a) and attempt to adapt to the queue length variations depicted in (b) and (c).

curve). BlowFish will then trigger a layer creation for R1 only. R1 can thus support higher throughput and its request queue length will decrease. BlowFish's scheduling technique kicks in here: incoming queries will now be routed to R1 rather than equal load balancing, resulting in lower load at R2. It is easy to see that at this point, BlowFish will load balance queries to R1 and R2 proportional to their respective throughputs.

3.5 Evaluation

BlowFish is implemented in ≈ 2K lines of C++ on top of Succinct. We apply BlowFish to application domains outlined in §3.1 and compare its performance against state-of-the-art schemes for each application domain.

Evaluation Setup. We describe the setup used for each application in respective subsections. We describe here what is consistent across all the applications: dataset and query workload. We use the `LINEITEM` table from the TPC-H benchmark dataset [191], that consists of records with 8 byte primary keys and roughly 140 byte values on an average; the values comprise of 15 attributes (or columns). We note that several of our evaluation results are independent of the underlying dataset (*e.g.*, bandwidth for data repair, time taken to navigate along the tradeoff curve, etc.) and depend only on amount of data per server.

We use a query workload that comprises of 50% random access queries and 50% search queries; we discuss the impact of varying the fraction of random access and search queries in §3.5.1. Random access queries return the entire value, given a key. Search queries take in an (attribute, value) pair and return all keys whose entry for the input attribute matches the value. We use three query distributions in our evaluation for generating queries over the key space (for random access) and over the attribute values (for search). First, *uniform distribution* with queries distributed uniformly across key space and attribute values; this essentially constitutes a worst-case scenario for BlowFish[3]. The remaining two query work-

[3]Intuitively, queries distributed uniformly across shards and across records alleviates the need for shards

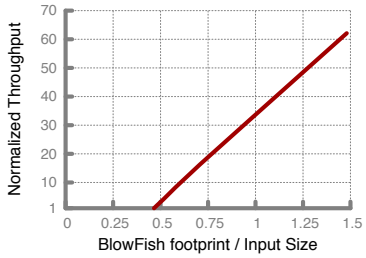


Figure 3.5: Storage-throughput tradeoff curve (per thread) enabled by BlowFish. The y-axis is normalized by the throughput of smallest possible storage footprint (71ops) in BlowFish.

loads follow *Zipf distribution with skewness* 0.99 *(low skew) and* 0.01 *(heavily skewed)*, the last one constituting the best-case scenario for BlowFish.

All our distributed experiments run on Amazon EC2 cluster comprising of c3.2xlarge servers, with 15GB RAM backed by two 80GB SSDs and 8 vCPUs. Unless mentioned otherwise, all our experiments shard the input data into 8GB shards and use one shard per CPU core.

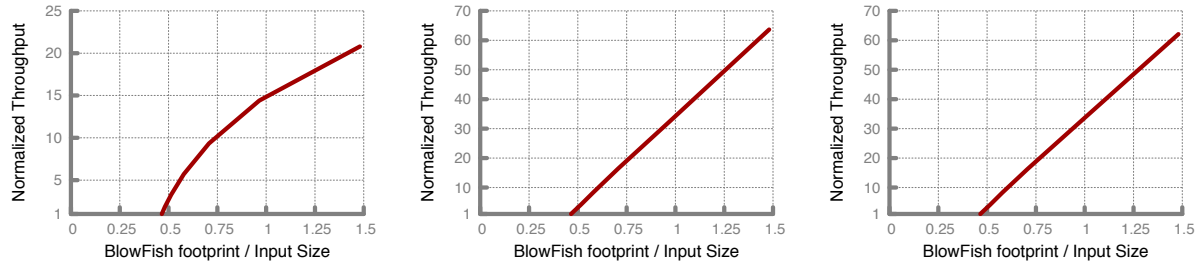
3.5.1 Storage Performance Tradeoff

We start by evaluating the storage-performance tradeoff curve enabled by BlowFish. Figure 3.5 shows this tradeoff for query workload comprising of 50% random access and 50% search queries. We make two observations; first, BlowFish achieves storage footprint varying from 0.5× to 8.7× the input data size (while supporting search functionality; the figure shows only up to 1.5× the data size for clarity)[4]. In particular, BlowFish does not enforce compression. Second, increase in storage leads to super-linear increase in throughput (moving from ≈ 0.5 to ≈ 0.75 leads to 20× increase in throughput) due to non-linear computational cost of operating on compressed data (see Chapter 2).

Storage-throughput Tradeoff for other workloads Figure 3.5 shows the storage-throughput tradeoff enabled by BlowFish for query workload comprising of 50% random access and 50% search queries. Figure 3.6 shows this tradeoff for other workloads. In particular, Figure 3.6a and Figure 3.6b show the storage-throughput tradeoff for workloads comprising of 100% random access and 100% search queries, respectively. Note that the tradeoff for mixed workload has characteristics similar to 100% `search` workload since, similar to other systems, execution time for search is significantly higher than random access. The throughput of the system is, thus, dominated by latency of search queries.

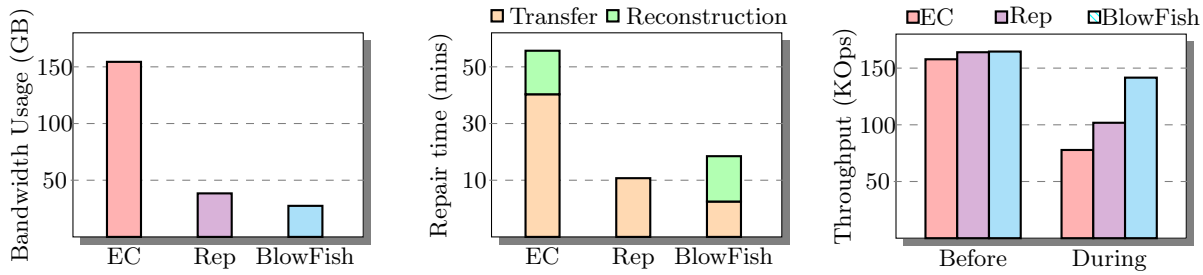
having varying storage footprints.

[4]The smallest footprint is 0.5× since TPC-H data is not very compressible, achieving compression factor of 3.1 using gzip.



(a) 100% random access (b) 100% search (c) 50% random access + 50% search

Figure 3.6: Storage-throughput tradeoff curve (per thread) enabled by BlowFish for workloads with varying fraction of `random access` and `search` queries. The y-axis is normalized by the throughput of smallest possible storage footprint in BlowFish (3874ops for random access only, 37ops for search only, and 71ops for the mixed workload).



(a) Bandwidth (b) Repair Time (c) Throughput

Figure 3.7: Comparison of BlowFish against RS erasure codes and replication (discussion in §3.5.2). BlowFish requires 5.4× lower bandwidth for data repair compared to erasure codes, leading to 2.5× faster repair time. BlowFish achieves throughput comparable to erasure codes and replication under no failures, and $1.4 - 1.8\times$ higher throughput during failures.

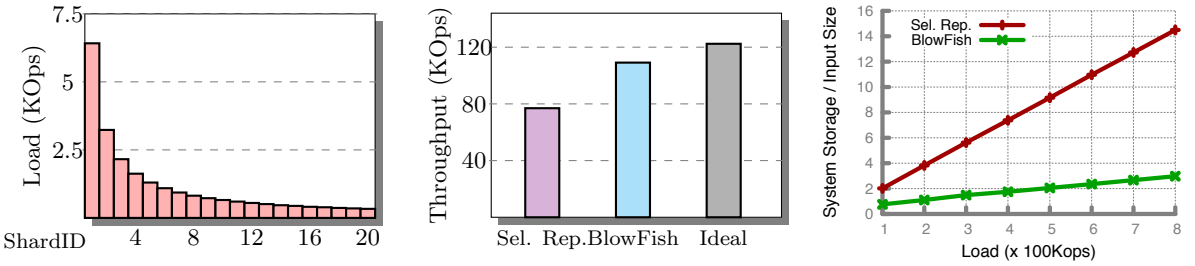
3.5.2 Data Repair During Failures

We now apply BlowFish to the first application: efficient data recovery upon failures.

Existing techniques and BlowFish tradeoffs. Two techniques exist for data repair during failures: replication and erasure codes. The main tradeoff is that of storage and bandwidth, as shown in Table 3.1. Note that this tradeoff is hard; that is, for both replication and erasure codes, the storage overhead and the bandwidth for data repair is fixed for a fixed fault tolerance. We discuss related work in §3.6, but note that erasure codes remain inefficient for data stores serving small objects due to high repair time and/or bandwidth requirements.

Experimental Setup

We perform evaluation along four metrics: storage overhead, bandwidth and time required for data repair, and throughput before and during failures. Since none of the open-source data stores support erasure codes, we use an implementation of Reed-Solomon (RS) codes [115]. The code use 10 data blocks and 2 parity blocks, similar to those used at Facebook [167, 133], but for two failure case. Accordingly, we use 3× replication. For BlowFish, we use an



(a) Load distribution across shards (b) Throughput for a fixed storage (c) Storage needed to sustain load

Figure 3.8: Comparison of BlowFish and selective caching for skewed workload application. See §3.5.3 for discussion.

instantiation that uses three replicas with storage $0.9\times, 0.5\times$ and $0.5\times$, aggregating to $1.9\times$ storage — an operating point between erasure codes and replication.

We use 12 server EC2 cluster to put data and parity blocks on separate servers; each server contains both data and parity blocks, but not for the same data. Replicas for replication and BlowFish were also distributed similarly. We use 160GB of total raw data distributed across 20 shards. The corresponding storage for erasure codes, replication and BlowFish is, thus, 192, 480 and 310GB. Note that the cluster has 180GB main memory. Thus, all data shards for erasure codes fit in memory, while a part of BlowFish and replication data is spilled to disk (modeling storage-constrained systems).

We use uniform query distribution (across shards and across records) for throughput results. Recall that this distribution constitutes a worst-case scenario for BlowFish. We measure the throughput for the mixed 50% random access and 50% search workload.

Results

Storage and Bandwidth. As discussed above, RS codes, replication and BlowFish have a storage overhead of $1.2\times$, $3\times$ and $1.9\times$. In terms of bandwidth, we note that the three schemes require storing $16, 40$ and 26GB of data per server, respectively. Figure 3.7a shows the corresponding bandwidth requirements for data repair for the three schemes. Note that while erasure codes require $10\times$ bandwidth compared to replication *for each individual failed shard*, the overall bandwidth requirements are less than $10\times$ since each server in erasure coded case also stores lesser data due to lower storage footprint of erasure codes (best case scenario for erasure codes along all metrics).

Repair time. The time taken to repair the failed data is a sum of two factors — time taken to copy the data required for recovery (transfer time), and computations required by the respective schemes to restore the failed data (reconstruction time). Figure 3.7b compares the data repair time for BlowFish against replication and RS codes.

RS codes require roughly $5\times$ higher transfer time compared to BlowFish. Although erasure codes read the required data in parallel from multiple servers, the access link at the server where the data is being collected becomes the network bottleneck. This is further

exacerbated since these servers are also serving queries. The decoding time of RS codes is similar to reconstruction time for BlowFish. Overall, BlowFish is roughly 2.5× faster than RS codes and 1.4× slower than replication in terms of time taken to restore system state after failures.

Throughput. The throughput results for the three schemes expose an interesting tradeoff (see Figure 3.7c).

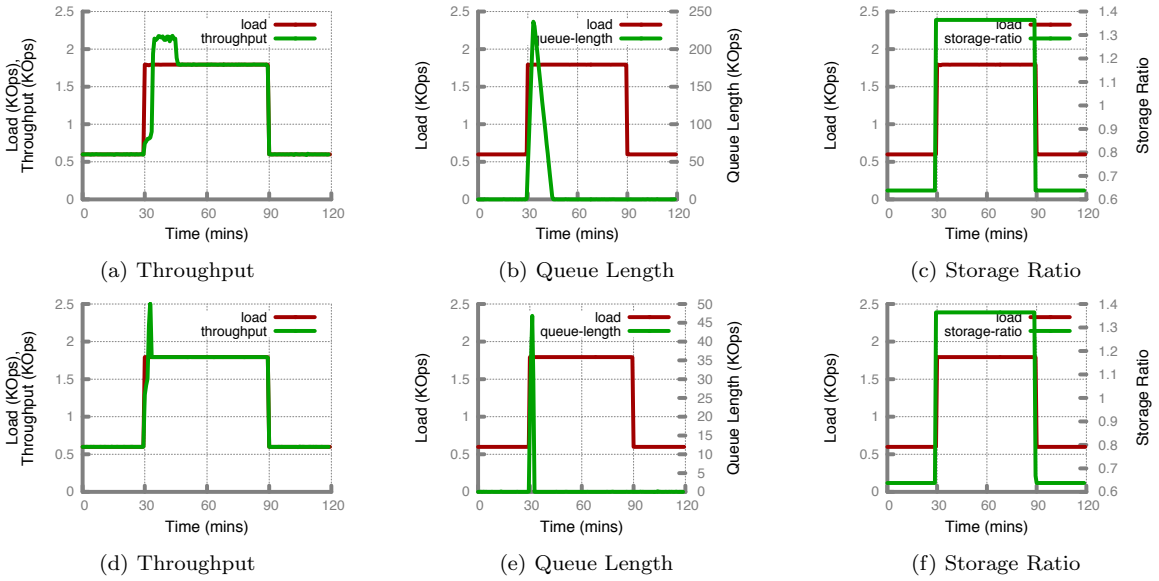
When there are no failures, all the three schemes achieve comparable throughput. This is rather non-intuitive since replication has three replicas to serve queries while erasure codes have only one and BlowFish has replicas operating at smaller storage footprints. However, recall that the cluster is bottlenecked by the capacity of faster storage. If we load balance the queries in replication and in BlowFish across the three replicas, many of these queries are executed off SSD, thus reducing the overall system throughput (much more for replication since many more queries are executed off SSD). To that end, we evaluated the case of replication and BlowFish where queries are load balanced to only one replica; in this case, as expected, all the three schemes achieve comparable throughput.

During failures, the throughput for both erasure codes and replication reduces significantly. For RS codes, 10 out of (remaining) 11 servers are used to both read the data required for recovery as well as to serve queries. This severely affects the overall RS throughput (reducing it by 2×). For replication, note that the amount of failed data is 40GB (five shards). Recovering these shards results in replication creating two kinds of interference: interfering with queries being answered on data unaffected by failures *and* queries answered on failed server now being answered off-SSD from remaining servers. This interference reduces the replication throughput by almost 33%. Note that both these interferences are minimal in BlowFish: fewer shards need be constructed, thus fewer servers are interfered with, and fewer queries go to SSD. It turns out that the interference is minimal, and BlowFish observes minimal throughput reduction (less than 12%) during failures. As a result, BlowFish throughput during failures is is $1.4 - 1.8\times$ higher than the other two schemes.

3.5.3 Skewed Workloads

We now apply BlowFish to the problem of efficiently utilizing the system cache for workloads with skewed query distribution across shards (*e.g.*, more queries on hot data and fewer queries on warm data). The case of skew across shards varying with time is evaluated in next subsection.

State-of-the-art. The state-of-the-art technique for handling spatially-skewed workloads is selective caching [9] which caches, for each object, number of replicas proportional to the load on the object.



(a) Throughput (b) Queue Length (c) Storage Ratio

(d) Throughput (e) Queue Length (f) Storage Ratio

Figure 3.9: Opportunistic layer construction with spiked changes in load for uniform workload (top three) and skewed workload (bottom three). The figures show variation in throughput (left), request queue length (center) and storage footprint (right).

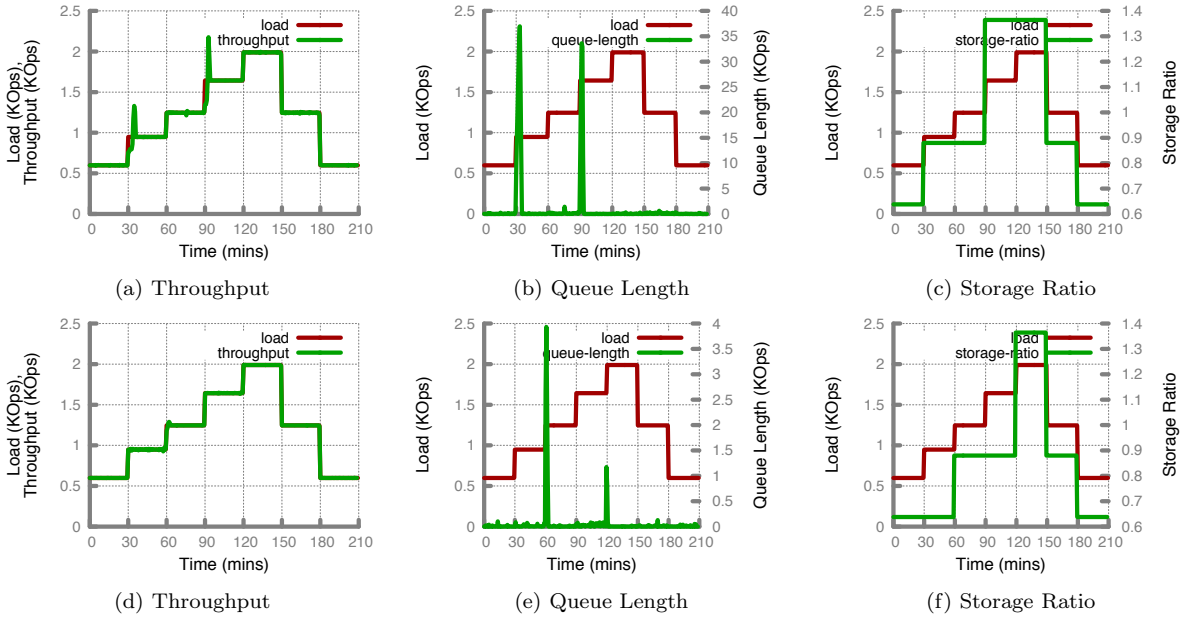
Experimental Setup

We use 20 data shards, each comprising of 8GB of raw data, for this experiment. We compare BlowFish and Selective caching using two approaches. In the first approach, we fix the cluster (amount of fast storage) and *measure* the maximum possible throughput that each scheme can sustain. In the second approach, we vary the load for the two schemes and *compute* the amount of fast storage required by each scheme to sustain that load.

For the former, we use a cluster with 8 EC2 servers. A large number of clients generate queries with a Zipf distribution with skewness 0.01 (heavily skewed) across the shards. As shown in Figure 3.8a, the load on the heaviest shard using this distribution is 20× the load on the lightest shard — this models the real-world scenario of a few shards being "hot" and most of the shards being "cold". For selective caching, each shard has number of replicas proportional to its load (recall, total storage is fixed); for BlowFish, the shard operates at a point on the tradeoff curve that can sustain the load with minimal storage overhead. We distribute the shards randomly across the available servers. For the latter, we vary the load and compute the amount of fast storage required by the two schemes to meet the load assuming that the entire data fits in fast storage. Here, we increase the number of shards to 100 to perform computations for a more realistic cluster size.

Results

For fixed storage. The storage required for selective caching and BlowFish to meet the load is 155.52GB and 118.96GB, respectively. Since storage is constrained, some shards



(a) Throughput (b) Queue Length (c) Storage Ratio

(d) Throughput (e) Queue Length (f) Storage Ratio

Figure 3.10: Opportunistic layer construction with gradual changes in load for uniform workload (top three) and skewed workload (bottom three). The figures show variation in throughput (left), request queue length (center) and storage footprint (right).

in selective caching can not serve queries from faster storage. Intuitively, this is because BlowFish provides a finer-grained tradeoff (increasing the storage overhead fractionally, just enough to meet the performance goals) compared to the coarse-grained tradeoff of selective replication (throughput can be increased only by 2× by adding another replica requiring 1× higher storage overhead). Thus, BlowFish utilizes the available system cache more efficiently. Figure 3.8b shows that this leads to BlowFish achieving 1.5× higher throughput than selective caching. Interestingly, BlowFish achieves 89% of the ideal throughput, where the ideal is computed by taking into account the load skew across shards, the total system storage, the maximum possible per-shard throughput per server, and by placing heavily loaded shards with lightly loaded shards. The remaining 11% is attributed to the random placement of shards across servers, resulting in some servers being throughput bottlenecked.

Fixed load. Figure 3.8c shows that, as expected, BlowFish requires 2.7 – 4.9× lower amount of fast storage compared to selective caching to sustain the load.

3.5.4 Time-varying workloads

We now evaluate BlowFish's ability to *adapt* to time-varying load, in terms of time taken to adapt and queue stability. We also evaluate the performance of BlowFish's scheduling technique during such time-varying loads.

Experimental Setup

We perform micro-benchmarks to focus on adaptation time, queue stability and per-thread shard throughput for time-varying workloads. We use a number of clients to generate time-varying load on the system. We performed four sets of experiments: uniform and skewed (Zipf with skewness 0.01) query distribution (across queried keys and search terms); and, gradual and spiked variations in load. Specifically perform the following micro-benchmarks:

- For the spiked variation in load, we increase the load on the shard from 600ops to 1800ops suddenly (3× increase in load models failures of two replicas, an extremely unlikely scenario) at time $t = 30$ and observe the system for an hour before dropping down the load back to 600ops at time $t = 90$.
- we increase the load from 600ops to 2000ops, with a gradual increase of 350ops at 30 minute intervals. This granularity of increase in load is similar to those reported in real-world production clusters [16], and constitutes a much easier case for BlowFish compared to the spiked increase in load.

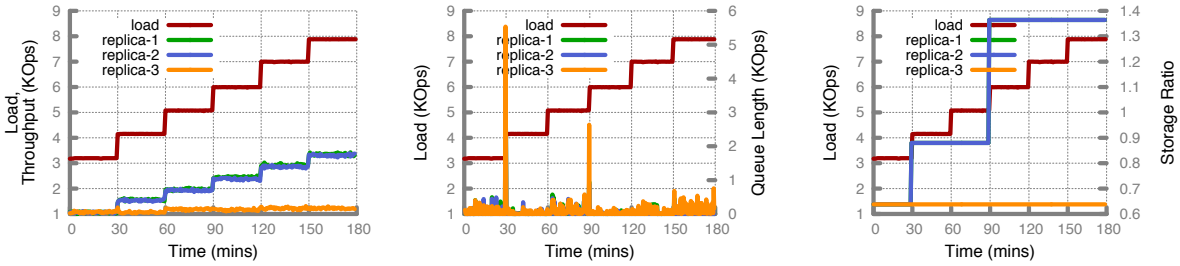
Results

BlowFish adaptation time and queue stability for spiked variation in load. As the load is increased from 600ops to 1800ops, the throughput supported by the shard at that storage ratio is insufficient to meet the increased load (Figures 3.9a and 3.9d). As a result, the request queue length for the shard increases (Figures 3.9b and 3.9e). At one point, BlowFish triggers *opportunistic layer creation* — the system immediately allocates additional storage for the two sampled arrays (increased storage ratio in Figures 3.9c and 3.9f); the sampled values are filled in gradually as queries are executed.

At this point, the results for uniform and skewed query distribution differ. For the uniform case, the already filled sampled values are reused infrequently. Thus, it takes BlowFish longer to adapt ($\approx$ 5 minutes) before it starts draining the request queue (the peak in Figure 3.9b). BlowFish is able to drain the entire request queue within 15 minutes, making the system stable at that point.

For the skewed workload, the sampled values computed during query execution are reused frequently since queries repeat frequently. Thus, BlowFish is able to adapt much faster ($\approx$ 2minutes) and drain the queues within 5 minutes. Note that this is akin to caching of results, explicitly implemented in many existing data stores [102, 130, 57] while BlowFish provides this functionality inherently.

Gradual workload variations. For the uniform query distribution (Figure 3.10, top), as the load increases from 600ops to 950ops (Figure 3.10a), the load becomes higher than the throughput supported by the shard at that storage ratio (800ops). Consequently, the request queue length starts building up (Figure 3.10b), and BlowFish triggers a layer addition by allocating space for the new layers (Figure 3.10c). BlowFish opportunistically fills up values



(a) Throughput (b) Queue Length (c) Storage ratio

Figure 3.11: **The effectiveness and stability of BlowFish's query scheduling mechanism** in a replicated system (discussion in §3.5.4). Variation in throughput (left), request queue lengths (center) and storage-footprints (right) for the three replicas.

in the new layer, and the throughput for the shard increases gradually. This continues until the throughput matches the load on the shard; at this point, however, the throughput continues to increase even beyond the load to deplete the outstanding requests in the queue until the queue length reduces to zero and the system resumes normal operation. A similar trend can be seen when the load is increased to 1650ops.

For the skewed query distribution (Figure 3.10, bottom), the trends observed are similar to those for the uniform workload, with two key differences. First, we observe that BlowFish triggers layer creation at different points for this workload. In particular, the throughput for the skewed workload at the same storage footprint (0.8 in Figure 3.10c and 3.10f) is higher than that for the uniform workload. To see why, note that the performance of `search` operations varies significantly based on the queries; while the different queries contribute equally for the uniform workload, the throughput for the skewed workload is shaped by the queries that occur more frequently. This effect attributes for the different throughput characteristics for the two workloads at the same storage footprint.

Second, as noted before (§3.5.4), BlowFish adaptation benefits from the repetitive nature of queries in the skewed workload, since repeated queries can reuse the values populated during their previous execution. In comparison to uniform query distribution, this leads to faster adaptation to increase in load and quicker depletion of the increased request queue lengths.

Note the difference in results for the case of spiked increase in load (Figure 3.9) and gradual increase in load (Figure 3.10). In the former case, the increase in load leads to significantly higher request queue lengths and hence, it takes much longer for the system to return to normal operations. In the latter, however, due to gradual increase in load, the system can drain the outstanding request queue significantly faster, can resume normal operations faster, and thus provides adaptation at much finer time granularity.

BlowFish scheduling. To evaluate the effectiveness and stability of BlowFish scheduling, we turn our attention to a distributed setting. We focus our attention on three replicas of the same shard. We make the server storing one of these replicas storage constrained (replica #3); that is, irrespective of the load, the replica cannot trigger navigation along the

storage-performance tradeoff curve. We then gradually increase the workload from 3KOps to 8KOps in steps of 1KOps per 30 minutes (Figure 3.11) and observe the behavior of request queues at the three replicas.

Initially, each of the three replicas observe a load of 1KOps since queue sizes are equal, and BlowFish scheduler equally balances the load. As the load is increased to 4KOps, the replicas are no longer able to match the load, causing the request queues at the replicas to build up (Figure 3.11b). Once the queue lengths cross the threshold, replica #1 and #2 trigger layer construction to match higher load (Figure 3.11c).

As the first two replicas opportunistically add layers, their throughput increases; however, the throughput for the third replicas remains consistent (Figure 3.11a). This causes the request queue to build up for the third replica at a rate higher than the other two replicas (Figure 3.11b). Interestingly, the BlowFish quickly adapts, and stops issuing queries to replica#3, causing its request queue length to start dropping. We observe a similar trend when the load increases to 5KOps. BlowFish does observe queue length oscillations during adaptation, albeit of extremely small magnitude.

3.6 Related Work

BlowFish's goals are related to three key areas:

Storage-performance tradeoff. Existing data stores usually support two extreme operating points for each cached shard — compressed but low throughput, and uncompressed but high throughput. Several compression techniques (*e.g.*, gzip) can allow achieving different compression factors by changing parameters. However, these require decompression and re-compression of the entire data on the shard. As shown in this chapter, a smooth and dynamic storage-performance tradeoff not only provides benefits for existing applications but can also enable a wide range of new applications.

Data repair. The tradeoff between known techniques for data repair — replication and erasure codes — is that of storage overhead and bandwidth. Studies have shown that the bandwidth requirement of traditional erasure codes is simply too high to use them in practice [167]. Several research proposals [88, 152, 167] reduce the bandwidth requirements of traditional erasure codes for batch processing jobs. However, these codes remain inefficient for data stores serving small objects. As shown in §3.5, BlowFish achieves storage close to erasure codes, while maintaining the bandwidth and repair time advantages of replication.

Selective Replication. As discussed in §3.5, selective replication can achieve good performance for workloads skewed towards a few popular objects. However, most approaches [9] only provide a coarse-grained support — increasing the throughput by 2× by increasing the storage overhead by 1×. BlowFish, instead, provides a much finer-grained control allowing applications to increase the storage fractionally, just enough to meet the performance goals.

In contrast, approaches that replicate data at finer-granularities (*e.g.*, per-record or per-object replication [202, 198]) to address these issues, need to maintain complex fine-grained

metadata (i.e., at the granularity of replication), which can add both performance (*e.g.*, under rapid changes in query workloads) as well as storage overheads (*e.g.*, for datasets with many small records).

3.7 Summary

BlowFish is a distributed data store that enables a smooth storage-performance tradeoff between two extremes — compressed but low throughput and uncompressed but high throughput. In addition, BlowFish allows applications to navigate along this tradeoff curve over fine-grained time scales. Using this flexibility, we explored several problems from real-world production clusters from a new "lens" and showed that the tradeoff exposed by BlowFish can offer significant benefits compared to state-of-the-art techniques for the respective problems.

Chapter 4

Interactive Queries on Compressed Graphs

While Succinct enables a wide range of queries directly on compressed data, emerging cloud applications demand even more in terms of both the structure of data to be queried, as well of the kinds of queries that they need to support. In keeping with these demands, this chapter, and the next, explores the problem of exploring richer data models and query semantics on Succinct. More specifically, we ask the following question: How can we encode different types of data and support different types of queries, so as to leverage Succinct's (and BlowFish's) unique advantages outlined in Chapters 2 and 3?

This chapter, in particular, focuses on graph data, and the queries that cloud applications perform on them. Large graphs are becoming increasingly prevalent across a wide range of applications including social networks, biological networks, knowledge graphs and cryptocurrency. Many of these applications store, in addition to the graph structure (nodes and edges), a set of *attributes* or *properties* associated with each node and edge in the graph [30, 15, 149, 136, 189]. Many recent industrial studies [15, 30, 137] report that the overall size of these graphs (including both the structure and the properties) could easily lead to terabytes or even petabytes of graph data. Consequently, it is becoming increasingly hard to fit the entire graph data into the memory of a single server [30, 171, 136].

How does one operate on graph data distributed between memory and secondary storage, potentially across multiple servers? This question has attracted a lot of attention in recent years for offline graph analytics, *e.g.*, recent systems like GraphLab [116], GraphX [79], GraphChi [100] and Trinity [171]. These systems now enable efficient "batch processing" of graph data for applications that often run at the scale of minutes.

Achieving high performance for interactive user-facing queries on large graphs, however, remains a challenging problem. For such interactive queries, the goal is not only to achieve millisecond-level latency, but also high throughput [30, 15, 56]. When the graph data is distributed between memory and secondary storage, potentially across multiple servers, achieving these goals is particularly hard.

For instance, consider the query: "Find friends of Alice who live in Berkeley". One

possible way to execute this query is to execute two sub-queries — find friends of Alice, and, find people who live in Berkeley — and compute the final result using an intersection of results from two sub-queries. Such joins may be complex[1], and may incur high computational and bandwidth[2] overheads [53].

Another interesting possibility to execute the above query is to first find friends of Alice, and then for each friend, check whether or not the friend lives in Berkeley. Executing the query in this manner alleviates the overheads of the first approach, but requires *random access* into the "location" property of each friend of Alice. The well-known problem here is that typical graph queries exhibit *little or no locality* — the query may touch arbitrary parts of the graph, potentially across multiple servers, some of which may be in memory and some of which may be on secondary storage. Unless the data for each of Alice's friends is stored in memory, the query latency and system throughput suffers. Thus, intuitively, to achieve high system performance, graph stores should store as large a fraction of graph data as possible in memory.

One way to store a larger fraction of graph data in memory is to use compression. However, traditional block compression techniques (*e.g.*, gzip) are inefficient for graph queries precisely due to lack of locality — since queries may touch arbitrary parts of the graph, each query may require decompressing a large number of blocks (*e.g.*, all blocks that contain Alice's friends in the above example). Thus, designing compression techniques specialized to graphs has been an active area of research for the better part of last two decades [24, 27, 38, 66, 85, 123, 84, 173, 118]. Many of these techniques even support executing queries on compressed graphs [27, 38, 66, 85, 123, 84, 118]. However, existing techniques ignore node and edge properties and are limited to a small subset of queries on graph structure (*e.g.*, extracting edges incident on a node, or subgraph matching). Contemporary applications require executing far more complex queries [30, 91, 31, 53], often involving node and edge properties.

We present ZipG — a memory-efficient, distributed graph store for efficiently serving interactive graph queries. ZipG achieves memory efficiency by storing the input graph data (nodes, edges and the associated properties) using Succinct's compressed representation, and consequently stores a larger fraction of graph data in memory when compared to existing graph stores. What differentiates ZipG from existing graph stores is its ability to execute a *wide range of queries* directly on this compressed representation — ZipG exposes a minimal API that is rich enough to implement functionalities from several graph stores including those from Facebook [30], LinkedIn [200] and Twitter [90]. We demonstrate this by implementing and evaluating the published graph queries from Facebook TAO, LinkBench, Graph Search and several other workloads on top of ZipG. Using a single server with 244GB memory, ZipG executes tens of thousands of TAO, LinkBench and graph search queries for raw graph data over half a Terabyte.

[1]Graph search queries (such as the ones that we evaluate later in §4.4) when implemented using naive join algorithms are referred to as "Join Bombs" by one of the state-of-the-art graph serving companies [53].

[2]The cardinality of results for the two sub-queries may be orders of magnitude larger than the final result cardinality.

Table 4.1: **ZipG's API** and an example for each API. See §4.1.2 for definitions and detailed discussion.

API	Example
`g = compress(graph)`	Compress **graph**.
`List<String> g.get_node_property(nodeID, propertyIDs)`	Get **Alice's age** and **location**.
`List<NodeID> g.get_node_ids(propertyList)`	Find people in **Berkeley** who like **Music**.
`List<NodeID> g.get_neighbor_ids(nodeID, edgeType, propertyList)`	Find **Alice's friends** who live in **Boston**.
`EdgeRecord g.get_edge_record(nodeID, edgeType)`	Get all information on **Alice's friends**.
`Pair<TimeOrder> g.get_edge_range(edgeRecord, tLo, tHi)`	§4.1.2; `tLo` and `tHi` are `timestamps`.
`EdgeData g.get_edge_data(edgeRecord, timeOrder)`	Find **Alice's most recent friend**.
`g.append(nodeID, PropertyList)`	Append new node for **Alice**.
`g.append(nodeID, edgeType, edgeRecord)`	Append new edges for **Alice**.
`g.delete(nodeID)`	Delete **Alice** from the graph.
`g.delete(nodeID, edgeType, destinationID)`	Delete **Bob** from **Alice's** friends list.

While ZipG builds upon Succinct, it must resolve a number of challenges regarding the graph layout. Specifically, it must not only transform the graph data into a format that Succinct understands, but ensure that the format still preserves the benefits of Succinct for graph queries, i.e., it should still be possible to perform graph queries directly on the compressed graph representation. To this end, ZipG uses a new simple and intuitive graph layout that transforms the input graph data into a flat unstructured file (§4.2). This layout admits memory-efficient representation using Succinct. In addition, this layout carefully stores small amount of metadata along with the original input graph data in a manner that the two primitives of Succinct (random access and substring search) can be extended to efficiently implement interactive graph queries as expressive as those in Facebook TAO, LinkBench and Graph Search workloads directly on compressed representation of ZipG.

There are two additional challenges associated with storing data using a compressed representation. The first challenge is to support high write rates. The traditional approach to resolving this challenge is to use a log-structured approach — updates are appended into a log; these logs are periodically compressed into an immutable representation, after which the new updates are written into a new log. However, naïvely using a log-structured approach in graph stores results in nodes and edges having their data "fragmented" across multiple logs; without any additional data structures, each query now requires touching all the logs resulting in reduced throughput (§4.2.5). ZipG uses the log-structured approach, but avoids touching all logs using the idea of fanned updates. Specifically, each server in ZipG stores a set of update pointers that ensure that during query execution, ZipG touches exactly those logs (and bytes within the logs) that are necessary for query execution. The second challenge with compressed representation is in providing strong consistency guarantees and transactions. ZipG currently does not attempt to resolve this challenge. While several graph stores used in production [30, 200, 90] make a similar design choice, extending ZipG to provide such guarantees is an interesting future direction.

We evaluate ZipG against Neo4j [136] and Titan [189], two popular open-source graph

stores. All our experiments run on a set of commodity Amazon EC2 machines, and use five workloads — Facebook TAO [30], LinkBench [15], Graph Search [91], Regular Path Queries [17] and simple graph traversals. We use graphs from the real-world (annotated with node and edge properties using TAO distribution) as well as LinkBench generated graphs containing millions of nodes and billions of edges. Our evaluation shows that ZipG significantly outperforms Neo4j and Titan in terms of system throughput, usually by an order of magnitude but sometimes by as much as 23×.

4.1 Data model and Interface

We start by outlining ZipG graph data model (§4.1.1) and the interface exposed to the applications (§4.1.2).

4.1.1 ZipG Data Model

ZipG uses the property graph model [149, 136, 189, 30], with graph data comprising of nodes, edges, and their associated properties.

Nodes and Edges. ZipG employs the usual definitions of nodes and edges. Edges in ZipG could be directed or undirected. To model applications where graphs may have different "types" of edges (*e.g.*, comments, likes, relationships) [30], ZipG represents each edge using a 3-tuple comprising of `sourceID`, `destinationID` and an `EdgeType`, where the latter identifies the type of the edge. Each edge may potentially have a different EdgeType and may optionally have a `Timestamp`. The precise representation is described in §4.2.

Node and Edge Properties. Each node and edge in ZipG may have multiple properties, represented by `PropertyList`. Each PropertyList is a collection of (PropertyID, PropertyValue) pairs; *e.g.*, the PropertyList for a node may be {(age, 20), (location, Berkeley), (zipcode, 14853)}. Each PropertyList in ZipG may have arbitrarily many properties.

4.1.2 ZipG Interface

ZipG exposes a minimal, yet functionally rich, interface that abstracts away the internal graph data representation details (*e.g.*, compression). Applications interact with ZipG as if they were working on original graph data. In this section, we outline this interface. We start with some definitions:

- `EdgeRecord`: An EdgeRecord holds a reference to all the edges of a particular EdgeType incident on a node and to the data corresponding to these edges (timestamps, destinationID, PropertyList, etc.).
- `TimeOrder`: EdgeRecord can be used to efficiently implement queries on edges. Many queries, however, also require a notion of time (*e.g.*, find all comments since last login).

To efficiently execute such queries, ZipG uses TimeOrder — for each node, the incident edges of the same type are logically sorted using timestamps. TimeOrder of an edge represents the order (*e.g.* i-th) of the edge within this sorted list.

- `EdgeData`: Given the TimeOrder within an EdgeRecord, the EdgeData stores the triplet (destinationID, timestamp, PropertyList) for the corresponding edge.

Table 4.1 outlines the interface exposed by ZipG, along with some examples. Applications submit the `graph`, represented using the property model in §4.1.1, to ZipG and generate a memory-efficient representation using `compress(graph)`. Applications can then invoke powerful primitives (Table 4.1) as if the input graph was stored in an uncompressed manner; ZipG internally executes queries efficiently directly on the compressed representation. Most queries in Table 4.1 are self-explanatory; we discuss some of the interesting aspects below, and return to details on query implementation in §4.3.2.

Wildcards. ZipG queries admit *wildcard* as an argument for `PropertyID, edgeType, tLo, tHi` and `timeOrder`. ZipG interprets wildcards as admitting any possible value. For instance, `get_node_property(nodeID,*)` returns all properties for the node, and `get_edge_record(nodeID,*)` returns all edgeRecords for the node (and not just of a particular edgeType).

Node-based queries. Consider again the query "Find friends of Alice who live in Berkeley". Letting Alice to be the NodeID and assuming friends have edgeType `0`, the query `get_neighbor_ids(Alice,0,{Location, Berkeley})` returns the desired results. Internally, ZipG implements this query by first finding Alice's friends, and then checking for each of the friends, whether or not the friend lives in Berkeley. Applications that have knowledge about the structure of the graph and/or queries can also execute the same query using a different approach — using `get_neighbor_ids(Alice,0,*)` $\cap$ `g.get_node_ids(Location, Berkeley)`, where the former returns all friends of Alice and the latter returns all people who live in Berkeley. We compare the performance of executing queries these two approaches in §4.4.6.

Edge-based queries and Updates. ZipG allows applications to get random access to any EdgeRecord using `get_edge_record`, and, into the data for any specific edge in EdgeRecord using `get_edge_data`. If edges contain timestamps, ZipG also allows applications to access edges based on timestamps using `get_edge_range`. Finally, applications can insert EdgeRecords using `append`, delete existing EdgeRecords using `delete`, and update an EdgeRecord using a `delete` followed by an `append`.

4.2 ZipG Design

In this section, we present ZipG's design and implementation.

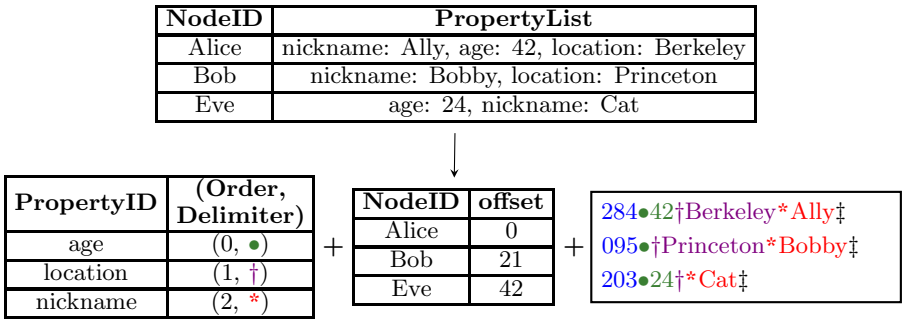


Figure 4.1: An example for describing the layout of NodeFile. See description in §4.2.2.

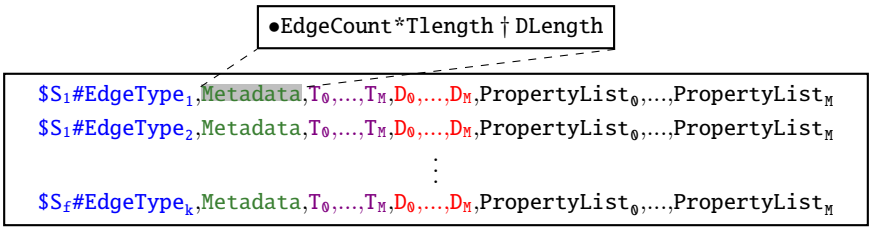


Figure 4.2: **EdgeFile Layout in ZipG (§4.2.2).** Each row is an EdgeRecord for a `(sourceID, edgeType)` pair. Each EdgeRecord contains, from left to right, metadata such as edge count and width of different edge data fields, sorted timestamps, destination IDs, and edge PropertyLists.

4.2.1 ZipG overview

ZipG builds on top of Succinct, and at a high level, employs the following strategy:

- Graph data is first converted into "flat file" representations so that they can be compressed using Succinct. This allows ZipG to exploit Succinct's search and random access operations.
- Graph queries are supported directly on the compressed graph representation using a combination of random access and search.

However, ZipG has to resolve a number of challenges to achieve the desired expressivity, scalability and performance. We outline some of these challenges below, and provide a brief overview of how ZipG resolves these challenges.

Storing graphs. One of the fundamental challenges that ZipG has to resolve is to design an efficient layout for storing graph data using the underlying data store. This is akin to GraphChi [100] and Ligra [172], that design new layouts for using the underlying storage system for efficient batch processing of graphs. ZipG's layout should not only admit a memory-efficient representation when using Succinct, but should also enable efficient implementation of interactive graph queries using the random access and search primitives of Succinct.

ZipG's graph layout uses two flat unstructured files:

- **NodeFile** stores all NodeIDs and corresponding properties. ZipG NodeFile adds a small amount of metadata to the list of (NodeID, nodeProperties) before invoking compression; this allows ZipG to tradeoff storage (in uncompressed representation) for efficient random access into node properties.
- **EdgeFile** stores all the EdgeRecords. By adding metadata and by converting variable length data into fixed length data before invoking compression, ZipG EdgeFile trades off storage (in uncompressed representation) to optimize random access into EdgeRecords and more complex operations like binary search over timestamps.

We discuss design of ZipG NodeFile and EdgeFile, and associated tradeoffs in §4.2.2.

Updating graphs. Another challenge for ZipG is to support high write rates over compressed graphs. As described earlier, while Succinct (§2) resolves this using log-structured storage [4, 95], naïvely adapting this approach to graph stores results fragmentation of node and edge data across multiple logs. This increases overheads for queries, which would now need to touch all the data. ZipG resolves this problem using the idea of fanned updates. At a high-level, ZipG servers stores *update pointers* that ensure that during query execution, ZipG touches only the logs necessary for query execution. We discuss fanned updates in §4.2.5 in greater detail.

4.2.2 Graph Representation

Existing graph stores use layouts that expose a hard tradeoff between flexibility and scalability. On the one hand, systems like Neo4j [136] heavily use pointers to store both the structure of the graph and the properties for nodes and edges. While flexible in representation, a pointer-based approach suffers from scalability issues when the entire graph data does not fit into the memory of a single server[3]. Systems like Titan [189], on the other hand, scale well by using a layout that can be mapped to a key-value (KV) store abstraction. However, KV abstraction is not very well suited for interactive graph queries [30] — by enforcing values to be stored as a "single opaque object", KV abstraction limits the flexibility of graph stores. Specifically, storing all the node properties (or, set of incident edges) as a single opaque object precludes these systems from fine-grained access into individual node properties (or, individual edges).

ZipG uses a new graph layout that, while simple and intuitive, provides both the scalability and flexibility by operating on flat unstructured files. ZipG uses two flat unstructured files, which we describe next.

[3]Pointer chasing during query execution requires multiple accesses to secondary storage and/or different servers, leading to undesired bottlenecks.

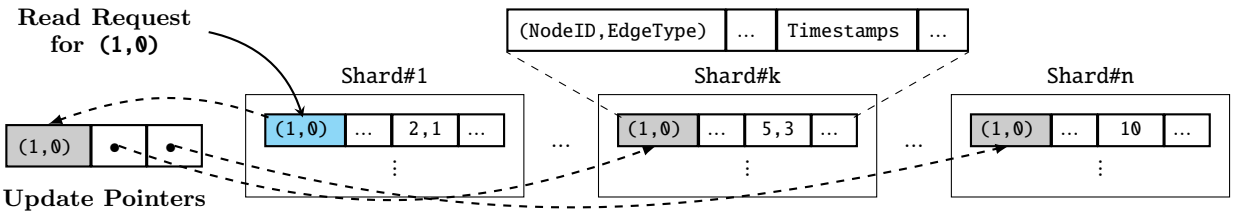


Figure 4.3: **Update Pointers for the EdgeFile (§4.2.5).**

NodeFile

NodeFile stores all the NodeIDs and associated properties, and is optimized for two kind of queries on nodes: (1) given a (NodeID, List<propertyID>) pair, extract the corresponding propertyValues; and (2) given a PropertyList, find all NodeIDs whose properties match the propertyList.

NodeFile consists of three data structures (see Figure 4.1). First, each propertyID in the graph is assigned a unique delimiter[4] and stored as a **PropertyID → (order, delimiter) map**, where `order` is lexicographic ranking of the propertyID among all propertyIDs.

The second data structure is a flat unstructured file that stores PropertyLists along with some metadata as described next. The propertyValues are prepended by their propertyID's delimiter and then written in the flat file in sorted order of propertyIDs; if a propertyID has a null propertyValue, we simply write down the delimiter. An end-of-record delimiter is appended to the end of the serialized propertyList of each node. For instance, Alice's propertyList in Figure 4.1 is serialized into •42†Berkeley*Ally‡, where ‡ is the end-of-record delimiter.

The metadata in the second data structure exposes a space-latency tradeoff. Specifically, the size of propertyValues within a node's propertyList vary significantly in real-world datasets (*e.g.*, Alice's age 42 and location Berkeley in above example) [30]. Using the largest size of PropertyValues (6bytes for Alice) as a fixed size representation for each propertyValue enables efficient random access but at the cost of space inefficiency. On the other hand, naïvely using a space-efficient variable size representation (2bytes for age, 8bytes for location, etc.) without any additional information leads to inefficient random access — Alice may put her age, name, nickname, location, status, workplace, etc. and accessing status may require extracting many more bytes than necessary. To that end, ZipG uses variable size representation for propertyValues but also explicitly stores the length of each propertyValue into the metadata for each propertyList. The lengths of propertyValues are encoded using a global fixed size `len`, since they tend to be short and of nearly similar size. In the example of Figure 4.1, the propertyList for Alice is thus encoded as 284•42†Berkeley*Ally‡.

[4]Graphs usually have a small number of propertyIDs across all nodes and edges; ZipG uses one byte non-printable characters as delimiters (for up to 25 propertyIDs) and two byte delimiters (for up to 625 propertyIDs).

The third data structure stored in NodeFile is a simple two-dimensional table that stores a sorted list of NodeIDs and the offset of node's PropertyList in NodeFile.

EdgeFile

EdgeFile stores the set of edges and their properties. Recall from §4.1 that each edge is uniquely identified by the 3-tuple `(sourceNodeID,destinationNodeID,EdgeType)` and may have an associated timestamp and a list of properties. See Figure 4.2 for an illustration.

Each EdgeRecord in the EdgeFile corresponds to the set of edges of a particular EdgeType incident on a NodeID. The EdgeRecord for (NodeID, EdgeType) pair starts with `$NodeID#EdgeType`, where $ and # are two delimiters. Next, the EdgeFile stores certain metadata that we describe below. Following the metadata, the EdgeFile stores the TimeStamps for all edges, followed by destinationIDs for all edges, finally followed by the PropertyLists of all edges. We describe below the design decisions made for each of these individually.

Edge Timestamps. Edge timestamps are often used to impose ordering (*e.g.*, return results sorted by timestamps, or find new comments since last login time [30]). Efficiently executing such queries requires performing binary search on timestamps. ZipG stores timestamps in each EdgeRecord in sorted order. To aid binary search, ZipG also stores the number of edges in the EdgeRecord within the metadata (denoted by `EdgeCount` in Figure 4.2).

There are several approaches for storing individual timestamp values. At one extreme are variable length encoding and delta encoding. In the former, each timestamp can be stored using minimum number of bytes required to represent that timestamp along with some additional bytes (delimiters and/or length) to mark boundaries of timestamp values. While space-efficient, this representation complicates random access on timestamps since extracting a timestamp requires extracting all the timestamps before it. Storing timestamps using delta encoding [129] also leads to a similar tradeoff. The other extreme is fixed length representation for all edge timestamps (*e.g.*, 64 bits) that enables efficient random access at the cost of increased storage.

ZipG uses a middle-ground: it uses a fixed length representation but rather than using a *globally* fixed length, it uses the maximum length required across edges within an EdgeRecord. Since this length varies across EdgeRecords, ZipG stores the fixed length for each EdgeRecord in the corresponding metadata (`TLength` in Figure 4.2).

DestinationIDs. A natural choice for storing the destinationIDs is to order them according to edge timestamps, such that the i^{th} timestamp and i^{th} destinationID correspond to the same edge. Such an ordering avoids the need to maintain an explicit mapping between edge timestamps and corresponding destinationIDs, enabling efficient random access. ZipG uses a fixed length representation similar to timestamps for destinationIDs and stores this length in the metadata (`DLength` in Figure 4.2).

Edge Properties. As with destinationIDs, edge propertyLists are ordered such that i^{th} timestamp and i^{th} propertyList correspond to the same edge. The edge properties are en-

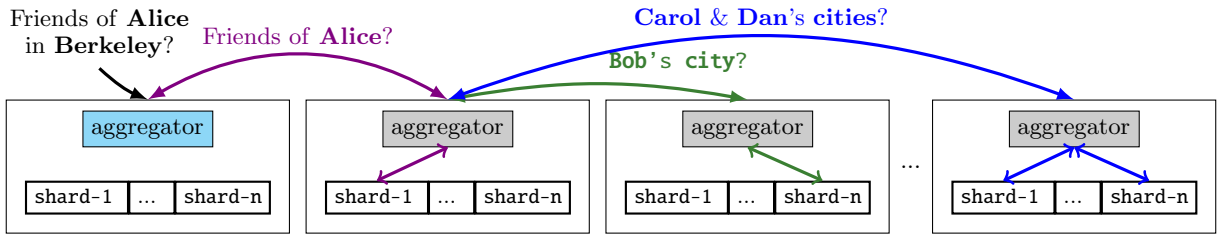


Figure 4.4: **Function Shipping in ZipG.** See §4.3.1 for discussion.

coded similar to node properties, since the layout design criteria and tradeoffs for both are identical. Specifically, the lengths of all the propertyLists are stored, followed by delimiter separated propertyValues (similar to NodeFile). ZipG currently does not support search on edge propertyLists, but can be trivially extended to do so using ideas similar to NodeFile.

4.2.3 ZipG Query Execution

We now describe how ZipG uses the graph layout from §4.2.2 to efficiently execute queries from Table 4.1.

Implementing node-based queries. It is easy to see that the NodeFile design allows implementing `get_node_property` query using two array lookups (one for property delimiter and one for ProperList offset), and one extra byte (for accessing propertyValue length) in addition to extracting the PropertyValue itself (using Succinct's `extract` primitive, see Figure 2.1 in Chapter 2). Implementing `get_node_ids` is more interesting; we explain this using an example. Suppose the query specifies {``nickname'' = ``Ally''} as the propertyList. Then, ZipG first finds the delimiter of the specified PropertyID (*, for `nickname`) and the next lexicographically larger PropertyID (in this case, ‡ for end-of-record delimiter). It then prepends and appends `Ally` by * and ‡, respectively and uses Succinct's `search` primitive (see Figure 2.1 in Chapter 2). This returns the offsets into the flat file where this string occurs, which are then translated into NodeIDs using binary search over the offsets in the two-dimensional array.

Implementing edge-based queries. The `get_edge_record` operation returns the `EdgeRecord` for a given (sourceID, edgeType) pair and is implemented using `search($sourceID #edgeType)` on the EdgeFile. This returns the offset for the EdgeRecord within the EdgeFile. Using the metadata in the EdgeRecord, ZipG can efficiently perform binary searches on the timestamps (`get_time_range`) and random access into the destination IDs and edge properties (`get_edge_data`).

4.2.4 Graph Partitioning (Sharding)

Several previous studies have established that efficient partitioning of social graphs is a hard problem [104, 106, 6]. Similar to existing graph stores [189], ZipG uses a simple

hash-partitioning scheme — it creates a number of shards, default being one per core, and hash partitions the NodeIDs on to these shards. All the data corresponding to NodeID (PropertyList and edge information of edges incident on NodeID) are then stored in that shard. This ensures that all node and edge data associated with a node is co-located on the same shard, enabling efficient execution of neighborhood queries. Finally, each of the shards is transformed into the ZipG layout comprising EdgeFiles and NodeFiles, as described in §4.2.2.

4.2.5 Fanned Updates

As outlined earlier, while storing data in a compressed form leads to performance benefits when the uncompressed data does not fit in faster storage, it also leads to the challenge of handling high write rates. Specifically, the overheads of decompressing and re-compressing data upon new writes need to be amortized over time, while maintaining low memory and computational overheads and while minimizing interference with ongoing queries on existing data. The traditional approach to achieve this is to use log-structured storage; within this high-level approach, there are two possible techniques that expose different tradeoffs.

The first technique is to maintain a log-structured store (LogStore), per shard or for all shards on a server, and periodically merge the LogStore data with the compressed data. To avoid scanning the entire LogStore during query execution (to locate the data needed to answer the query), additional `nodeID` → `LogStore-offset` pointers can be stored that allow random access for each node's data. The benefit of this approach is that all the data for any graph node remains "local". The problem however is that such an approach requires over-provisioning of memory for LogStore at each server, which reduces the overall memory efficiency of the system. Moreover, periodically merging the LogStore data with compressed data interferes with ongoing queries (as does copying the data to background processes for merging process). Finally, this approach requires using concurrent data structures to resolve read-write conflicts at *each* server (for concurrent reads and writes over LogStore data).

ZipG instead uses a *single* LogStore for the entire system — all write queries are directed to a query-optimized (rather than memory-optimized) LogStore. Once the size of the LogStore crosses a certain threshold, the LogStore is compressed into a memory-efficient representation and a new LogStore is instantiated. A single LogStore in ZipG offers three advantages. First, ZipG does not require decompressing and re-compressing the previously compressed data, and thus observes minimal interference on queries being executed on previously compressed data. Second, a single LogStore in ZipG also achieves higher memory efficiency than the first technique discussed above — rather than over-provisioning each server for a LogStore, ZipG requires just one dedicated LogStore server which can be optimized for query efficiency rather than memory efficiency. Finally, a single LogStore in ZipG avoids complicated data structures for concurrent reads and writes at each server, making the entire system simpler.

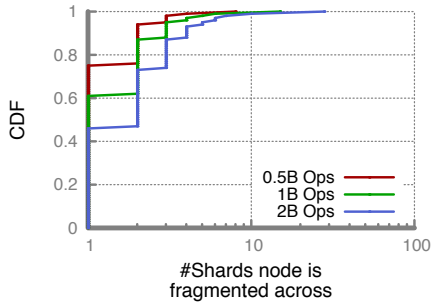
For unstructured and semi-structured data, where records are usually modeled as disjoint entities, a single LogStore leads to benefits compared to per-server LogStores [4, 95, 36, 8].

However, for graph structured data, using a single LogStore leads to a new challenge — *fragmented* storage. In the absence of decompression and re-compression, as new edges are added on a node, the `EdgeRecord` of the node will be fragmented across multiple shards. For instance, suppose at time $t = 0$ we upload the graph data to ZipG and create a single LogStore ℓ. Consider a node `u` that has some data in the originally uploaded data to ZipG. Suppose that multiple updates happen between time $t = 0$ and $t = t_1$ such that: (1) the size of ℓ crosses the threshold at time t_1; and (2) some of these updates are on node `u`. Then, at time t_1, we convert ℓ into ZipG's memory-efficient representation and create a new LogStore ℓ'. Then, for all updates on node `u` after time t_1, the data belonging to node `u` will now be fragmented across at least three shards: the original one that had the data for node `u` in pre-loaded graph data, the shard corresponding to the old LogStore ℓ and the new LogStore ℓ'. Depending on the update rates and on the skew in update queries, any node in the graph may thus have data fragmented across multiple shards over time (we evaluate the fragmentation over time and across nodes below). We thus need some additional techniques to efficiently exploit all the benefits of having a single LogStore.

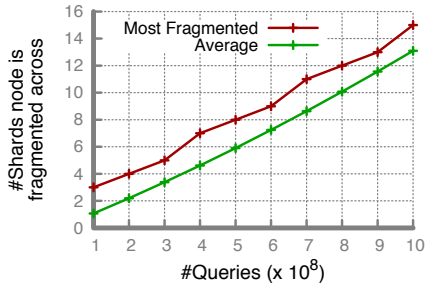
One way to handle fragmented data is to send each query to all shards, and retrieve the corresponding results. This is extremely inefficient — as our results Figure 4.5 below suggest, most queries can be answered by touching an extremely small fraction of the shards (less than 10% for 99.9% of the nodes); executing each query at all shards would thus have high unnecessary CPU overhead. ZipG instead uses the idea of `Fanned Updates`. Consider a static graph, that is, a graph that has never been updated since the initial upload to ZipG. The sharding scheme used in ZipG (described in §4.2.4) ensures that most ZipG queries are first forwarded to a single shard[5]. At a high-level, Fanned Updates avoid touching all shards using a set of update pointers that logically *chain* together data correspond to the same node or edge. As shown in Figure 4.3, these pointers store a reference to offsets of NodeFile and/or EdgeFile at other shards that store the updated data. ZipG stores these update pointers only at the shard where the node/edge first occurs; that is, in our example above, only the shard containing the data for node `u` in pre-loaded graph will store the update pointers for all occurrences in shards corresponding to ℓ, ℓ' and any other future shards. We describe below how ZipG uses fanned updates to optimize query execution. As the graph is updated or is fragmented over time, these update pointers are updated as well. For workloads where updates form a small fraction of all queries [30, 15, 91], the overhead of storing and updating these pointers is minimal. ZipG, thus, keeps these pointers uncompressed.

Data Fragmentation We now provide more insights on how data fragments over time and across nodes. We start with the LinkBench dataset (described later in Table 4.4) and partition it across 40 shards. We then execute LinkBench queries for a varying amount of time over a running system — we start with a single LogStore and when the size of

[5]Most ZipG queries in Table 4.1 are node-based and are first forwarded to the shard that stores queried node's data. Some of these queries may then be forwarded to shards that store node's neighbors's data. The only exception is `get_node_ids` that requires touching all shards.



(a) CDF of Fragmentation



(b) Fragmentation vs #Queries

Figure 4.5: (a) Most of the nodes have their data fragmented across a small number of shards. In addition, as expected, nodes have their data fragmented across more shards as more queries are executed by the system. (b) As more queries are executed, the data fragmentation gets worse for both: average fragmentation across all the nodes, and fragmentation for the most fragmented node (the one that has its data fragmented across the maximum number of shards).

the LogStore crosses 8GB threshold, we compress the LogStore data into a new shard and create a new LogStore. We then take snapshots of the system every time the system has executed 100 million LinkBench queries. Using these snapshots, we can evaluate the data fragmentation. Figure 4.5a shows the CDF (across all nodes) for the fraction of shards (among all shards at the time of the snapshot) that a particular node's data is fragmented across for three snapshots — after the system has executed $0.5, 1$ and 2 billion queries. We make two observations. First, for more than 99% of the nodes, their data is fragmented across a very small albeit non-trivial number of shards ($< 10\%$ of the shards in the system). This is precisely the case where update pointers help — ZipG needs to touch more than a single server, but touching all servers for query execution has significantly higher overheads than using update pointers. Moreover, as more queries are executed, data gets fragmented across a larger number of shards (also shown in Figure 4.5b).

ZipG query execution for updates. Fanned Updates require minimal extension to ZipG query execution for static graphs. In addition to executing query as in a static graph, the ZipG servers also forward the query to the precise servers that store updated data by following the update pointers, and collect the additional query results while avoiding touching all servers. Note that since most nodes and edges are unlikely to be updated frequently in real-world graph workloads [30, 15], a majority of read queries would be confined to a single server. ZipG implements deletes as lazy deletes with a bitmap indicating whether or not a node or an edge has been deleted; finally, updating a previously written record in ZipG is implemented as deletes followed by an append.

4.3 ZipG Implementation

We have implemented ZipG on top of Succinct using roughly 4000 lines of C++ code, as well as a package running atop Apache Spark in Scala. We start this section by outlining

some of the system implementation details (§4.3.1). We then show how ZipG API can be used to implement published functionalities from a variety of graph stores (§4.3.2).

4.3.1 System Implementation

We now outline the key aspects of ZipG implementation.

Fault Tolerance and Load Balancing. ZipG currently uses traditional replication-based techniques for fault tolerance; an application can specify the desired number of replicas per shard. Queries are load balanced evenly across multiple replicas. While orthogonal to ZipG design, extending current implementation to incorporate storage-efficient fault tolerance and skew-tolerant load balancing techniques [95, 46] is an interesting direction.

Data Persistence and Caching. To achieve data persistence, ZipG stores NodeFiles, EdgeFiles, newly added data on LogStore and the update pointers on secondary storage as serialized flat files. ZipG maps these files to virtual memory using the `mmap` system call, and all writes to them are propagated to the secondary storage before the operation is considered complete.

Query Execution via Function Shipping (Figure 4.4). Graph queries often require exploring the neighborhood of the queried node (*e.g.*, "friends of Alice who live in Berkeley"). To minimize network roundtrips and bandwidth utilization in a distributed setting, ZipG pushes computation closer to the data via *function shipping* [73, 177]. Each ZipG server hosts an *aggregator* process that maintains a pool of local threads for executing queries on the server. When an aggregator receives a query that requires executing subqueries on other servers, it *ships* the subqueries to the corresponding servers, each of which execute the subquery locally. Once all the subquery results are returned, the aggregator computes the final result. Indeed, ZipG also supports multi-level function shipping; that is, a subquery may be further decomposed into sub-subqueries and forwarded to respective servers.

Concurrency Control. Having a log-store for data updates significantly simplifies concurrency control in ZipG. The compressed data structures are immutable (except periodic garbage collection) and see only read queries; locks are only required at uncompressed update pointers and deletion bitmaps (§4.2.5), that are fast enough and do not become system bottleneck.

4.3.2 ZipG Expressiveness

ZipG design and interface is rich enough to implement published functionalities from several industrial graph stores. To demonstrate this, we have implemented all the published queries from Facebook TAO [30], LinkBench [15], Graph Search [91] as well as more complex graph queries such as regular path queries [18] and graph traversal queries [29] on top of ZipG. We now discuss these implementations and associated tradeoffs. Table 4.2 outlines the

Table 4.2: Queries in TAO [30] and LinkBench [15] workloads.

Query	Execution in ZipG	TAO %	LinkBench %
assoc_range	Algorithm 3	40.8	50.6
obj_get	get_node_property	28.8	12.9
assoc_get	Algorithm 4	15.7	0.52
assoc_count	get_edge_record	11.7	4.9
assoc_time_range	Algorithm 5	2.8	0.15
assoc_add	append	0.1	9.0
obj_update	delete, append	0.04	7.4
obj_add	append	0.03	2.6
assoc_del	delete	0.02	3.0
obj_del	delete	< 0.01	1.0
assoc_update	delete, append	< 0.01	8.0

Table 4.3: **The Graph Search Workload** and implementation using ZipG API; p1 and p2 are node properties, id and type are NodeID and EdgeType. All queries occur in equal proportion in the workload.

QID	Example	Execution in ZipG
GS1	All friends of **Alice**	get_neighbor_ids(id,*,*)
GS2	**Alice**'s friends in **Berkeley**	get_neighbor_ids(id,*,{p1})
GS3	**Musicians** in **Berkeley**	get_node_ids({p1,p2})
GS4	**Close** friends of **Alice**	get_neighbor_ids(id,type,*)
GS5	All **data** on **Alice**'s friends	assoc_range(id,type,0,*)

implementation for TAO and LinkBench queries[6], and Table 4.3 outlines the implementation of Graph Search queries using ZipG API.

Algorithm 3 assoc_range(id, atype, idx, limit): Obtain at most limit edges with source node id and edge type atype ordered by timestamps, starting at index idx.

```
1: rec ← get_edge_record(id, atype)
2: results ← ∅
3: for i ← idx to idx+limit do
4:     edgeEntry ← get_edge_data(rec, i)
5:     Add edgeEntry to results
6: return results
```

Facebook TAO queries are of two types. First, those that do not operate on Timestamps (obj_get and assoc_count in Table 4.2). These queries translate to obtaining all properties for a NodeID and counting edges of a particular type incident on a given NodeID. These are easily mapped to ZipG API — get_node_property(id, *) and the EdgeCount metadata using get_edge_record, respectively.

The second type of queries are based on Timestamps. For instance, consider the following query: "find all comments from Alice between SIGMOD abstract and paper submission

[6]The nodes and edges in ZipG are equivalent to *objects* and *associations* in TAO and LinkBench.

Algorithm 4 `assoc_get(id1, atype, id2set, hi, lo)`: Obtain all edges with source node `id1`, edge type `atype`, timestamp in the range `[hi,lo)`, and destination ∈ `id2set`.

```
1: rec ← get_edge_record(id1, atype)
2: (beg, end) ← get_time_range(rec, hi, lo)
3: results ← ∅
4: for i ← beg to end do
5:     edgeEntry ← get_edge_data(rec, i)
6:     Add edgeEntry to results if destination ∈ id2set
7: return results
```

Algorithm 5 `assoc_time_range(id, atype, hi, lo, limit)`: Obtain at most `limit` edges with source node `id`, edge type `atype` and timestamps in the range `[hi,lo)`.

```
1: rec ← get_edge_record(id, atype)
2: (beg, end) ← get_time_range(rec, hi, lo)
3: results ← ∅
4: for i ← beg to min(beg+limit, end) do
5:     edgeEntry ← get_edge_data(rec, i)
6:     Add edgeEntry to results
7: return results
```

deadlines". ZipG is particularly efficient for such queries due to its ability to efficiently perform binary search on timestamps (§4.2) and return corresponding edges and their properties. Algorithms 3, 4 and 5 show that these fairly complicated Facebook TAO queries can be implemented in ZipG using less than 10 lines of code.

LinkBench models Facebook's database workload for social graph queries. Note that TAO and LinkBench have the same set of queries, but vary significantly in terms of query distribution (LinkBench is much more write-heavy). Thus, ZipG implements LinkBench queries similar to TAO queries, as outlined above.

Facebook Graph Search originally supported several interesting, and complex, queries on graphs [91]. Implementing graph search queries is even simpler in ZipG since most queries directly map to ZipG API, as shown in Table 4.3.

Regular path queries and graph traversals differ significantly from the above queries. In particular, while the queries discussed above usually require information from the immediate neighborhood of a single node, path queries and traversals examine the structure for larger subgraphs. However, both of these classes of queries can be implemented in a recursive manner where each step requires access to the set of edges incident on a subset of nodes (and the corresponding neighbor nodes). In ZipG, this translates to sequences of `get_neighbor_ids`, `get_edge_record` and `get_edge_data` operations.

Table 4.4: Datasets used in our evaluation.

	Dataset	#nodes & #edges	Type	On-disk Size
Real-world	orkut [93]	∼ 3M & ∼ 117M	social	20 GB
	twitter [27]	∼ 41M & ∼ 1.5B	social	250 GB
	uk [27]	∼ 105M & ∼ 3.7B	web	636 GB
LinkBench	small	∼ 32.3M & ∼ 141.7M	social	20 GB
	medium	∼ 403.6M & ∼ 1.76B	social	250 GB
	large	∼ 1.02B & ∼ 4.48B	social	636 GB

4.4 Evaluation

We evaluate ZipG against popular open-source graph stores across graphs of varying sizes, real-world and benchmark query workloads, and varying cluster sizes.

Compared Systems. We compare ZipG against two open-source graph stores. Neo4j [136] is a single machine graph store and does not support distributed implementations. Our preliminary results for Neo4j were not satisfactory. We worked with Neo4j engineers for over a month to tune Neo4j and made several improvements to the Neo4j query execution engine. Along with the original version (Neo4j), we also present the results for this improved version (Neo4j-Tuned).

We also compare ZipG against Titan, a distributed graph store that requires a separate storage backend. We use Titan version 0.5.4 [189] with Cassandra 2.2.1 [102] as the storage backend. We also experimented with DynamoDB 0.5.4 for Titan but found it to be performing worse. Titan supports compression. We present results for both uncompressed (Titan) and compressed (Titan-compressed) representations.

We note that our comparisons are restricted to graph stores that serve interactive graph queries — graph processing systems like GraphX [79], GraphChi [100] and Pregel [119] support offline graph analytics (which run at the scale of minutes), and are not well suited to serve interactive graph queries.

Experimental Setup. All our experiments run on Amazon EC2. To compare against Neo4j, we perform single machine experiments over an r3.8xlarge instance with 244GB of RAM and 32 virtual cores. Our distributed experiments use 10 m3.2xlarge instances each with 30GB of RAM and 8 virtual cores. Note that all instances were backed by local SSDs and *not* hard drives. We warm up each system for 15 minutes prior to running experiments to cache as much data as possible. To make results consistent (Neo4j does not support graph partitioning across servers), we configured all systems to run without replication.

Workloads. Our evaluation employs a wide variety of graph workloads. We use TAO and Linkbench workloads from Facebook (with original query distributions, Table 4.2), and a synthetic Graph Search workload (Table 4.3). We also evaluate more complex workloads such as regular path queries [77] and graph traversal queries [29]. In addition, we evaluate the performance for each workload's component queries in isolation to build in-depth insights

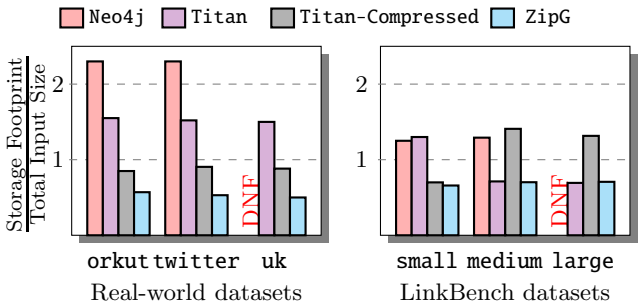


Figure 4.6: **ZipG's storage footprint (§4.4.1)** is $1.8-4\times$ lower than Neo4j and $1.8-2\times$ lower than Titan. `DNF` denotes that the experiment did not finish after 48 hours of data loading.

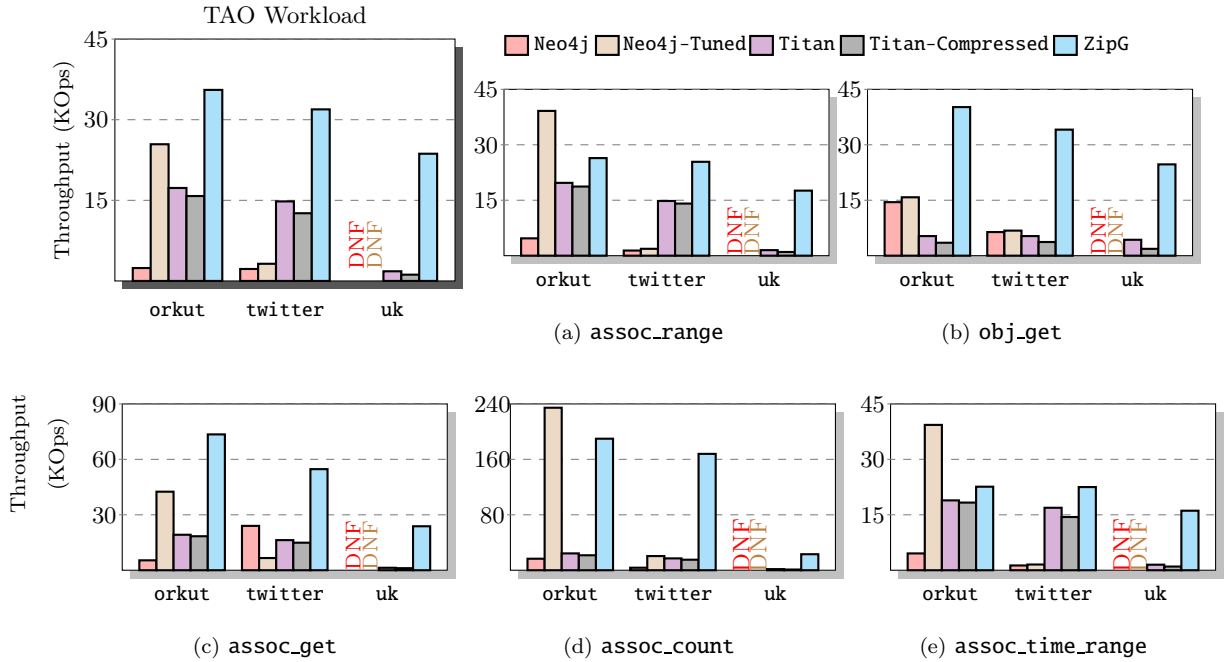


Figure 4.7: **Single server throughput for the TAO workload, and its top 5 component queries in isolation.** `DNF` indicates that that the experiment did not finish after 48 hours of data loading. Note that the figures have different y-scales.

on the performance of the three systems.

Datasets. Table 4.4 shows the datasets used in our evaluation. For real-world datasets, we used the node and edge property distribution from Facebook TAO paper [30]. Each node has an average propertyList of 640 bytes distributed across 40 propertyIDs. Each edge is randomly assigned one of 5 distinct `EdgeType`s, a POSIX timestamp drawn from a span of 50 days, and a 128-byte long edge property. For LinkBench datasets, we directly use the LinkBench benchmark tools [113] to generate three datasets: `small`, `medium` and `large`. These datasets mimic the Orkut, Twitter and UK graphs in terms of their total on-disk size. LinkBench assigns a single property to each node and edge in the graph, with the properties

Table 4.5: Summary of which datasets fit completely in memory.

Dataset	Neo4j	Titan-C	Titan	ZipG
orkut / linkbench-small	✓	✓	✓	✓
twitter / linkbench-medium		✓		✓
uk / linkbench-large				

having a median size of 128 bytes.

4.4.1 Storage Footprint

Figure 4.6 shows the ratio of total data representation size and raw input size for each system. We note that ZipG can put $1.7 - 4\times$ larger graphs in main memory compared to Neo4j and Titan uncompressed (which, as we show later, leads to degraded performance[7]). The main reason is the secondary indexes stored by Neo4j and Titan to support various queries efficiently; ZipG, on the other hand, executes queries efficiently directly on a memory-efficient representation of the input graph.

Since LinkBench assigns synthetically generated properties to nodes and edges, LinkBench datasets have lower compressibility compared to the real-world datasets. Accordingly, ZipG's compression factor is roughly 15% lower for the LinkBench datasets than the corresponding real-world datasets. The storage overheads for Neo4j and Titan, on the other hand, are lower for the LinkBench datasets, since they have to maintain much smaller secondary indexes for a single node property. As such, ZipG's storage footprint is $1.8 - 2\times$ smaller than Neo4j and Titan uncompressed, while being comparable to Titan-Compressed.

Table 4.5 summarizes which datasets fit *completely* in memory for different systems our experiments.

4.4.2 Single Machine (Figure 4.7, 4.8, 4.9)

We now analyze the performance of different graph stores on a single server with 244GB of RAM and 32 CPU cores. We note that across all experiments, Neo4j-Tuned achieves strictly better performance than Neo4j. Similarly, Titan uncompressed achieves strictly better performance Titan compressed (for reasons discussed in Footnote 7). The discussion thus focuses on Neo4j-Tuned, Titan uncompressed and ZipG.

TAO Workload (Figure 4.7)

We start by observing that when the dataset fits in memory (*e.g.*, Orkut), all systems achieve comparable performance. There are two reasons for ZipG achieving slightly better

[7]Intuitively, Titan uses delta encoding for edge destinationNodeIDs, and variable length encoding for node and edge attributes [190] which leads to high CPU overhead during query execution. Moreover, enabling LZ4 compression for Cassandra's SSTables reduces the storage footprint for Titan, but required data decompression for query execution.

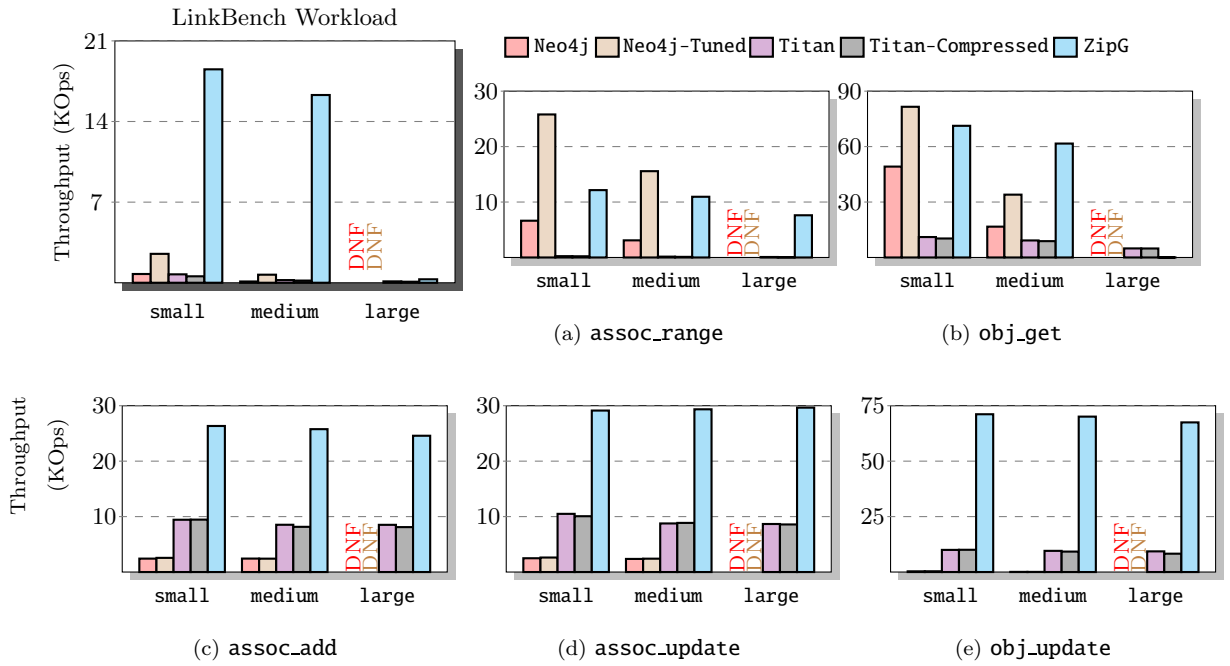


Figure 4.8: **Single server throughput for the LinkBench workload, and its top 5 component queries in isolation.** `DNF` denotes that the experiment did not finish after 48 hours of data loading. Note that the figures have different y-scales.

performance than Neo4j and Titan. First, ZipG is optimized for random access on node PropertyList while Neo4j and Titan are not — Neo4j requires following a set of pointers on NodeTable, while Titan needs to first extract the corresponding (key, value) pair from Cassandra and then scan the value to extract node properties. The second reason ZipG performance is slightly better is that ZipG extracts all edges of a particular edgeType directly, while other systems have to scan the entire set of edges and filter out the relevant results.

For the Twitter dataset, Neo4j can no longer keep the entire dataset in memory; Titan, however, retains most of the working set in memory due to its lower storage overhead than Neo4j and also because TAO queries do not operate on edge PropertyList. Neo4j observes significant impact in throughput for a reason that highlights the limitations of pointer-based data model of Neo4j — since pointer-based approaches are "sequential" by nature, a single application query leads to multiple SSD lookups leading to significantly degraded throughput. Titan, on the other hand, maintains its throughput for all queries. This is both because Titan has to do fewer SSD lookups (once the key-value pair is extracted, it can be scanned in memory) and also because Titan essentially caches most of the working dataset in memory.

For the UK dataset, none of the systems can fit the data in memory (Neo4j cannot even scale to this dataset size). Titan now starts experiencing significant performance degradation due to a large fraction of queries being executed off secondary storage (similar to the performance degradation of Neo4j in Twitter dataset). ZipG also observes performance degradation but of much lesser magnitude than other systems because of two reasons. First,

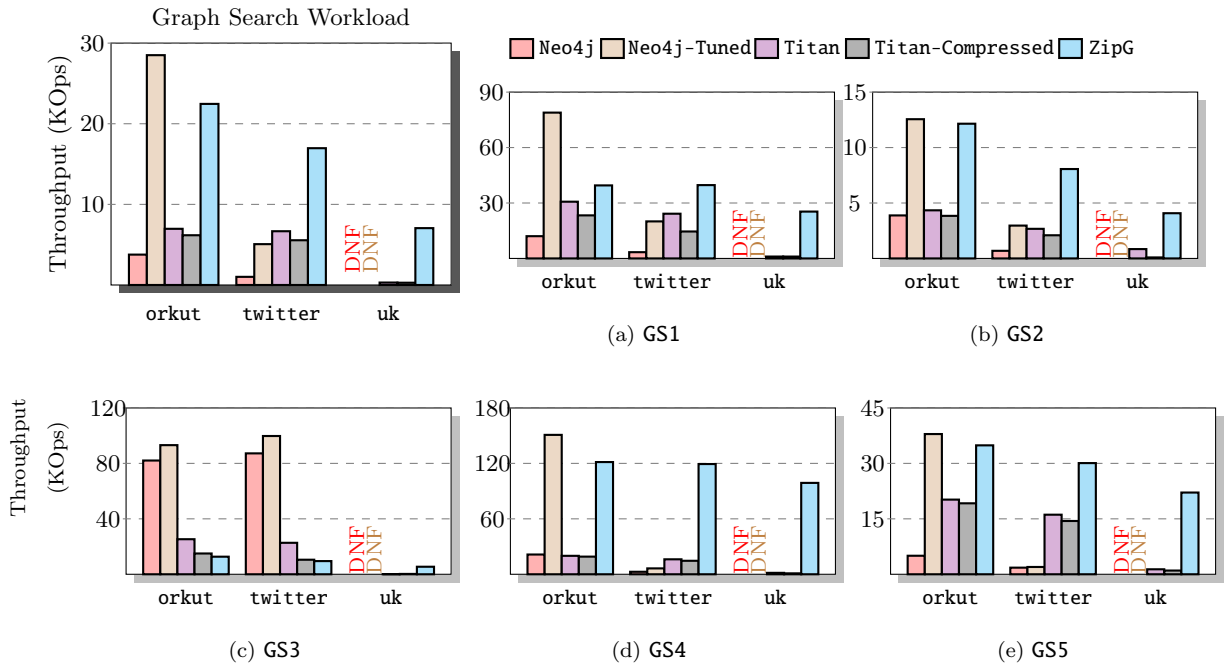


Figure 4.9: **Single server throughput for the Graph Search workload, and its component queries in isolation.** `DNF` indicates that that the experiment did not finish after 48 hours of data loading. Note that the figures have different y-scales.

ZipG is able to execute a much larger fraction of queries in memory due to its lower storage overhead; and second, even when executing queries off secondary storage, ZipG has significantly lower I/O since it requires a single SSD lookup for all queries unlike Titan and Neo4j.

Individual TAO queries (Figure 4.7a-4.7e). Analyzing the performance of the top 5 TAO queries for the Orkut dataset, node-based queries involving random access (`obj_get`, Figure 4.7b) perform better for ZipG than Neo4j and Titan due to reasons cited earlier. Similarly, for the edge-based queries (*e.g.*, `assoc_get`, Figure 4.7c), ZipG achieves higher throughput by avoiding the overheads of scans employed by other systems. When queries have a `limit` on the result cardinality, other systems can stop scanning earlier and thus achieve relatively improved performance, *e.g.*, `assoc_range` and `assoc_time_range` in Figures 4.7a and 4.7e respectively. For larger datasets, while compared systems fail to keep their data in memory, ZipG achieves considerably higher throughput for all the individual queries, since its lower storage footprint permits execution of most queries in memory.

LinkBench Workload (Figure 4.8)

Despite having the same set of queries as TAO, the absolute throughput for the LinkBench workload is distinctly lower for all systems. This is due to two main reasons: first, a much larger fraction of the queries (see Table 4.2) are either write, update or delete operations,

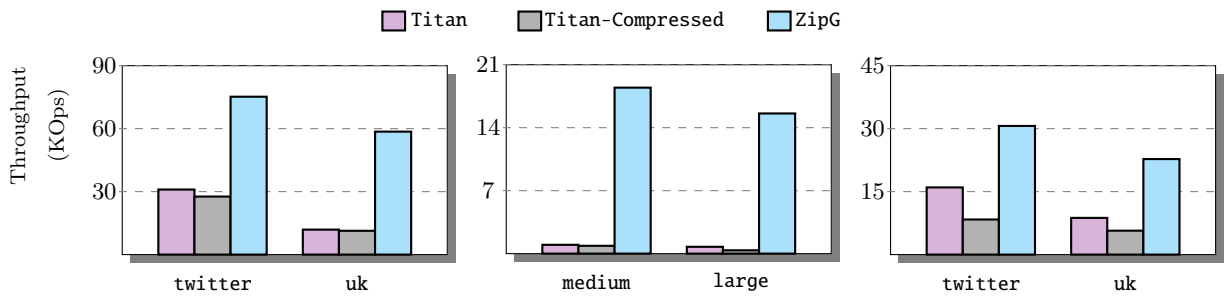
requiring modification of graph elements. This leads to overheads due to data persistence, as well as lock-based synchronization for atomicity and correctness of graph mutations in all compared systems. Second, most of the queries perform filters on node neighborhoods, with their accesses being skewed towards nodes with more neighbors [15] — as a result, the average number of edges accessed per query is much larger than in the TAO workload, leading to lower query throughput.

Also observe that Neo4j and Titan observe much lower throughput than ZipG for all datasets. While Neo4j is relatively efficient in executing read-only queries, write queries become a significant performance bottleneck, since they require modifications at multiple random locations due to Neo4j's pointer based-approach. Titan, on the other hand, is able to support write and update operations at relatively higher throughput due to Cassandra's write-optimized design. However, the throughput for edge-based operations is significantly lower because Cassandra is not optimized for range queries.

ZipG avoids both of the above issues for the `small` and `medium` datasets. In particular, all graph mutations are isolated to a write-optimized LogStore through Fanned Updates (§4.2.5), while edge-based operations do not need to scan the entire neighborhood to filter the required edges (§4.2.2). However, ZipG's throughput drops for the `large` dataset; this is due to the relatively lower compressibility of LinkBench generated graphs, which prevents crucial data-structures in the underlying Succinct representation for the NodeFile from fitting in memory.

Individual LinkBench Queries (Figures 4.8a-4.8e). Note that the top 5 queries in the LinkBench workload differ from the TAO workload. The performance trends for the `assoc_range` (Figure 4.8a) and `obj_get` (Figire 4.8b) queries are similar to the corresponding TAO queries, except for a few key differences. First, Titan observes significantly worse performance for `assoc_get` query in the LinkBench workload. This is because the average number of neighbors for each node in the LinkBench dataset is much larger than the TAO workload, and is heavily skewed, i.e., some nodes have very large neighborhoods, while most others have relatively few neighbors. Titan's performance drops significantly due to range queries over large neighborhoods, since Cassandra is not optimized for such queries, resulting in reduced throughput. Second, Neo4j observes better performance for `obj_get` query in the LinkBench workload, because of the workload's *query* skew. Since the accesses to the nodes for `obj_get` query are heavily skewed, Neo4j is able to cache the most frequently accessed nodes, leading to higher throughput. Finally, for the `large` dataset, ZipG is unable to keep one of Succinct's component data structures in memory that is responsible for answering node-based queries, leading to a reduced throughput for the `obj_get` query.

Finally, we note that ZipG outperforms compared systems for write-based queries including `assoc_add` (Figure 4.8c), `assoc_update` (Figure 4.8d) and `obj_update` (Figure 4.8e). As discussed above, Neo4j's write performance suffers since each write incurs updates at multiple random locations in its graph representation. Titan achieves a relatively better performance due to Cassandra's write-optimized design. ZipG, on the other hand, is able to maintain a high write throughput due to its write optimized LogStore and Fanned Updates



(a) TAO Workload (b) LinkBench Workload (c) Graph Search Workload

Figure 4.10: **Throughput for TAO, LinkBench and Graph Search workloads for the distributed cluster.**

approach.

Graph Search Workload (Figure 4.9)

We designed the graph search workload for two reasons. First, while TAO and LinkBench workloads are mostly random access based, graph search workload mixes random access (`GS1, GS4, GS5`) and search (`GS2, GS3`) queries. Second, this workload highlights both the power and overheads of ZipG. In particular, as shown in Table 4.3, ZipG's powerful API enables simple implementation of queries that are far more complex than the TAO and LinkBench queries. Indeed, most of the graph search queries can be implemented using a couple of lines of code on top of ZipG API. On the flip side, the graph search workload also highlights the overheads of executing queries on compressed graphs. We discuss the latter below.

The results for the graph search workload follow a very similar pattern as for TAO workload (Figure 4.7, left), with two main differences. First, as with the LinkBench workload, the overall throughput reduces for all systems. This is rather intuitive — search queries are usually far more complex than random access queries, and hence have higher overheads. Second, when the uncompressed graph fits entirely in memory, Neo4j-Tuned achieves better performance than ZipG. The latter highlights ZipG overheads. In particular, for the Orkut dataset, both Neo4j-Tuned and ZipG fit the entire data in memory. However, in graph search workload, Neo4j could use its indexes to answer search queries (and avoid heavy-weight neighborhood scans). As a result, for the Orkut dataset, Neo4j starts observing roughly 1.23× higher throughput than ZipG as opposed to lower throughput for TAO queries, which is attributed to ZipG executing queries on compressed graphs. Of course, as the graph size increases, the overhead of executing queries off secondary storage becomes higher than executing queries on compressed graphs, leading to ZipG achieving 3× higher throughput than Neo4j-Tuned.

The second overhead of ZipG is for search-based queries like "Find musicians in Berkeley". For such a query, ZipG's partitioning scheme requires ZipG touching all partitions. Neo4j and Titan, on the other hand, use global indexes and thus require touching no more than two partitions. Thus, for small datasets, ZipG observes significantly lower throughput for

this query than Neo4j and Titan. As earlier, for larger graph sizes, this overhead becomes smaller than the overhead of executing queries off secondary storage and ZipG achieves higher throughput.

Individual Graph Search Queries (Figure 4.9a-4.9e). Most Graph Search queries follow trends similar to the TAO workload since they are random-access intensive. In particular, queries `GS1` (Figure 4.9a), `GS4` (Figure 4.9d) and `GS5` (Figure 4.9e) perform random-access on edge data, while query `GS2` (Figure 4.9b) performs random access on both edge and node data. For all such queries, Neo4j-Tuned achieves high throughput for the `orkut` dataset since all the data fits in memory, but as the datasets no longer fit in memory the performance drops drastically due to Neo4j's pointer-based approach, similar to the TAO workload. Titan, on the other hand, extracts the data corresponding to nodes and edges as key value pairs and scans the value component to obtain the relevant data (node or edge properties), resulting in lower throughput, which drops even lower when the datasets no longer fit in memory. ZipG, on the other hand, exploits its random-access friendly layout to achieve high throughput for all such queries, with little degradation on increasing the dataset size due to its memory-efficient graph representation.

Finally, query `GS3` (Figure 4.9c) is unique, in that, it performs search queries on node attributes. As discussed before, ZipG's performance for this query is comparable or worse than the compared systems for the `orkut` dataset, since it touches multiple partitions to evaluate search results while other systems use global indexes. However, as the dataset sizes grow larger, the compared systems observe much lower throughput since these indexes no longer fit in memory, resulting in expensive accesses to secondary-storage.

We note that, as discussed earler, while executing queries `GS2` and `GS3`, ZipG exploits the cardinality of intermediate outputs (*e.g.*, number of friends Alice's friends vs. the number of people in Berkeley) to pick the appropriate join strategy. We provide performance comparisons for these queries executed with different join strategies in ZipG in Appendix 4.4.6.

4.4.3 Distributed Cluster (Figure 4.10)

Neo4j does not have a distributed implementation. We therefore restrict our discussion to the performance of ZipG and Titan on a distributed cluster of 10 servers, with a total of 300GB of RAM and 80 CPU cores across the cluster.

TAO Workload (Figure 4.10a)

We make two observations. First, Titan can now fit the entire Twitter dataset in memory leading to 2× higher throughput compared to single server setting, despite the increased overhead of inter-server communication (similar remarks for UK dataset). The second observation is that ZipG achieves roughly 2.5× higher throughput in distributed settings compared to single server setting. Note that our distributed servers have 10×8 cores, 2.5× of the single beefy server that has 32 cores. ZipG thus achieves throughput increase proportional to the increase in number of cores in the system, an ideal scenario.

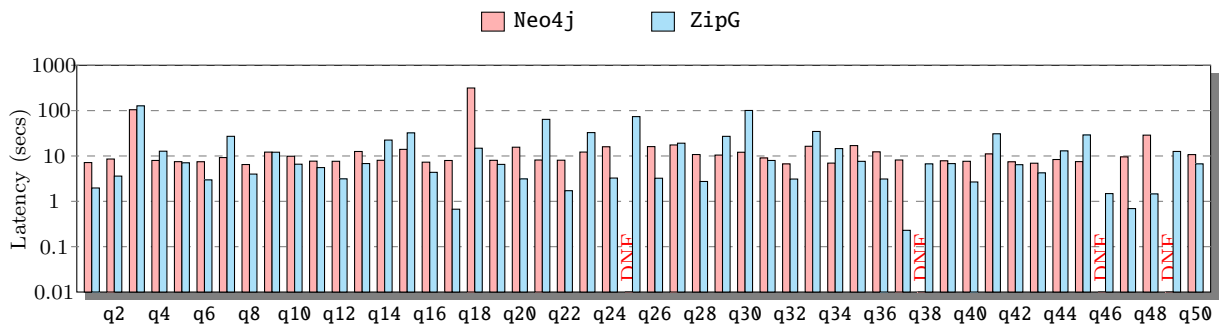


Figure 4.11: **Latency for executing Regular Path Queries from gMark workload.** Note that the y-axis has a `log` scale. For most of the queries, ZipG performs significantly better than Neo4j. However, for some queries, ZipG does perform worse than Neo4j. We discuss the properties of the queries that lead to such performance in §4.4.4.

LinkBench Workload (Figure 4.10b)

These results allow us to make an interesting observation — unlike single machine setting, ZipG is able to cache a much larger fraction of crucial Succinct data-structures, leading to almost negligible performance degradation on going from the `medium` to the `large` dataset. Second, unlike the TAO workload, ZipG is unable to achieve throughput increase proportional to the number of cores. This is because the access pattern for edge-based queries in LinkBench is skewed towards nodes that have larger neighborhoods. As a consequence, a small set of servers that store nodes with large neighborhoods remain bottlenecked due to higher query volume and computational overheads.

Graph Search Workload (Figure 4.10c)

Again, most performance trends for the Graph Search workload are similar for the distributed cluster and single server settings. We note that Titan's performance for the Graph Search workload scales better than ZipG's performance when the number of servers is increased. This is due to the contribution of search based queries, i.e., query `GS3`. As discussed in §4.4.2, the difference in performance for such queries lie in the choice of partitioning scheme for the two systems. While ZipG touches all partitions, and therefore all servers in the cluster for search-based queries, Titan's global index approach confines such queries to a single server for most situations, allowing the query performance to scale better. However, Titan's global index approach suffers in performance when the index grows too large to fit in memory.

4.4.4 Regular Path Queries

General regular path queries [45, 33, 110] identify paths in graphs through regular expressions over the edge labels (`edgeType`s in ZipG terminology) of the graph. Their results are collections of paths, where the concatenation of consecutive edge labels in each path satisfy the regular expression.

Implementation. We implemented *unions of conjunctive regular path queries* [18] in ZipG by executing regular expressions over edge labels on ZipG layout, translating them into sequences of operations from ZipG's API. In fact, ZipG is able to execute all regular path queries generated by the gMark benchmark tool [17].

The execution of regular path queries in ZipG begins by obtaining all EdgeRecords corresponding to the first edge in the path expression using `get_edge_record(*,edgeType)`. Subsequently, ZipG identifies the neighbor nodes in these EdgeRecords using `get_edge_data`, and iteratively searches for non-empty EdgeRecords corresponding to the neighbor nodes and the subsequent edge labels in the path expression. To minimize communication overheads, ZipG *ships* the `get_edge_record` requests to the shards that hold the data for the particular node using function shipping (§4.3.1). ZipG supports recursion in path queries via the Kleene-star ("*") operator by computing the transitive closure of the set of paths identified by the path expression under the Kleene-star. Currently, the transitive closure computation requires collecting all the paths at an aggregator and employs a serial algorithm; this can be further optimized using careful modifications in the underlying data structures and query execution.

Performance. In order to evaluate the performance of regular path queries, we used the gMark [17] benchmark tool to generate both the graph datasets and the path queries. We used gMark's encoding of the schema provided with the LDBC Social Network Benchmark [58], which models user activity in a typical social network. gMark generates 50 queries of widely varying nature [77], ranging from linear path traversals, to branched traversals and highly recursive queries; these can be easily mapped to their Cypher representations [139]. We ran our benchmarks for datasets with varying number of nodes and edges for ZipG and Neo4j; Figure 4.11 shows results for graphs with 2 million nodes on a single machine setup[8].

Note that given the dataset size, both systems are able to fit their entire data completely in memory. For most queries, ZipG's performance is either comparable to or better than Neo4j. The queries where ZipG outperforms Neo4j by a large margin (*e.g.*, `q18`, `q25`, `q38`, `q48`, `q49`, etc) are typically branched or long linear path traversals, with little or no recursion in them. On the other hand, Neo4j outperforms ZipG for queries that are heavy on recursion, or where computing the transitive closure becomes a bottleneck in ZipG (*e.g.*, `q4`, `q15`, `q21`, `q29`, `q30`, etc.). This is precisely due to the communication and serial bottlenecks in executing transitive closure in ZipG, as discussed above.

4.4.5 Graph Traversal

We now compare the performance for breadth first traversal of graphs for Neo4j and ZipG on a single machine. The traversals were performed starting at 100 randomly selected nodes,

[8]Due to the complexity of the queries, both systems timed out (with a time limit of 10 minutes) in executing most queries on graphs with more than 2 million nodes on a single 8 core machine. Moreover, despite significant effort, we could not run these queries on Titan even for smaller graphs due to some bug in the Titan release that supports regular path queries.

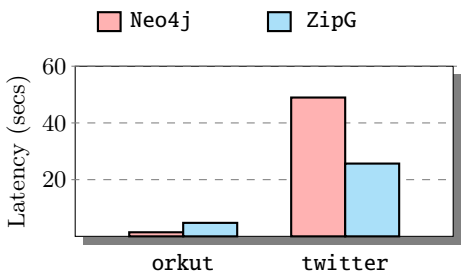


Figure 4.12: **Breadth First Traversal Latency for Neo4j and for ZipG.** When the entire graph data fits in memory (orkut), Neo4j performs better than ZipG. However, when the data does not fit in memory for Neo4j, ZipG outperforms Neo4j for graph traversal queries.

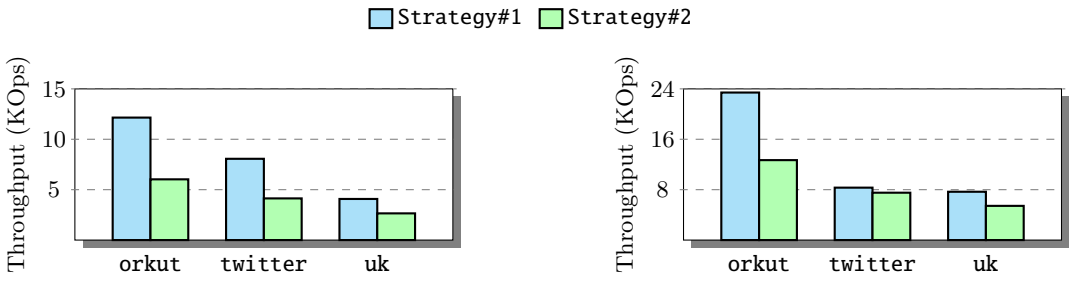
and the average traversal latencies for the two systems are shown in Figure 4.12. We bound the traversal depth to 5 for the datasets — unbounded depth led to timeouts for Neo4j for the larger dataset (that does not fit in memory).

For the orkut dataset, the graph fits completely in memory for both systems. For this case, Neo4j achieves lower latency. This is because ZipG has overheads to execute queries on the compressed representation; in addition, ZipG stores its graph data across multiple shards, and incurs some aggregation overheads in combining results from different shards. However, for the twitter dataset, Neo4j is no longer able to fit its data in memory, and incurs significantly higher latency for breadth first traversals. This is because even the data for a single node not fitting in memory requires Neo4j to access the slower storage, significantly slowing down the overall graph traversal query. ZipG, on the other hand, is able to maintain its data in memory, and thus achieves lower query latency compared to Neo4j.

4.4.6 Graph Queries with different Join Strategies

ZipG implementation for queries outlined so far avoids inefficient join strategies to ensure optimal performance of graph queries (as discussed in the introduction). In order to highlight the impact of inefficient join strategies, we execute queries `GS2` and `GS3` from the Graph Search workload using two different join strategies. In particular, a `GS2` query of the form "Find Alice's friends in Ithaca" (Table 4.3), can be executed by finding the intersection of all of Alice's friends and all people living in Ithaca, or by finding all of Alice's friends, and then checking if they live in Ithaca. The same holds true for a `GS3` query.

Figure 4.13 shows the performance for the two execution alternatives in ZipG on a single machine. Clearly, the alternative that employs the second strategy (`Strategy#2`) yields higher throughput across different graph datasets. Intuitively, this is because Alice is likely to have much fewer friends than the people living in Ithaca, and checking if Alice's friends is more efficient than finding the intersection of her friends and the people living in Ithaca. Moreover, the former approach has added communication overheads in distributed settings.



(a) GS2 (b) GS3

Figure 4.13: **Executing queries with different join strategies in ZipG.**

4.5 Related Work

Graph Stores. In contrast to graph batch processing systems [79, 116, 172, 100], graph stores [30, 72, 189, 136, 141, 171, 178, 127, 122, 56] usually focus on queries that are user-facing. Consequently, the goal in design of these stores is to achieve millisecond-level query latency and high throughput. We already compared the performance of ZipG against Neo4j and Titan, two popular open-sourced graph stores. Other systems, *e.g.*, Virtuoso [197], GraphView [127] and Sparksee [178] that use secondary indexes for efficiently executing graph traversals suffer from storage overhead problems similar or to Neo4j (high latency and low throughput due to queries executing off secondary storage).

Graph Compression. Traditional block compression techniques (*e.g.*, gzip) are inefficient for graphs due to lack of locality: each query may require decompressing many blocks. Several graph compression techniques that focus on supporting queries on compressed graphs [24, 27, 38, 66, 85, 123, 84, 173, 118] are limited in expressiveness to queries like extracting adjacency list of a node, or matching subgraphs. Graph serving often requires executing much more complex queries [30, 91, 200, 53] involving node and edge attributes. ZipG achieves compression without compromising expressiveness, and is able to execute all published queries from Facebook TAO [30], LinkBench [15] and graph search [91].

4.6 Summary

We have presented ZipG, a distributed memory-efficient graph store that supports a wide range of interactive graph queries on compressed graphs. ZipG exposes a minimal but functionally rich API, which we have used to implement all the published queries from Facebook TAO, LinkBench, and Graph Search workloads, along with complex regular path queries and graph traversals. Our results show that ZipG can executes tens of thousands of queries from these workloads for a graph with over half a TB of data on a single 244GB server. This leads to as much as 23× higher throughput than Neo4j and Titan, with similar gains in distributed settings where ZipG achieves up to 20× higher throughput than Titan.

Chapter 5

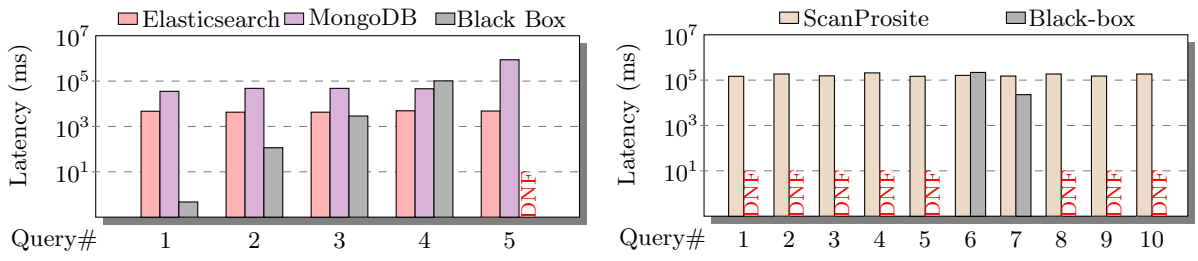
Executing RegEx Queries on Compressed Data

Continuing the theme of supporting richer query semantics using Succinct from Chapter 4, this chapter focuses on *regular expression* (RegEx) queries, a powerful tool for text analytics and information extraction. Traditionally, RegEx have been used in applications like textual data analytics [57, 145], information extraction [62, 28, 109, 47, 26, 98, 39] and bioinformatics [132, 75]. Unsurprisingly, efficiently executing queries involving RegEx is a problem that has been studied for decades.

However, RegEx have recently witnessed a renewed interest due to queries involving RegEx becoming both more important and more challenging. Increasingly many applications use RegEx across various stages in their data analytics pipeline including natural language processing [138, 170, 76, 179], recommender systems [168, 153] and even interactive queries on graph data [65, 64, 20, 19]. One case in point is Apache Spark [204], a popular open-source framework for distributed data analytics, where users frequently execute complex RegEx queries for text analytics and machine learning pipelines.

Queries involving RegEx have also become more challenging due to increasingly large data sizes in above applications. Traditional techniques for executing RegEx queries (*e.g.*, full-data scans [156, 155] and word-based indexes supported by partial data scans [165, 57, 145]) are memory-efficient, allowing the data to be stored and scanned in main memory. However, these techniques suffer from new scalability issues — data scans do not scale well with input data size, resulting in high query latency as the input size grows to tens or hundreds of gigabytes [40, 192, 146, 2]. On the other hand, powerful indexes like suffix trees and tries [40, 201, 120, 11] have significantly better query latency. However, these indexes often have high storage overheads [99, 86, 120]; for large datasets, these indexes suffer from degraded performance when the index size grows larger than the available memory [4, 40].

This chapter builds on Succinct to achieve the best of the two worlds — memory-efficiency of scan-based techniques and performance of powerful indexes. These compressed data structures support exact match of strings of arbitrary length in the input data as well as random access of the input data. Our main contribution is Sprint, a query execution engine that



(a) Text Analytics

(b) Bioinformatics

Figure 5.1: The black-box approach for RegEx execution can be just as slow as, or even slower than, existing scan-based approaches for many RegEx queries (see §5.3.5 for details on queries and experimental setup). Queries marked **DNF** did not finish within 10 minutes of execution time.

extends the functionality from exact string match to RegEx queries directly on these compressed data structures (that is, without requiring decompression). By storing and querying a compressed representation of powerful indexes, Sprint not only avoids data scans but also avoids the performance degradation due to indexes not fitting in main memory.

Sprint uses two key insights. The first insight is regarding the main challenge in efficiently executing RegEx queries on compressed data. Consider the following "black-box" approach (§5.2) — decompose the RegEx into *tokens*[1], search for individual tokens using compressed indexes (that support search of arbitrary substrings in input file), and combine the intermediate results along the RegEx operators. Figure 5.1 shows that naïvely executing the black-box approach can actually lead to performance even worse than scan-based techniques. The observation in Figure 5.1 is not merely an experimental artifact; our key insight here is a simple, yet surprising, analytical result supporting the result of Figure 5.1 (§5.2) — under the standard algorithmic cost model, if the RegEx query contains `Concatenation` operator, the execution time of the black-box approach could be arbitrarily far from optimal. Perhaps more surprisingly, we show that the black-box approach executes in near-optimal time if the RegEx query comprises of `Union, Repeat` and `Wildcard` operators only.

Our second insight is that RegEx queries containing `Concatenation` can be efficiently handled via query re-writing. Intuitively, given an input RegEx query, we can perform a series of transformations to eliminate the `Concatenation` operator (by concatenating multiple smaller tokens into a longer token); this results in a new equivalent RegEx query that contains only `Union, Repeat` and `Wildcard` operators along with (potentially longer) tokens. Since the compressed indexes support exact match of arbitrary strings, we could then execute the black-box approach on this new equivalent query. We present the Sprint transformations for such RegEx query re-writing in §5.3.

We present evaluation of Sprint[2] over real-world and benchmark datasets in §5.3.5. We compare Sprint against four popular open-source systems that support RegEx query execu-

[1]Tokens are parts of RegEx that do not contain operators. For instance, a RegEx `(Yo|Ho)(Ho+)` has two tokens `Yo` and `Ho`.

[2]Our implementation of Sprint, including all the datasets and queries necessary to reproduce our results are open-source: https://github.com/amplab/sprint. We have also implemented Sprint on top of Apache Spark; this implementation is being used in production and can be easily run on any Apache Spark cluster.

tion, including ElasticSearch [57], MongoDB [145], ScanProsite [74] and Apache Spark [204]. We find that Sprint achieves significant speedups compared to these systems, often as high as two orders of magnitude.

Interestingly, many Sprint techniques turn out to have more general applicability and lead to performance improvements even for uncompressed data structures. We have implemented Sprint on top of a variety of data structures, including inverted indexes [165], suffix trees [201], suffix arrays [120], compressed suffix trees [11], and compressed suffix arrays [82, 162, 164][3].

In summary, this chapter makes three contributions:

- We analyze the black-box approach to executing RegEx queries on compressed data. We show that the black-box approach over RegEx queries containing only `Union, Wildcard` and `Repeat` operators executes in near-optimal time; however, when the query contains `Concat` operator, the execution time of black-box approach could be far from optimal.
- We present Sprint — a simple, yet efficient, RegEx query engine that enables execution of RegEx queries directly on compressed data. We evaluate Sprint against four popular open-source systems that support RegEx queries. The evaluation shows that Sprint leads to significant speed up in RegEx query execution latency, sometimes by as much as two orders of magnitude.
- We show that Sprint techniques are applicable to several uncompressed data structures as well. In addition, we provide an open-source implementation of Sprint on top of a wide range of data structures including inverted indexes, suffix trees and compressed indexes, as well as on top of Apache Spark [204].

5.1 Preliminaries

We start with a description of the notation used in the chapter.

Notation. Throughout the chapter, we use the usual definitions of RegEx operators, as summarized in Table 5.1. The supported RegEx syntax is the POSIX extended standard [60]. Let Σ denote a totally ordered set of alphabets in the input. The operators are interleaved by *tokens*, which can be either (a) *character class*, denoted by '`[]`'; for example, `[0-9a-dA-F]` represents any character from `0` through `9`, `a` through d, and `A` through `F`; or (b) *m-gram*, which is a sequence of m alphabets from Σ.

RegExTree. A RegEx can equivalently be represented as a binary tree that takes standard precedence constraints between operators into account [78, 186]. We call this tree a RegExTree. Each internal node of the RegExTree represents a RegEx operator, while the leaves represent tokens (see Figure 5.2). The problem of constructing an optimized RegExTree has been explored in a number of previous works [7, 78, 186, 37] and is orthogonal to Sprint techniques. We use an optimized RegExTree from [37] as an input to Sprint.

[3]Implementation also available in the open-source release.

Table 5.1: Supported operator classes.

Operator	Contents	Explanation
Concat	(RE_1)(RE_2)	RE_2 immediately follows RE_1
Union	RE_1—RE_2	Either RE_1 or RE_2
Repeat	RE?, RE*, RE+	Concat of RE with RE Zero or one (?), Zero or more (*), One or more (+)
Wildcard	(RE_1).* (RE_2)	RE_2 occurs anywhere after RE_1

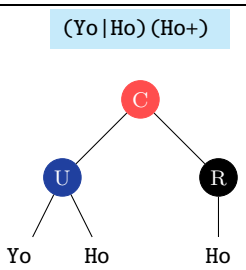


Figure 5.2: RegExTree for RegEx `(Yo|Ho)(Ho+)`. Nodes represent `Concat (C)`, `Union (U)` and `Repeat (R)` operators.

Building on Succinct (and other Data Structures). All the algorithms discussed in subsequent sections build on Succinct (Chapter 2). In particular, the algorithms assume the underlying input is a flat, unstructured file, and encoded using Succinct in order to efficiently support (a) *search* for arbitrary *m-grams*, and (b) *random access* into the file itself. Since this functionality is not unique to Succinct, our algorithms can work with a range of other data structures, including inverted indexes [165], suffix trees [201], suffix arrays [120], compressed suffix trees [11], and compressed suffix arrays [4, 82, 162, 164].

5.2 Need for Sprint

In this section, we outline the need for Sprint using a naïve black-box approach to executing RegEx queries on compressed data.

Black-box RegEx. The "black-box" approach can be summarized in three steps (see example below):

1. Construct a RegExTree;
2. Compute search results (offsets into the input file) for each leaf of the tree (token) individually.
3. Traverse the tree bottom up, generating the results at each operator node using intermediate results for left and right subtrees. Algorithms to combine intermediate results for each operator are outlined in §5.2.1 and are illustrated in Figure 5.3.

Example. Consider a query `(Yo|Ho)(Ho+)` over the input file of Figure 5.3. The black-box approach first constructs a RegExTree (Figure 5.2) and computes the offsets for individual tokens ({`Yo, Ho`}). The RegExTree is then traversed bottom-up — token results are first

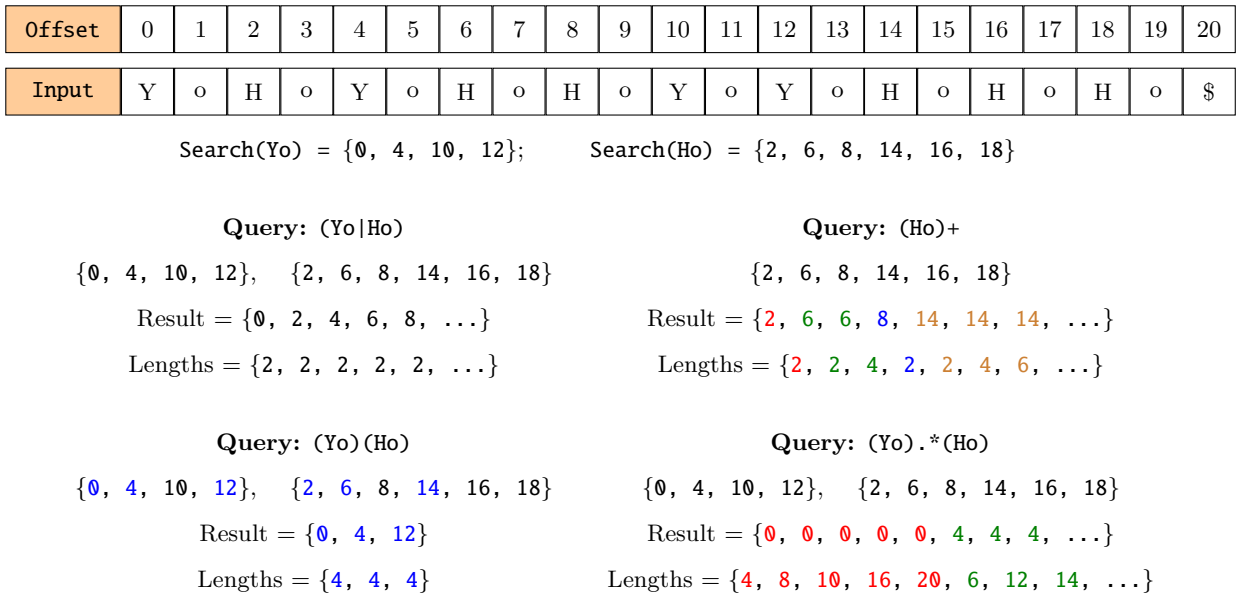


Figure 5.3: Illustration of the third step in black-box approach — executing algorithms in §5.2.1 on an example input file (the top row shows the file offsets for ease of illustration). The intermediate search results (i.e., offsets into the input file) for the 2-grams `Yo` and `Ho` are shown next. (top left) The `Union` operator outputs the set union of the offsets for the two operands. (bottom left) The `Concat` operator outputs all left operand offsets for which there exists a right operand offset satisfying $\texttt{offset}_{\texttt{right}} = \texttt{offset}_{\texttt{left}} + \texttt{length}_{\texttt{left}}$. (top right) The `Repeat` operator is similar to the `Concat` operator except for `length` admits values depending on last result. (bottom right) The `Wildcard` operator outputs all left operand offsets for which there exists a right operand offset satisfying $\texttt{offset}_{\texttt{right}} \geq \texttt{offset}_{\texttt{left}} + \texttt{length}_{\texttt{left}}$.

used to compute the result for `(Yo|Ho)` and for `(Ho)+`, as in Figure 5.3, and then combined along the `Concat` operator to get the final result `{4, 12, 14}`. Note that to combine the results across multiple operators, the length for corresponding intermediate results (*e.g.*, 2 for `(Yo|Ho)`) also needs to be tracked.

5.2.1 Black Box Algorithms

We describe the algorithms for combining the intermediate results (corresponding to the left and right subtree) for individual operators using the black-box approach[4]. We assume the input to be a flat unstructured file, where a `ResultSet` is a collection of `(offset, length)` pairs, corresponding to the offsets and the match length for the sub-RegEx rooted at a node in the RegExTree. We discuss extending these algorithms to support RegEx on semi-structured data in §5.3.9.

Union. The trivial algorithm for the `Union` operator outputs the set union of left (`L`) and right (`R`) subtree results. Trivially, the complexity of the algorithm is $O(|\texttt{L}| + |\texttt{R}|)$. Since the output cardinality is also $s_o = |\texttt{L}| + |\texttt{R}|$, the complexity of the algorithm is $O(s_o)$.

[4]We believe these algorithms to be standard, but outline them for sake of completeness of our analysis results

Algorithm 6 Concat

```
1: procedure Concat(L: ResultSet, R: ResultSet)        ▷ L, sorted by (offset + length), R sorted by offset
2:     i ← 0, j ← 0;        O ← ∅
3:     while i < L.size and j < R.size do
4:         if (L[i].offset + L[i].length = R[j].offset) then
5:             Put (L[i].offset, L[i].length + R[j].length) in O
6:             i ← i + 1, j ← j + 1
7:         else if (L[i].offset + L[i].length < R[j].offset) then
8:             i ← i + 1
9:         else
10:            j ← j + 1
11:    return O
```

Algorithm 7 Repeat

```
1: procedure Repeat(L: ResultSet)                      ▷ L, sorted by (offset + length)
2:     for i ← 0 to L.size do
3:         j ← i;        ℓ ← 0
4:         while (L[i].offset + ℓ = L[j].offset) do
5:             ℓ += L[j].length
6:             Put (L[i].offset, ℓ) in O
7:             j ← j + 1
8:     return O
```

Concat. Algorithm 6 for the `Concat` operator scans `L` and `R`, and outputs all offsets `L[i].offset` in `L` for which there exists an offset `R[j].offset` in `R` such that `R[j].offset = L[i].offset + L[i].length` indicating that the sub-RegEx corresponding to results in `R` immediately follows the one in `L`.

The algorithm maintains two pointers (each initialized to the first index of the two sets). Whenever the above condition is satisfied, the pointers are advanced to the next index for both the sets; else the pointer corresponding to the smaller offset is advanced. The algorithm terminates when one of the sets is completely scanned. Since the algorithm accesses each element in `L` and `R` at most once, the complexity is $O(|\mathtt{L}| + |\mathtt{R}|)$.

Repeat. Algorithm 7 for `Repeat` is similar to that of `Concat`; the main difference is that the `length` variable (denoted by ℓ) now depends on the number of valid repetitions.

The algorithm maintains two pointers (on the same set) and checks, in each step, whether the offset for the first pointer summed up with the current length matches the offset for the second pointer. If the condition matches, a single result is output, the length value is updated to reflect another repetition and the second pointer is advanced to check for further repetitions; otherwise, the first pointer is advanced, the length is re-initialized to zero and the second pointer is brought back to the position of the first pointer. Note that each input value corresponds to at least one output value (for single repetitions). Moreover, note that the first pointer access each element in `L` once; the second pointer may access any element

Algorithm 8 `Wildcard`

```
1: procedure WILDCARD(L: ResultSet, R: ResultSet)   ▷ L, sorted by (offset + length), R sorted by offset
2:     O ← ∅
3:     Binary search to find smallest index idx2 in R such that, L[0].offset+L[0].length ≤ R[idx2].offset
4:     for i ← idx2 to R.size do
5:         Binary search to find largest index idx1 in L such that,L[idx1].offset+L[idx1].length ≤ R[i].offset
6:         for j ← 0 to idx1 do
7:             ℓ ← (R[i].offset − L[j].offset) + R[i].length
8:             Put (L[j].offset, ℓ) in O
9:     return O
```

more than once but outputs at least one output for each access. The complexity of the algorithm is, thus, $|\texttt{L}| + |\mathcal{O}| < 2|\mathcal{O}| = 2s_o$, since $\texttt{L} \subseteq \mathcal{O}$.

Wildcard. Algorithm 8 for the `Wildcard` operator takes `L` and `R` and outputs all pairs of elements `(`ℓ`, r)` such that `r` occurs after ℓ.

The algorithm has two main ideas. First, to avoid unnecessary operations, the algorithm first picks the element in `R` that occurs after than the first element in `L` into the file — this ensures that there exists at least one element in `L` corresponds to the `Wildcard` results. Second, to find the smaller element in `L`, the algorithm performs a binary search rather than a scan. The binary search takes time $\log(|\texttt{L}|+|\texttt{R}|)$, and outputs, say x_1 results (the first idea ensures that $x_1 \neq 0$). The complexity of each step is, thus, $x_1 + \log(|\texttt{L}|+|\texttt{R}|) <= x_1 \cdot \log(|\texttt{L}|+|\texttt{R}|)$. The end-to-end complexity of the algorithm is: $(x_1+x_2+\dots)\cdot\log(|\texttt{L}|+|\texttt{R}|) = s_0 \cdot \log(|\texttt{L}|+|\texttt{R}|) \leq s_0 \cdot \log(n)$.

5.2.2 Analysis of Black-box RegEx

We now analyze the black-box approach under the standard RAM computational model [43] [5]. Specifically, we obtain the following result for the individual operator algorithms:

Lemma 1 *Given the intermediate results for the left and the right subtree as sorted arrays of size m and $n \geq m$, there exist algorithms for* `Union, Repeat, Wildcard` *and* `Concat` *operators that combine the intermediate results in time $O(s_o), O(s_o), O(s_o \log n)$ and $O(m+n)$, respectively, where s_o is the final output cardinality.*

It is known that, under the RAM computational model, the time complexity of an algorithm is lower bounded by the output size [43]. Since the output cardinality s_o is dependent on the input file and is unknown a priori, the above lemma shows that *independent of the cardinality of the results for the left and the right subtree*, the `Union, Repeat` and `Wildcard` operators

[5] While a standard for algorithmic analysis, the RAM computation model ignores effects of data caching. Nevertheless, it provides a rough estimate of the efficiency of the individual operators in the black-box approach. Our evaluation (§5.3.5) takes this limitation into account by ensuring that all data fits in memory.

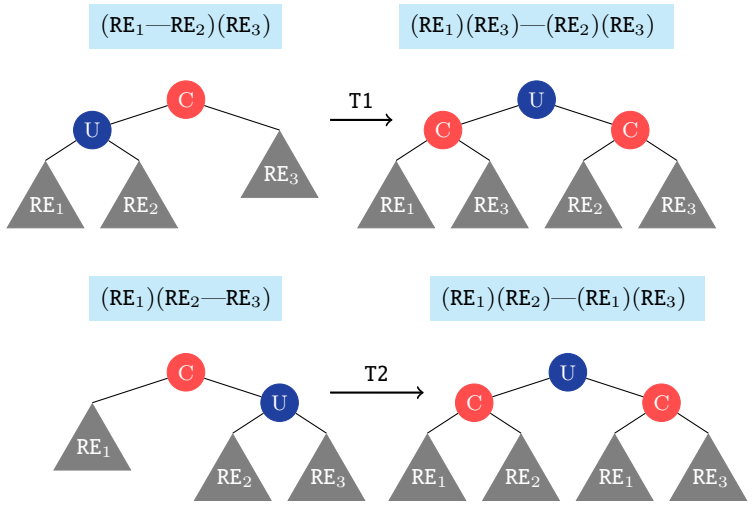


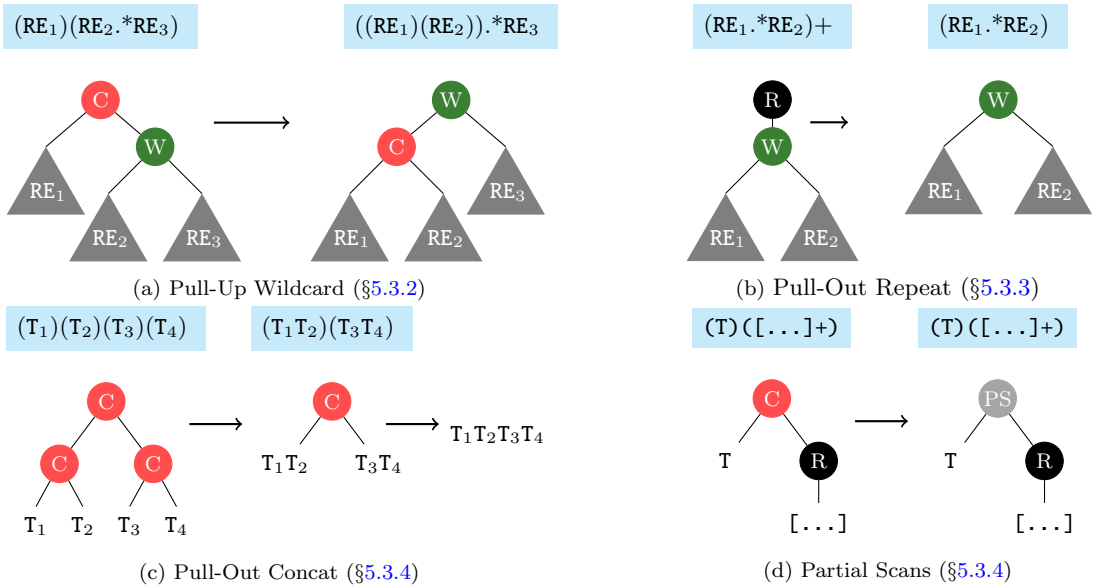
Figure 5.4: `Pull-Up Union` (§5.3.1): transformation `T1` is used if the `Union` operator is the left child, and `T2` otherwise.

combine these results in *almost optimal* time for any fixed RegExTree[6]. However, such is not the case for the `Concat` operator — the output cardinality for the `Concat` operator ($O(1)$ in the worst-case) can be arbitrarily smaller than the cardinality of results for the left or the right subtree. Thus, the `Concat` operator when operating on intermediate results of the left and the right subtree may end up performing significantly more operations than ideal — linear in the output size — making the black-box approach inefficient.

The end-to-end performance of the black-box approach depends on the time taken to construct the RegExTree, to search for leaf tokens, and to traverse up the tree combining intermediate results at nodes. In our experiments, we found that the last step is indeed the performance bottleneck (thus making Lemma 1 result more relevant). Intuitively, this is because constructing a RegExTree (scanning the RegEx once) and searching for individual tokens in index (binary search) is extremely fast. The performance of the third step, in turn, requires combining intermediate results across the operators along the RegExTree, which is significantly more complex.

Need for Sprint. Lemma 1 outlines the central problem in devising a technique for executing RegEx queries on compressed data. As shown in Figure 5.1, the performance for queries containing `Concat` operator can be arbitrarily far from optimal, and requires careful handling for efficient execution. In the following section, we outline a query re-writing technique that enables the efficient execution of queries containing `Concat` operator through simple transformations of the query RegExTree.

[6]The `Wildcard` operator requires an extra logarithmic factor in terms of the cardinality of the intermediate results.



(a) Pull-Up Wildcard (§5.3.2)

(b) Pull-Out Repeat (§5.3.3)

(c) Pull-Out Concat (§5.3.4)

(d) Partial Scans (§5.3.4)

Figure 5.5: Sprint Transformations

5.3 Sprint

We now describe Sprint, a query re-writing technique that improves upon the black-box approach using two ideas. First, it transforms a naïvely built RegExTree into one where most `Union, Wildcard` and `Repeat` operators are not the children of a `Concat` operator (§5.3.1, §5.3.2, §5.3.3). These operators are, thus, pushed up the tree and operate in a near-optimal manner as shown in Lemma 1. Second, it avoids the black-box approach for the `Concat` operator (§5.3.4). We finally show how to combine these two ideas to construct an efficient end-to-end RegEx execution engine (§5.3.5).

5.3.1 Pull-Up Union

The `Pull-Up Union` transformation attempts to transform a given RegExTree into one where `Union` operator is not a child of a `Concat` operator. The transformation is formally described in Algorithm 9, and is illustrated in Figure 5.4.

The transformation uses a simple observation that a RegEx of the form $(\texttt{RE}_1|\texttt{RE}_2)(\texttt{RE}_3)$ is equivalent to $(\texttt{RE}_1)(\texttt{RE}_3)|(\texttt{RE}_2)(\texttt{RE}_3)$, for arbitrary RegEx $\texttt{RE}_1, \texttt{RE}_2, \texttt{RE}_3$, i.e., the `Concat` operator is both left- and right-distributive [54] with the *Union* operator (see Figure 5.4 for an example). Note that if both children of the `Concat` operator are `Union` operators, the transformation needs to be applied recursively (as in Algorithm 9) since the transformation introduces new `Concat` nodes in the RegExTree; this also follows from the left- and right-distributive properties of the `Concat` operator over the `Union` operator.

Algorithm 9 `Pull-Up-Union` (**node**: `RegExTree`)

```
    /* Base case: terminate if leaf node is a token. */
 1: if node type is Token then
 2:     return
    /* Pull up unions in left and right sub-tree. */
 3: pullUpUnion(node.left)
 4: pullUpUnion(node.right)
 5: if node type is Concat then
    / *Apply transformations (recursively)*/
 6:     if node.left type is Union then
 7:         apply transformation T1 to node (Figure 5.4)
 8:     else if node.right type is Union then
 9:         apply transformation T2 to node (Figure 5.4)
10:     pullUpUnion(node.left)
11:     pullUpUnion(node.right)
12: return
```

Table 5.2: Protein Signature RegEx queries taken from the Prosite Database [174]

QueryID	Query	Protein Family
Query#1	[DE][SN]L[SAN][ACDFHKMLNQPSRTWVY][ACDGFIHKMNQPSRWVY][DE].EL	GRANINS_1
Query#2	[LIVMF][LIMN]E[LIVMCA]N[PATLIVM][KR][LIVMSTAC]	CPSASE_2
Query#3	[KRG][KR].[GSAC][KRQVA][LIVMK][WY][LIVM][KRN][LIVM][LFY][APK]	RIBOSOMAL_L16_1
Query#4	[DE]GSW.[GE].W[GA][LIVM].[FY].Y[GA]	TERPENE_SYNTHASES
Query#5	Q[LIV]HH[SA]..DG[FY]H	CAT
Query#6	[AC]GL.FPV	HISTONE_H2A
Query#7	CKPCLK.TC	CLUSTERIN_1
Query#8	Y..[HP]W[FYH][APS][DE].P.KG.[GA][FY]RC[IV][RH][IV]	BTG_1
Query#9	G[MV]ALFCGCGH	MYELIN_PLP_1
Query#10	[FYW]P[GS]N[LIVM]R[EQ]L.[NHAT]	SIGMA54_INTERACT_3

5.3.2 Pull-Up Wildcard

The `Pull-Up Wildcard` transformation attempts to ensure that the resulting RegExTree does not have a `Wildcard` operator as a child of a `Concat` operator. The transformation builds upon another simple observation that a RegEx of the form $(RE_1)(RE_2.*RE_3)$ is equivalent to $(RE_1)(RE_2).*RE_3$. Figure 5.5a illustrates this transformation on a RegExTree containing `Wildcard` as a child of the `Concat` operator. Note that no new nodes are introduced, and thus, the transformation does not need to be applied recursively.

5.3.3 Pull-Out Repeat

Unlike `Union` and `Wildcard` operators, ensuring that a `Repeat` operator is not a child of a `Concat` operator is more challenging. Sprint only partially handles this case — when the child of the `Repeat` operator is either a `Wildcard` operator or an `m-gram` token, the transformation *pulls out* the `Repeat` operator from the RegExTree. Otherwise, the subtree

Table 5.3: Text analysis RegEx queries taken from [40]; \d and \. refer to any digit (i.e.[0-9]) and to the dot ('.') character, respectively.

Query ID	Query	Description
Query#1	<script>.*</script>	HTML Scripts
Query#2	Motorola.*(XPC|MPC)([0-9])+([0-9a-z])*	Motorola PowerPC chip numbers
Query#3	William [A-Z]([a-z])+ Clinton	President Clinton's middle name
Query#4	1-\d\d\d-\d\d\d-\d\d\d\d	US Phone Numbers
Query#5	([a-z0-9_\.])+(([a-z0-9])+\.)*stanford\.edu	Stanford domain URLs.

rooted at the Repeat operator (denoted by RE+ below) is left as is.

RE with Wildcard. Note that if RE contains a Wildcard operator, the child of the Repeat operator is the Wildcard operator (due to standard precedence order). If $\texttt{RE} \equiv \texttt{RE}_1\texttt{.*RE}_2$, then it is easy to see that results for RE+ are same as that of RE, by definition of the Wildcard operator. Therefore, if the (only) child of the Repeat operator is a Wildcard operator, we simply remove the corresponding Repeat node from the RegExTree (see Figure 5.5b).

RE with m-gram token. Now consider the case when RE does not contain a Wildcard operator; since Sprint does not transform the RegExTree when RE contains either of Union or Concat operators, RE must be a token. If RE is an m-gram, the transformation exploits the observation that a Repeat operator can equivalently be represented as a Union of Concatenations. Specifically, let $\texttt{RE}^i$ represent exactly i self-concatenations of RE; that is, $\texttt{RE}^1 = \texttt{RE}$, $\texttt{RE}^2 = \texttt{(RE)(RE)}$, and so on. Then, the expression RE+ can be written as $\texttt{RE+} = (\texttt{RE}^1|\texttt{RE}^2|\texttt{RE}^3|...|\texttt{RE}^n)$, where n is the number of characters in the input file. The transformation, thus, replaces the repeat operator by a subtree composed of Union and Concat operators corresponding to the above expression.

However, naïvely doing this transformation will result in RegExTree having very large depth (due to expanding RE+ for length n, the number of characters in the input file). Indeed, in practice, there exists a small k such that $\texttt{RE}^k$ has non-zero number of occurrences while $\texttt{RE}^{k+1}$ has zero occurrences. It is therefore sufficient to expand the Repeat operator for only k terms. Furthermore, since RE is an m-gram, it suffices to perform a binary search for k — each step in the binary search looks up the index to check whether $\texttt{RE}^i$ has non-zero occurrences. This requires $\log(n)$ index lookups but is still faster than the black-box approach. The subtree rooted at the Repeat operator is thus replaced by a combination of Union and Concat operators. We then apply the transformations from §5.3.1 and §5.3.2 to ensure that Concat is not a parent of the Union or Wildcard operators.

5.3.4 Pull-Out Concat

Finally, we introduce a simple Pull-Out Concat transformation, which is executed when either of the two conditions are met. First, if both the children of a Concat operator are tokens (say, T and T'), the transformation *pulls out* the Concat operator and replaces the subtree rooted at the Concat operator with a new token TT', a longer string that is a *string concatenation* of the two children tokens (Figure 5.5c). Second, if the child of the Concat

operator is a `Repeat` operator with character class token as its child, the sub-RegEx must be of the form $(R_1)(R_2+)$. As discussed in §5.2, Sprint executes this sub-expression using *partial scans*. The transformation thus pulls out the `Concat` operator and replaces it with a partial scan (`PS`) operator (Figure 5.5d).

5.3.5 Putting it all together

We finally connect all the pieces together, and show how Sprint executes a given RegEx query. Given the query, we construct a RegExTree; we then traverse the RegExTree in a `bottom-up` fashion, applying the transformations from §5.3.1, §5.3.2 and §5.3.3 to transform the original RegExTree into one with the property that most of the `Concat` operators only have tokens or other `Concat` operators as its children. Given this new RegExTree, we again traverse the tree *bottom-up*, applying `Pull-Out Concat` transformation. Finally, we execute search for the tokens (corresponding to the leaves of the new RegExTree), and traverse the RegExTree bottom-up combining the intermediate results across the operators. Once the root of the tree is reached, the final query results are returned.

We now evaluate the performance of Sprint against popular open-source systems that support RegEx query execution.

5.3.6 Experimental Setup

Datasets and Queries. Our datasets and queries are drawn from three applications: bioinformatics [132, 75], text analytics [57, 145], and distributed computing framework pipelines [204].

For the bioinformatics application, we use the standard Pfam-A Protein dataset [71], which is 8GB in size and consists of 46 million protein sequences, each composed of 20 distinct amino-acids represented by the standard IUPAC one letter codes [92]. Typical RegEx queries on these sequences search for *protein signatures*, that are certain important regions within the sequence. We present results for 10 randomly selected protein signature RegEx queries from the Prosite [174] database (see Table 5.2).

For the text analytics application, we use a collection of 4.8 million English Wikipedia articles, constituting roughly 10GB of data for our single machine experiments, and a collection of 19.2 million Wikipedia articles ($\sim$ 10GB of data) for our distributed experiments. Unfortunately, there is no standard workload for RegEx queries in text analytics; to that end, we ran all the queries from [40], and present results for queries that output non-zero results for Wikipedia dataset (see Table 5.3). For Apache Spark [204], we use the same dataset and queries as text analytics application, but increase both the dataset size and cluster size by 4×. We provide details on the cluster used in our experiments below.

Compared Systems. We compare the performance of Sprint against several open-source systems that support RegEx— Elasticsearch [57] and MongoDB [145] for the text analytics

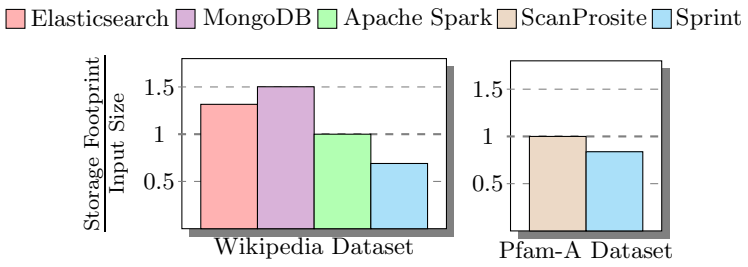
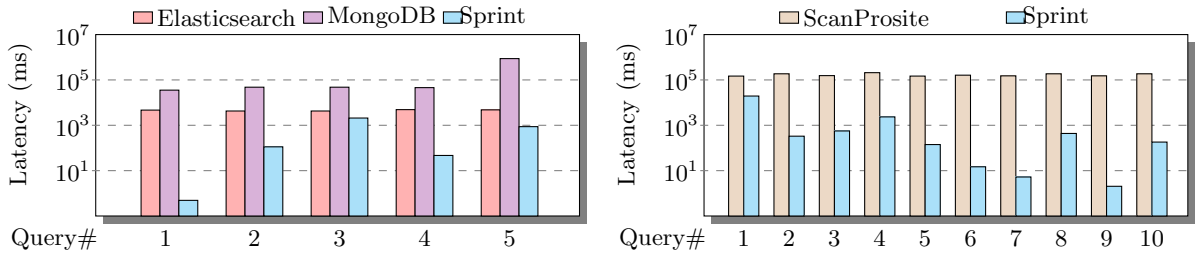


Figure 5.6: Storage footprint for different systems for the Wikipedia and Pfam-A datasets.



(a) Text Analytics

(b) Bioinformatics

Figure 5.7: Sprint executes RegEx significantly faster than popular open-source systems across various application domains.

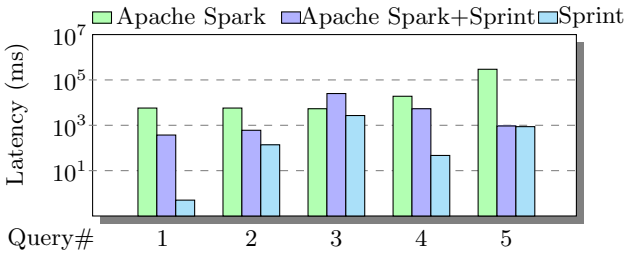


Figure 5.8: Sprint optimizations significantly speed up analytics pipelines involving RegEx queries on distributed frameworks like Apache Spark.

application, Apache Spark [204] for text analytics on a distributed computing platform, and ScanProsite [74] for the bioinformatics application.

Elasticsearch uses Lucene [124] as its underlying searching and indexing engine, and executes RegEx queries using an automaton-based approach. MongoDB indexes are not supported for text documents larger than 1KB (which is the case for some of the Wikipedia articles); thus, MongoDB executes RegEx queries using full-data scans. Apache Spark is a compute engine that can support arbitrary operations; prior to Sprint, Apache Spark used Scala's full-scan based RegEx engine to execute queries in a distributed manner. Finally, ScanProsite is a publicly available tool for executing RegEx on protein sequences using in-memory data scans.

Finally, Sprint executes RegEx queries directly on Succinct, as outlined in §5.2 and

§5.3. Figure 5.6 compares the storage overhead for the different systems. Elasticsearch and MongoDB have storage footprint of $1.3 - 1.5\times$ the input size, while Apache Spark and ScanProsite use storage exactly $1\times$ the input size. Finally, Sprint on Succinct has the lowest storage footprint of $0.6-0.8\times$ the input size for different application domains, i.e., it operates on compressed data.

The rest of the chapter focuses on latency of executing RegEx, over an Amazon EC2 r3.8xlarge instance with 244GB RAM (for bioinformatics and text analytics applications), and a cluster of 4 c3.4xlarge instances with 30GB RAM each (for distributed computing framework application). In both settings, the available RAM is large enough to fit each of the data structures completely in memory (for all systems).

5.3.7 Comparison against Existing Systems

We start by discussing the performance of Sprint against existing systems that support RegEx query execution.

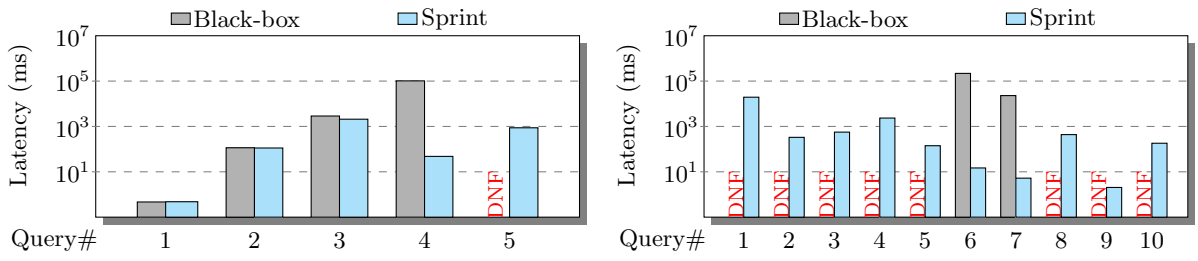
Text Analytics. Figure 5.7a summarizes the query latency results for the text analytics application. MongoDB scans through all of the documents to find matches to the RegEx, while Elasticsearch scans through all the index entries. Sprint, however, transforms the RegExTree to efficiently search for component m-grams within the RegEx, avoiding data scans as much as possible. This enables Sprint to achieve much lower query latency compared to existing systems, with benefits varying from 1–3 orders of magnitude across the evaluated queries.

Bioinformatics. The query latencies for Sprint and ScanProsite are summarized in Figure 5.7b. Sprint significantly outperforms ScanProsite, often as much as by four orders of magnitude. This is primarily because ScanProsite scans the entire data for each query (leading to similar latency across queries). Sprint, on the other hand, avoids scans and can efficiently lookup the RegEx tokens from the underlying data structure (Succinct, in this case), allowing it to find matches for the protein signatures much faster.

Distributed Computing Framework. Figure 5.8 compares the RegEx query latency for Apache Spark, with and without Sprint; the figure also shows the performance of Sprint (outside Apache Spark) for relative comparison with Figure 5.7a results. We observe that Sprint significantly speeds up Apache Spark (often by $\sim$ 1–2 orders of magnitude) by avoiding Apache Spark's full-scan based approach. For `Query#3`, however, Sprint's implementation on Apache Spark suffers from Java's GC overheads (since the intermediate results contain a large number of small objects) and Apache Spark's task startup time overheads. Sprint's standalone implementation, on the other hand, observes consistently low latency.

5.3.8 Benefits of Sprint Optimizations

We now evaluate the benefits of Sprint optimizations on top of the black-box approach. Our key observation is that when a query comprises of `Union, Repeat` and `Wildcard` operators



(a) Text Analytics (b) Bioinformatics

Figure 5.9: **Performance gains for Sprint optimizations over Black-box approach across different application domains.** Sprint achieves significant speedups for queries where Sprint transformations are applicable (`Query#4-5` for Text Analytics, all queries for Bioinformatics); queries where the transformations are not applicable or require partial scans see performance similar to the black box approach (`Query#1-3` for Text Analytics). Queries marked **DNF** did not finish within 10 minutes of execution time.

only (that execute in near-optimal time as shown in Lemma 1), Sprint optimizations do not provide benefits over the black-box approach. However, most queries (12 out of 15 in our evaluation) can benefit significantly using Sprint, sometimes by as much as two orders of magnitude. We discuss the results in depth below.

Queries for which Sprint is unnecessary. We start the discussion with queries where Sprint transformations are unnecessary (3 out of 15 queries in our evaluation). These queries either: (1) do not contain sub-optimal operators for the black-box approach (*e.g.*, `Query#1` for Wikipedia); or (2) contain character classes where both the black-box and the Sprint approaches perform partial scans (*e.g.*, `Query#2, #3` for Wikipedia). Figure 5.9 shows that Sprint has performance similar to the black-box approach for these queries.

Benefits of Sprint. For most of the queries (12 out of 15 queries in our evaluation; see Figure 5.9), Sprint approach yields significant speedup over the black-box approach. These queries have three peculiar properties that make the black-box approach inefficient. First, some of these queries (*e.g.*, `Query #1--#5, #8, #10` in Pfam) contain a large number of `Concat` operators, making the black-box approach inefficient due to Lemma 1. Second, queries that contain fewer `Concat` operators (*e.g.*, `Query #6, #7, #9` in Pfam) often have large number of occurrences for individual tokens; Lemma 1 shows that as the cardinality of results for the left and the right subtree increases, the black-box approach may get worse for the `Concat` operator. Finally, all Pfam queries as well as some Wikipedia queries (*e.g.*, `Query #4, #5`) have character classes around frequently occurring tokens, making partial data scans inefficient since a large fraction of the input needs to be scanned. Sprint overcomes these inefficiencies of the black-box approach using its transformations, leading to one to two orders of magnitude faster query execution than the black-box approach.

5.3.9 Generality of Sprint

Although our discussions so far have been restricted to flat unstructured inputs encoded using Succinct, Sprint algorithms can be adapted to more general data representations, and even several uncompressed index structures.

Sprint RegEx Execution with Other Data Structures

Recall from §5.2 and §5.3 that the Sprint query execution relies on Succinct for arbitrary token searches and random access to the input. Interestingly, Sprint leads to performance improvements even for uncompressed index structures that provide functionality similar to Succinct [40, 165, 201, 120, 11, 4, 82]. Although not originally our goal, we have implemented Sprint on top of a variety of data structures, including inverted indexes [165], suffix trees (ST) [201], suffix arrays (SA) [120], compressed suffix trees (CST) [11], and compressed suffix arrays (CSA) [4, 82, 162, 164].

Each of these data structures achieves a unique tradeoff between the storage footprint and the search latency for `m-gram` tokens. We present results for ST, SA, and CSA, since these achieve strictly better space-latency tradeoff than other data structures. CSA can achieve multiple operating points on the storage-latency tradeoff space depending on the desired compression factor; we present the results for the two extremes (termed CSA1 and CSA2).

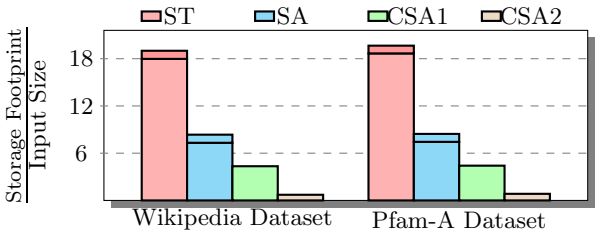
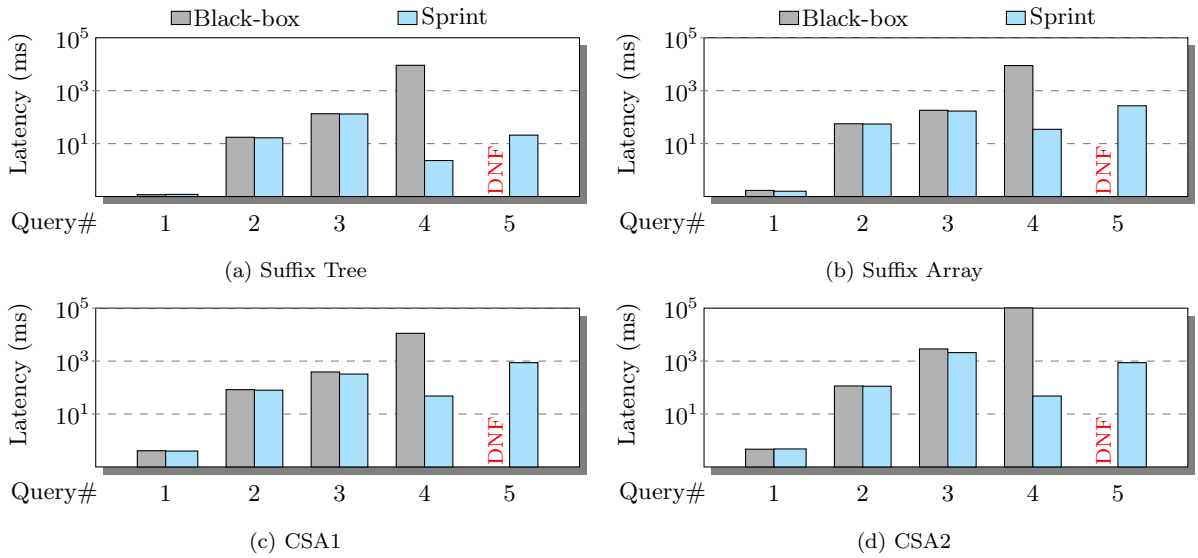


Figure 5.10: Storage footprint for different data structures for the Wikipedia and Pfam datasets. Note that for ST and SA, we store the original input as well (shown as solid fill), while CSA implicitly encodes the input.

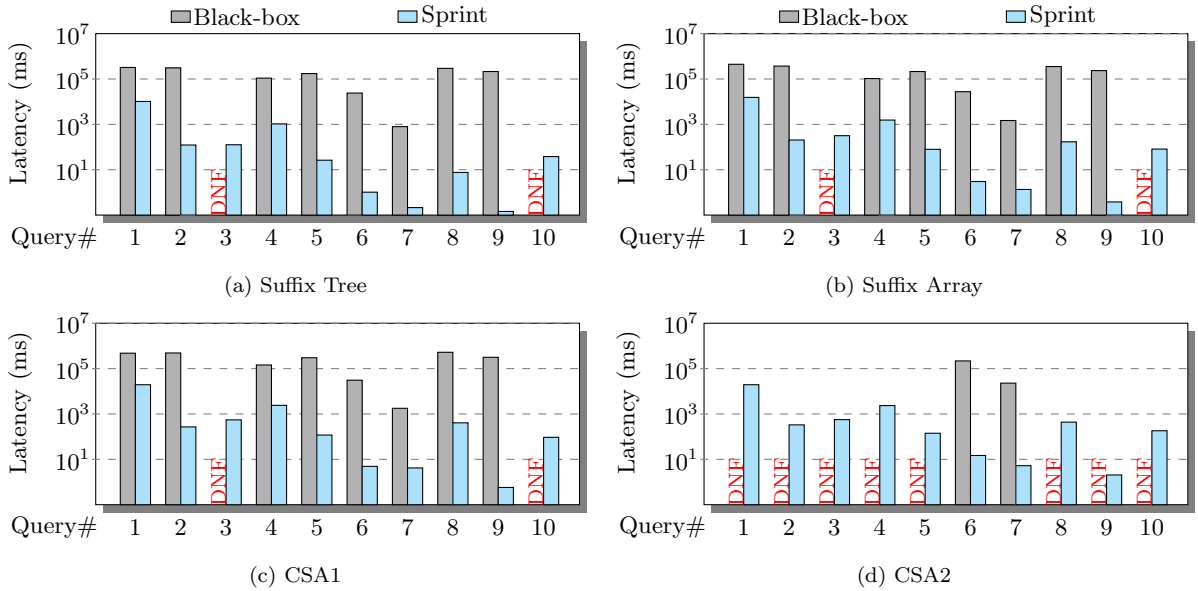
On choice of data structure. While Sprint offers performance benefits across all the evaluated data structures, the absolute performance depends on the underlying data structure. Figure 5.13 shows the performance of SA, and the two versions of CSA relative to the ST data structure; these are the same results as in Figure 5.11 and Figure 5.12, just focusing on Sprint performance and scaled by the ST latency. Interestingly, the higher storage footprint of ST often offers super-linear latency benefits when the system is not memory-constrained — ST requires $2.2\times, 4.3\times$ and $26.2\times$ higher storage than SA, CSA1 and CSA2, and offers $4.7\times, 10\times$ and $13.3\times$ lower latency on an average, respectively. Indeed, the tradeoff may be different for memory-constrained systems; we leave a through evaluation of this case for future work.



(a) Suffix Tree (b) Suffix Array

(c) CSA1 (d) CSA2

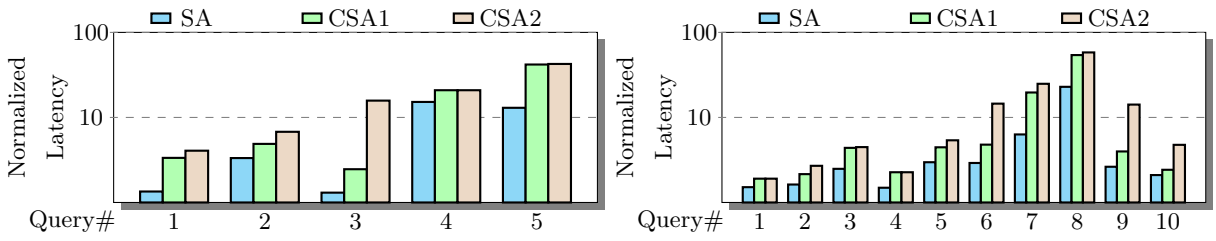
Figure 5.11: **Performance gains for Sprint optimizations over Black-box approach across different data structures for the Wikipedia dataset.** Sprint achieves significant speedups for queries where Sprint transformations are applicable (`Query#4-5`); queries where the transformations are not applicable or require partial scans see performance similar to black box approach (`Query#1-3`). Queries marked **DNF** did not finish within 10 minutes of execution time.



(a) Suffix Tree (b) Suffix Array

(c) CSA1 (d) CSA2

Figure 5.12: **Performance gains for Sprint optimizations over Black-box approach across different data structures for the Pfam-A dataset.** Since Sprint transformations are applicable for all queries, Sprint offers significantly lower latency compared to the black box approach. Queries marked **DNF** did not finish within 10 minutes of execution time.



(a) Wikipedia Dataset

(b) Pfam-A Dataset

Figure 5.13: **Comparison of Sprint latency across different data-structures.** Query latency results are normalized against Suffix Tree latency. Note that the higher storage footprint of Suffix Tree offers super-linear gains over Suffix Array and Compressed Suffix Arrays.

Semi-structured Data

For semi-structured data, we assume that the indexes above map each token to a `(documentID, offset)` pair, where the latter is the `offset` into the document where the token occurs. This allows us to adapt Sprint algorithms for flat unstructured files to semi-structured data without any change in the asymptotic complexity. We discuss extensions required for semi-structured data below.

We assume that indexes map tokens to a pair `(documentID, offset)`, where `offset` is the starting offset of the document into a flat file containing all documents. The pairs `(documentId, offset)` are sorted by `offset`s; given an offset, the corresponding `documentID` can be found via binary search.

Union. No modifications required, since each `(documentID, offset)` pair already corresponds to a valid result.

Concat. Line 4 in Algorithm 6 is modified to additionally check if both `L[i].offset` and `R[j].offset` have the same `documentID`. This ensures that two offsets are concatenated only if they belong to the same `documentID`.

Repeat. As above, Line 4 in Algorithm 7 is modified to check if both `L[i].offset` and `L[j].offset` have the same `documentID`.

Wildcard. Line 8 in Algorithm 8 is modified to insert only those results into `R` for which `L[j]` and `R[i]` have the same `documentID`. For each `R[i]`, we determine the start and end offset for the corresponding document by consulting the `(documentId, offset)` pairs; while inserting corresponding `L[j]` entries in `ROut`, we check if `L[j].offset` lies between the begin and end offsets for `R[i]`'s document.

Since we perform an additional binary search on the list of documents for each `R[i]`, this adds an additional $\log(\texttt{\#documents})$ term to the complexity, bringing the overall complexity to $s_0 \cdot (\log(\texttt{|L|+|R|}) + \log(\texttt{\#documents}))$.

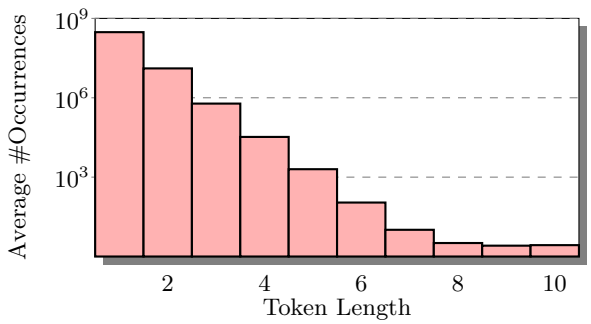


Figure 5.14: **Why Sprint works.** Variation of token frequency with token length for the Pfam-A dataset — the average number of occurrences of the tokens decrease as their length is increased.

5.3.10 Digging deeper into Sprint Performance: When & why it works

Irrespective of the underlying data structure, Sprint achieves its performance benefits by avoiding the `Concat` operator over the intermediate results altogether. This is, for instance, the case for all queries in the bioinformatics application. Besides avoiding the suboptimal `Concat` operator, Sprint achieves performance benefits due to another interesting reason. Intuitively, after the transformations are applied on the RegExTree, the leaves of the resulting RegExTree has tokens that are of length longer than the tokens in the original query. Figure 5.14 shows that, for the Pfam-A dataset, the number of occurrences (and hence, the cardinality of intermediate results) decreases exponentially as the length of the tokens increase; we see a similar trend for the Wikipedia dataset. The operators up the RegExTree, hence, operate on smaller cardinality sets leading to further improvements in the query latency.

Finally, we observe that Sprint performance varies significantly across queries. Interestingly, there is a particular parameter that allows us to explain this performance difference. It turns out that Sprint performance is proportional to the number of leaves with non-zero occurrences in the *transformed* RegExTree. Of course, it is hard to find the number of leaves with non-zero occurrences apriori since it depends on the input file. We can, however, estimate this by assuming that each leaf in the RegExTree has non-zero number of occurrences. The number of leaves are then given by the cartesian product of the sets corresponding to each token in the original RegExTree. Our evaluation suggests that in most cases (except for one query, `Query#8`), the total number of leaves computed using the cartesian product provides a good estimate for the number of leaves in the transformed RegExTree. Intuitively, this is because most of the tokens have at least a few occurrences in large datasets.

5.4 Related Work

We compare and contrast Sprint against the two traditional approaches for RegEx queries.

Index-based approaches. There are a multitude of techniques both for indexing and for using indexes. On the indexing front, note that tokens in RegEx by nature are not linguistically meaningful, making traditional indexing techniques (*e.g.*, inverted indexes) that use English words or other linguistic constructs [165] as keys less useful. As a result, specialized indexes for RegEx have been designed — m-gram indexes [40, 159], full-text indexes [124], and tree-based indexes [34, 201, 11], among others.

How these indexes are used to execute RegEx typically depends on the underlying indexing technique. However, at a high-level, there are two possible approaches. First, using indexes as a mechanism to filter the documents to be scanned [40]; and second, executing the entire RegEx using indexes (the black-box approach from §5.2). The first approach is extremely fast when the selectivity of indexed tokens is high, that is, filtering results in very few documents to be scanned. However, such is often not the case (*e.g.*, all Pfam-A queries), leading to full data scans. Sprint improves the state-of-the-art for both approaches, by avoiding full-data scans as well as using optimizations to speed up the black-box approach.

Scan-based approaches, and why are index-based approaches not used in practice? Most popular open-source data stores that support RegEx queries [57, 145] resort to data scans rather than using index based techniques. We believe this is for two reasons: (i) the storage overhead of indexes specialized for RegEx queries [40]; and (ii) index-based techniques do not offer latency gains over data scans (even in our evaluation, compare results for black-box approach with scan-based approaches in Figure 5.1). Indexes thus use more storage while providing little or no latency benefits.

However, recent research has shown that the storage overhead of indexes can be reduced down to no more than the input size without asymptotic increase in query latency (Chapter 2, [82]), thus motivating us to revisit index-based approaches. Moreover, Sprint leads to orders of magnitude speed up over the scan-based approaches for most of the evaluated queries. Sprint, when operating on Succinct, resolves both the above issues with index-based approaches making them an interesting choice for executing RegEx queries.

5.5 Summary

Motivated by new challenges due to growth in data sizes, this chapter revisits the problem of efficient RegEx query execution — a powerful primitive for applications ranging from text analytics to distributed data analytics pipelines in machine learning. We present Sprint — a query execution engine that builds upon recent advances in compressed data structures to enable RegEx query execution directly over compressed data. Evaluation of Sprint against popular open-source data stores shows that Sprint leads to significant speed ups in RegEx query execution, sometimes by $2 - 3$ orders of magnitude.

Chapter 6

Conclusions and Future Work

In this dissertation, we explored the problem of designing data stores that strive for three goals: handling massive volumes of data (*scale*), supporting rich query semantics (*functionality*) and ensuring low latency and high throughput for such queries (*performance*). We argued for a data structure and algorithm driven solutions to address the systems challenges entailed in achieving these goals.

Our design philosophy culminated in Succinct, a distributed data store that takes a radically new approach to tackle these challenges — by enabling a wide range of queries directly on a compressed representation of data. Succinct allows applications to serve sophisticated queries in memory for an order of magnitude larger datasets than traditional approaches. As a result, applications are able to achieve performance at scale, without compromising on functionality.

To address the issue of dynamism in query workloads — arising out of skew in query distributions, and variation of skew over time — we augmented Succinct compression techniques in BlowFish. In contrast to traditional replication-based techniques that expose a coarse-grained tradeoff between storage and performance, BlowFish provides applications the flexibility to trade-off storage footprint for performance (and vice versa) in a fine-grained manner, just enough to meet the requirements for data under skew. BlowFish achieves this through a fine-grained, dynamic adaptation of Succinct's compression factor, using a new data structure called Layered Sampled Array. Interestingly, the unique storage-performance tradeoff enabled by BlowFish allowed us to explore several classical systems problems through a new "lens".

Finally, we explored how we can enable even richer query semantics on compressed data to meet the demands of increasingly sophisticated cloud applications. We pursued this in two specific directions: supporting graph queries, and regular expression queries. The first of these led to the design of ZipG, a memory efficient graph store for interactive queries. ZipG employed a novel graph layout that is amenable to compression via Succinct, while supporting expressive graph queries from a wide range of real-world workloads efficiently, directly on the compressed graph representation. The second led to the design of Sprint, a query re-writing technique that enables efficient execution of regular expression queries on compressed data

by exploiting Succinct's performance characteristics for different operations. This allowed Sprint to execute regular expression queries often as much as two orders of magnitude faster than traditional approaches.

The techniques developed for this dissertation have been incorporated into an open source system stack [182, 180]. We have also integrated Succinct and Sprint techniques into Apache Spark, where users can execute several queries directly on compressed RDDs [183, 5].

6.1 Future Work

We next discuss problems in both in system and algorithm design that we leave open in this dissertation.

Improving space and time complexity for compression. While we have made several optimizations to Succinct's compression algorithm, the current bottleneck in the process is the suffix array construction. The we rely on the current state-of-the-art approach [131], which has an $\mathcal{O}(n \log n)$ time complexity and employs $5n + O(1)$ memory. A $5\times$ overhead in memory and a super-linear time complexity can be prohibitive in some deployments. Recent advancements in parallel [101] and external suffix array construction [94] provide promising alternatives, but these approaches tradeoff memory overheads for construction time, and vice versa. As such, exploring the design of memory and time-efficient suffix array construction is an interesting direction for future research.

Improving sequential throughput. Recall from §2.5.5 that Succinct achieves a throughput of roughly 13Mbps per core; the throughput increases linearly with number of threads and/or cores. Succinct effectively trades off high sequential throughput to achieve high throughput for short reads and for search queries using a small memory footprint. However, analytics applications [204, 49] that could benefit from Succinct's efficient search and random access primitives, often also require high sequential read throughput. There is potential for interesting research in designing new algorithms to improve Succinct's decompression throughput.

A simpler alternative might be to maintain an *additional* uncompressed data copy, or one compressed using an approach optimized for sequential throughput (*e.g.*, Snappy [176] or LZ4 [117]). This approach would tradeoff some amount of storage for achieving high throughput for sequential scans as well as random access and search queries. Again, results from Figure 2.11 (§2.5) suggest that Succinct would still push 5-5.5$\times$ more data than popular open-source systems with similar functionality.

Efficient SQL semantics on compressed data. While Succinct, along with extensions in ZipG and Sprint, supports a wide gamut of queries on compressed data, supporting SQL queries on Succinct compressed representation is an open problem. Our interactions with the open source community indicates there is sufficient need for supporting SQL semantics, but without the high storage overheads of traditional relational databases. While our preliminary analysis indicates that selection and projection primitives can be efficiently supported in Suc-

cinct, the join primitive is relatively inefficient, as outlined in ZipG (Figure 4.13, Chapter 4). Designing efficient distributed join algorithms on Succinct presents interesting algorithmic as well as system design challenges.

Approximate/statistical queries on compressed data. While Sprint enables efficient execution of RegEx queries on compressed data, today's text analytics applications often require support for approximate queries, such as fuzzy searches (*e.g.*, search for terms similar to a given term, based on Levenshtein distance [107]) and proximity queries (*e.g.*, search for terms within bounded distance of a given term, based on some distance metric). While it is possible to use additional indexes like Apache Lucene [124] in conjunction with Succinct, an interesting direction for future work would be to provide such functionality directly on Succinct data structures, or on compressed data in general.

Update Efficiency and Strong Consistency Semantics. The Succinct stack employs a multi-store design to efficiently absorb updates, while merging these updates with the compressed representation in batches in the background. Since Succinct targets read-heavy workloads, such our approach is sufficient for low volumes of writes. However, supporting high volume writes introduces challenges along two dimensions.

First, if the fraction of writes exceeds the rate at which Succinct can compress new data, a majority of the data would now be uncompressed, negating some of the benefits of the Succinct stack. Possible directions to explore in resolving this challenge include improving the time complexity for Succinct compression as discussed above, and supporting in-place updates in Succinct data structures — a far more daunting task.

Second, maintaining large volumes of data across multiple stores introduces challenges in maintaining strong consistency semantics, when data items may have multiple versions across multiple stores. Systems that employ similar multi-store approaches (*e.g.*, columnar stores [1, 181, 103]) typically provide snapshot isolation [22] guarantees. Efficient support for stronger consistency semantics, such as serializability [23], would not only benefit Succinct, but multi-store architectures in general.

We hope that open access to our techniques will help drive both academic and open source community to develop novel solutions to these challenges.

Appendix A

Succinct Data Structures

We describe the four arrays used in Succinct, show how we achieve a *compressed* representation for each of the arrays and how these arrays are used for performing queries directly on a compressed representation of the data.

AoS. AoS stores the set of suffixes in a file in lexicographically sorted order, with AoS[i] storing the i^{th} lexicographically smallest suffix.

AoS2Input. AoS2Input maps the suffixes in AoS to corresponding locations in the input. That is, AoS2Input[i] stores the location of AoS[i] in the input file.

Input2AoS. Input2AoS maps locations in the input file to indexes into AoS2Input that stores these locations. That is, Input2AoS[loc] stores an index idx such that AoS2Input[idx] = loc.

NextCharIdx. NextCharIdx`[i]` stores the AoS2Input index that stores AoS2Input[i]+1.

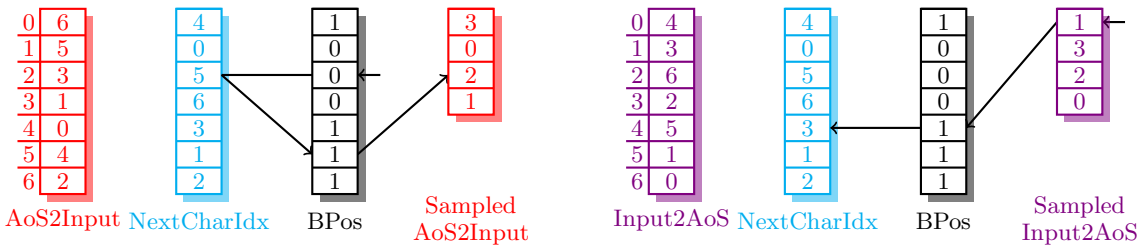
A.1 Compression

Suppose input file contains n ASCII characters, and is of size $8n$ bits. Since there are n suffixes varying from length 1 to n, AoS stores $0.5n(n+1)$ characters requiring $4n(n+1)$ bits. Each of AoS2Input, Input2AoS and NextCharIdx require $\lceil \log n \rceil$ bits for each entry, leading to a total space of $4n(n+1)+3n\lceil \log n \rceil$ bits. These arrays contain a lot of redundancy since the size of input file is just $8n$.

A.1.1 Compressing AoS2Input and Input2AoS

We now describe the details pertaining to the compression of the AoS2Input and Input2AoS arrays.

Sampling. Succinct employs *sampling* to reduce the storage footprint of the AoS2Input and Input2AoS arrays. Various sampling techniques have been used in theory [81, 82, 162,



(a) Compressing AoS2Input (b) Compressing Input2AoS

Figure A.1: Lookups on AoS2Input and Input2AoS. (a) To find AoS2Input[2], note that BPos[2]=0; following NextCharIdx[2]=5, find that BPos[5]=1, i.e., the value is sampled. rank(5)=2 gives us the corresponding index into the SampledAoS2Input. Multiplying SampledAoS2Input[2]=2 with $\alpha = 2$ and subtracting the number of NextCharIdx hops gives us AoS2Input[2]=2×2-1=3. (b) To lookup Input2AoS[1], we find the smallest multiple of $\alpha = 2$ less than loc=1, i.e., loc1=0. SampledAoS2Input[0] gives us the SampledAoS2Input index corresponding to loc1; we translate the SampledAoS2Input index to the corresponding AoS2Input index from select(1)=4. Finally, we follow loc-loc1=1 NextCharIdx pointers to get the required value Input2AoS[1]=3.

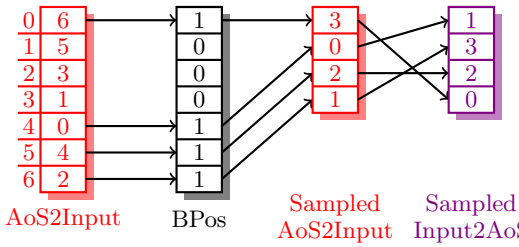


Figure A.2: **Sampling AoS2Input and Input2AoS:** We sample AoS values that are multiples of α ($\alpha = 2$ in this example) and mark the positions in BPos. Note that we store each sampled value val as val/α in SampledAoS2Input. SampledInput2AoS maps the sampled locations in the input file to indexes into SampledAoS2Input that store these locations.

164, 163]. Our implementation samples and stores all AoS2Input values that are a multiple of a configurable integer parameter α, with default being $\lceil \log n \rceil$. Each sampled value val is stored as val/α in an array called the SampledAoS2Input, leading to a more space-efficient representation. Additionally, we store a bitmap BPos that marks the positions in AoS2Input where the values are sampled. Since AoS2Input stores locations of suffixes in input file, SampledAoS2Input values correspond to sampled locations. Succinct's compressed version of Input2AoS maps the sampled locations in the input file to indexes into SampledAoS2Input that store these locations; these indexes are stored in the SampledInput2AoS array. Observe that the SampledInput2AoS is simply the *inverse mapping* of the SampledAoS2Input array. Figure A.2 shows an example of sampling for the AoS2Input and Input2AoS arrays.

Rank and Select Data Structures. In order to efficiently translate an index into a sampled array to the corresponding index into the unsampled array (and vice versa), we

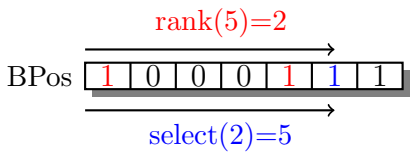


Figure A.3: **Rank and Select Data Structures:** rank(i) counts the number of 1's before i, while select(i) gives the position of the $(i+1)^{th}$ 1 in the bitmap.

store *rank* and *select* data structures for the BPos bitmap. The rank data structure takes an index idx as an input and returns the number of 1's before idx in the bitmap, whereas the select data structure takes integer i as an input and returns the location of $(i+1)^{th}$ set bit in the bitmap (see Figure A.3). Rank and Select are one of the most well-researched data structures [160, 196, 135, 205]. We implemented several of the known algorithms and found the one in [205] to perform the best in practice; we use this implementation in Succinct. In order further reduce storage overheads, we compress the bitmap itself using entropy compression.

Figure A.1 illustrates how lookups can be performed on compressed versions of AoS2Input and Input2AoS through examples. We store only the sampled versions of each of the two arrays, along with the BPos bitmap and associated rank/select data structures. The NextCharIdx array along with BPos enables looking up unsampled values in the two sampled arrays.

Looking up AoS2Input values. Looking up values in AoS2Input (Algorithm 10) can be broken down into three steps:

1. Follow NextCharIdx pointers until we reach an index idx1 in AoS2Input where the value is sampled; count the #hops it took to get there.
2. Translate the index into AoS2Input to the corresponding index into SampledAoS2Input by consulting the rank data structure (i.e., rank(idx1)).
3. Multiply the corresponding SampledAoS2Input value with α and subtract the #hops to get the AoS2Input value.

To see why Step 1 works, note that NextCharIdx[2] in Figure A.1a tells us where AoS2Input[2]+1=4 is stored; or, NextCharIdx[i] tells us where AoS2Input[i]+1 is stored in AoS2Input. Hence, to find an unsampled AoS2Input value, we check if AoS2Input at NextCharIdx[i] is sampled by consulting the BPos array. If yes, we know the value AoS2Input[i]+1 and hence, AoS2Input[i]. This process can be repeated until we find a sampled value.

Note that the SampledAoS2Input values must be multiplied by α (Step 3) to get back the corresponding AoS2Input value. Once we obtain the sampled value, we simply subtract the number of additional hops we took to reach the sampled value to get the required AoS2Input value. See Figure A.1a for an example.

Algorithm 10 Algorithm for lookupAoS2Input(i).

```
1: #hops ← 0
2: idx1 ← i
3: While !BPos[idx1]
4:     idx1 ← lookupNextCharIdx[idx1]
5:     #hops += 1
6: #BitsSet ← rank(BPos, idx1)
7: Val ← SampledAoS2Input[#BitsSet]
8: Return Val ×α - #hops
```

Looking up Input2AoS values. Similar to lookups on AoS2Input, lookups on Input2AoS can also be viewed as three steps:

1. Find the largest multiple of α loc1 smaller than or equal to the location loc being looked up; obtain the SampledInput2AoS value idx1 at loc1.

2. Translate this index into SampledAoS2Input to the corresponding index into AoS2Input by consulting the select data structure (i.e., select(idx1).

3. Follow (loc-loc1) NextCharIdx pointers to get the required Input2AoS value.

Since Input2AoS and SampledInput2AoS are inverse mappings of AoS2Input and SampledAoS2Input respectively, observe that the algorithm for looking up Input2AoS values is the *inverse* of the AoS2Input lookup algorithm.

Step 1 exploits the fact that given a location loc in the file, we know that the largest multiple of α that is smaller than or equal to loc (say, loc1) must be an AoS2Input value that is sampled. The corresponding SampledInput2AoS value would give us the SampledAoS2Input index for the sampled value.

Once we obtain the corresponding index into AoS2Input (Step 2), we observe that we would have to follow (loc-loc1) NextCharIdx pointers to get from the sampled AoS2Input value to the location which we started with (i.e., our query for the Input2AoS lookup algorithm). Therefore, it suffices to follow as many NextCharIdx pointer to get the corresponding AoS2Input index (Step 3), which would give us the required Input2AoS value. Figure A.1b illustrates the lookup algorithm with an example.

Algorithm 11 Algorithm for lookupAoS2Input(loc).

```
1: loc1 ← α × ⌊i/α⌋
2: idx1 ← lookupInput2AoS[⌊i/α⌋]
3: idx ← select(BPos, idx1)
4: For i=1 to loc-loc1
5:     idx ← lookupNextCharIdx[idx]
6: Return idx
```

A.1.2 Compressing AoS

The largest of the four arrays is AoS. The redundancy in AoS is best understood by observing that AoS[4] and AoS[3] in Figure 2.3 (Chapter 2) overlap at "anana$". Indeed, the second character of AoS[4] is the first character of AoS`[3]`. Observe that interestingly, the value of NextCharIdx[4] is 3. This is not a coincidence; intuitively, since AoS2Input stores locations of suffixes in the input, AoS2Input[4] and AoS2Input[4]+1 are the locations of the first and the second characters of AoS[4]. Since NextCharIdx[4] tells us where AoS2Input[4]+1 is stored into AoS2Input, the first character of AoS[3]=AoS[NextCharIdx[4]] is the second character of AoS[4]. It follows that we only need to store the first character for each index of AoS; the remaining characters can be computed on the fly using NextCharIdx. In fact, we can do even better — since AoS stores suffixes in sorted order, all we need is each unique character in the input file and the first index into AoS at which the suffix starts with the character. This reduces the size of AoS from $4n(n+1)$ bits to $512\lceil \log n \rceil$ bits.

Succinct representation of AoS stores: (a) all unique characters in the input file in sorted order; and (b) for each character, first AoS index with suffix starting with that character.

Looking up AoS values. Let characters be the sorted array of unique characters, and char-indexes be the corresponding array of indexes. Observe that binary searching for an index idx on char-indexes yields the index of the *first* suffix that shares the same starting character as AoS[i]; consulting the characters array gives us the first character. As described above, the first character of AoS[NextCharIdx[idx]] is the second character of AoS[idx]. Therefore, to get the second character of AoS[idx], we simply determine idx1 = NextCharIdx[idx] and determine the first character of AoS[idx1]. If we repeat this process len times, we obtain the first len bytes of AoS[idx]. Algorithm 12 summarizes this process.

Algorithm 12 Algorithm for lookupAoS(idx, len).

```
1: str ← NULL
2: idx1 ← idx
3: If i=0 to len-1
4:     idx2 = BinSearch(char-indexes, idx1)
5:     str += characters[idx2]
6:     idx1 ← lookupNPA[idx1]
7: Return str
```

A.1.3 Compressing NextCharIdx

To compress NextCharIdx, Succinct exploits two properties identified in theory literature [82, 81, 162, 164, 163]. These properties are based on a two-dimensional representation of NextCharIdx, where columns are indexed by all unique characters and rows are indexed by all unique $t-$length strings, both in sorted order. The value NextCharIdx[idx] belongs to cell (rowID, columnID) if the corresponding suffix at AoS[idx] starts with character columnID, followed by string rowID.

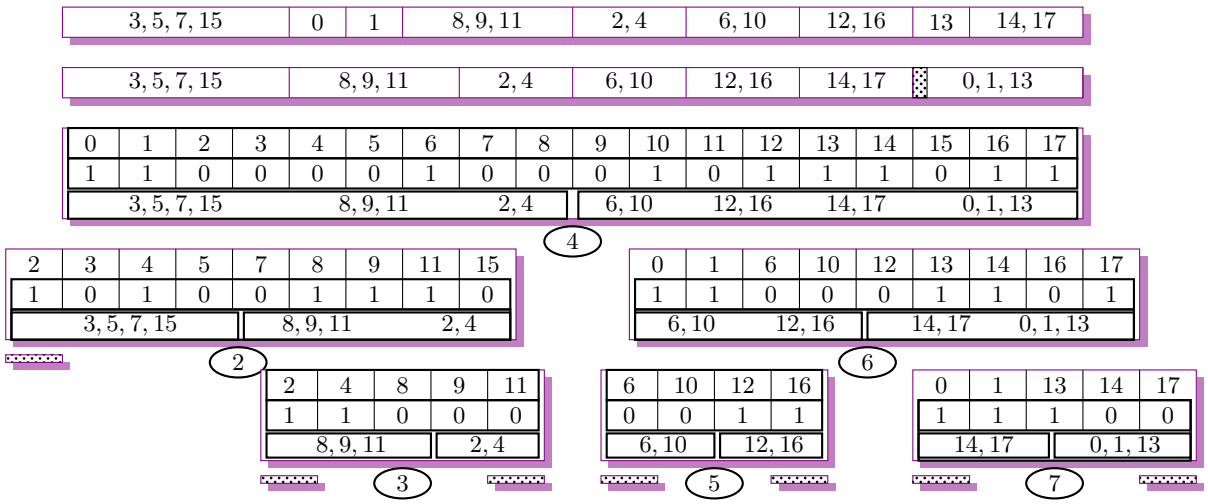


Figure A.4: Example for constructing the tree for each row of two-dimensional representation of NextCharIdx.

Property1: NextCharIdx values in any column form an *increasing* sequence of integers.

Property2: NextCharIdx values in any row form a *contiguous* sequence of integers.

Skewed Wavelet Trees We use the same two-dimensional representation of uncompressed NextCharIdx with default value of t set to 3. However, our compressed representation differs from previous ones in terms of a number of optimizations. In particular, the NextCharIdx compression techniques in [81, 82, 162, 164, 163] were focused on providing theoretical guarantees; our implementation uses subtle changes to trade-off theoretical guarantees in a few corner cases to achieve lower memory footprint and query latency.

Recall that the NextCharIdx representation consists of as many rows as the number of unique $t-$length substrings in input file. Each row contains a contiguous (not necessarily sorted) sequence of integers (see Figure A.4, top row) distributed across multiple cells. To start with, given an index idx, it is easy to find the row, the cell and the offset into the cell where NextCharIdx[idx] is stored by storing a few small arrays.

Succinct compresses each row independently in a manner that allows looking up the NextCharIdx value at a given offset into a cell. To achieve this, we construct a binary tree over the cells (the leaves of the tree correspond to cells in the row), partitioning the cells at each level such that the number of values in the cells assigned to the left children and to the right children are as close as possible. We use simple heuristics to identify cells that contain very few values (cells $1, 2$ and 7 in top row each contain a single value) and store these separately; all values in separated cells are put in a dummy cell (last cell in second row of Figure A.4). By separating sparse cells and by partitioning cells based on number

of values, we attempt to reduce the height of the tree and to ensure that cells with a large number of values have lower depth. These lead to significant reduction in memory footprint *and* latency, at the cost of theoretical guarantees in some corner cases.

Next we construct a bit array for each node of the binary tree as follows. Let S be the set of values contained in the cells being partitioned at the root. We create an array of —S— bits; the i^{th} bit of this array is set if the i^{th} largest value in S is contained in a cell assigned to the right child, and unset otherwise. See Figure A.4 and consider the left child of the root of the tree. The root is assigned the first three cells and $\mathtt{S} = \{3, 5, 7, 15, 11, 8, 9, 2, 4\}$; the sorted version of S is $\{2, 3, 4, 5, 7, 8, 9, 11, 15\}$ with values assigned to the right child being $\{2, 4, 8, 9, 11\}$, precisely the set bits in the bit array stored at that root.

In addition to the values in separated cells, Succinct stores (a) cell identifiers at which partitioning occurs at each node in the binary tree (circled values in Figure A.4) and (b) a compressed representation of the bit arrays at each node for each level of the tree, together with *rank* and *select* data structures.

Now suppose we want to locate the NextCharIdx value at the second offset in the second cell, which is equal to 9. We first locate the cell by traversing down the tree, comparing the identifier stored at the node to the identifier of the cell being located until we hit a leaf. Once the cell is located in the tree, we start traversing up the tree. At each level, we check if the current node is the left child or the right child of the parent. If it is the right child, we update offset to be the index into parent's bit array where the offset-th 1 (and 0 if it is the left child) lies. Note that these translate into select operations on the bitmap corresponding to the node; we represent searching for the position of the i^{th} 1 as select1, and the i^{th} 0 as select0. Intuitively, offset maintains the order of the desired NextCharIdx value among (the sorted) set of values at the binary tree node. **Since the root of the tree contains the set of contiguous integers (Property 2)**, the final value of offset gives us the desired integer. For our example, offset = 2 at the leaf. Since the second cell is the left child of the parent, we find the location of second '0' in parent's bit array and set offset = 4. At the next level, we find the location of fourth '1' in parent's bit array since the current node is the right child and set offset = 7. At the root, we find the location of the seventh '0', giving us offset = 9 as desired.

A.2 Query Algorithms

We described how lookups can be performed on the compressed representations of AoS, AoS2Input and Input2AoS using lookups on the NextCharIdx array in §A.1.1); we now describe how we can perform search, count and extract using the compressed arrays.

A.2.1 Random access in Succinct

The extract operation forms the basis of random access in Succinct. In order to extract len bytes starting at offset off in the *uncompressed* input file, we first lookup Input2AoS[off] to

obtain the index idx where AoS2Input[idx]=off. Once we have this index, we simply need to obtain the first len bytes of the suffix at AoS[idx], as described in Algorithm 12.

Algorithm 13 Algorithm for `extract(off, len)`.

```
1: idx ← lookupInput2AoS(off)
2: str ← lookupAoS (idx, len)
3: Return str
```

A.2.2 Counting and Searching

It is simple to see that two binary searches over the AoS array for an input string would give us the first and the last suffix that start with a given input string. If the indexes of these suffixes be idx1 and idx2 respectively, then the count of the string occurrences is simply (idx2-idx1+1), and the AoS2Input values {AoS2Input[idx1], ..., AoS2Input[idx2]} would give us the locations of each of these occurrences (i.e., the search results).

Succinct, however, exploits the two dimensional representation of NextCharIdx and the information contained within it obtain the two indexes idx1 and idx2 more efficiently. Consider the input string "banana"; recall from §A.1.3 that the cell=⟨n, ana⟩ contains all NextCharIdx[idx] values for which the first character of the suffix at AoS [idx] is "a" and the following three characters are "ana".

Algorithm 14 Algorithm for `findRange(str)`.

```
1: len ← length(str)
2: cell ← ⟨str[len-t-1], str[(len-t)...(len-1)]⟩
3: (idx1, idx2) ← (firstIdx(cell), lastIdx(cell))
4: For i=len-1 to 0
5:     cell ← ⟨str[i-t-1], str[(i-t)...(i-1)]⟩
6:     idx01 ← BinSearch(cell, idx1)
7:     idx02 ← BinSearch(cell, idx2)
8:     (idx1, idx2) ← (idx01, idx02)
9: Return (idx1, idx2)
```

Algorithm 15 Algorithm for `count(str)`.

```
1: (idx1, idx2) ← findRange(str)
2: cnt ← idx2 - idx1 + 1
3: Return cnt
```

Our algorithm aggressively exploits the above interpretation. In particular, the algorithm first finds indexes idx1 and idx2 for NextCharIdx values that belong to cell = ⟨n, **ana**⟩. In the next step, the algorithm looks in cell = ⟨a, nan⟩ and performs a binary search over the NextCharIdx values in the cell (they are sorted, not necessarily contiguous) and finds

Algorithm 16 Algorithm for `search(str)`.

```
1: res ← {}
2: (idx1, idx2) ← findRange(str)
3: For i=idx1 to idx2
4:     loc ← lookupAoS2Input[i]
5:     Insert loc in res
6: Return res
```

indexes idx01 and idx02 such that NextCharIdx[idx01] = idx1 and NextCharIdx[idx02] = idx2. Intuitively, after this round, idx01 and tt idx02 are indexes into AoS for which suffixes start with a and are followed by nana. The final binary search is done in cell = $\langle$b, ana$\rangle$ and the results are the desired indexes. The algorithm for finding idx1 and idx2 is depicted in Algorithm 14, while count and search are summarized in Algorithms 15 and 16.

Bibliography

[1] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. "Integrating Compression and Execution in Column-Oriented Database Systems". In: *ACM International Conference on Management of Data (SIGMOD)*. 2006.

[2] *Accelerating Text Analytics Queries on Reconfigurable Platforms*. URL: http://ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-atasu.pdf.

[3] Gediminas Adomavicius and Alexander Tuzhilin. "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions". In: *IEEE Transactions on Knowledge & Data Engineering* 6 (2005), pp. 734–749.

[4] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. "Succinct: Enabling Queries on Compressed Data". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015.

[5] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. *Succinct Spark from AMPLab: Queries on Compressed RDDs*. URL: https://bit.ly/1LpVOxt.

[6] Amr Ahmed et al. "Distributed Large-scale Natural Graph Factorization". In: *ACM International Conference on World Wide Web (WWW)*. 2013.

[7] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison Wesley, 1986.

[8] Ashok Anand et al. "Cheap and Large CAMs for High Performance Data-Intensive Networked Systems." In: *NSDI*. 2010.

[9] Ganesh Ananthanarayanan et al. "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters". In: *ACM European Conference on Computer Systems (EuroSys)*. 2011.

[10] Michael P Andersen and David E. Culler. "BTrDB: Optimizing Storage System Design for Timeseries Processing". In: *USENIX FAST*. 2016.

[11] Aoe, Jun-ichi and Morimoto, Katsushi and Sato, Takashi. "An Efficient Implementation of Trie Structures". In: *Software: Practice and Experience* (1992).

[12] *Apache Hive*. https://hive.apache.org.

[13] *Apache Impala*. https://impala.apache.org/.

[14] *Apache Kafka*. URL: https://kafka.apache.org.

[15] Timothy G. Armstrong et al. "LinkBench: A Database Benchmark Based on the Facebook Social Graph". In: *ACM SIGMOD*. 2013.

[16] Berk Atikoglu et al. "Workload Analysis of a Large-scale Key-value Store". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 40. 1. 2012, pp. 53–64.

[17] G. Bagan et al. "Generating Flexible Workloads for Graph Databases". In: *Proceedings of the VLDB Endowment (PVLDB)* 9.13 (2016), pp. 1457–1460.

[18] Pablo Barceló Baeza. "Querying Graph Databases". In: *ACM Symposium on Principles of Database Systems (PODS)*. 2013, pp. 175–188.

[19] Pablo Barceló Baeza, Miguel Romero, and Moshe Y. Vardi. "Semantic Acyclicity on Graph Databases". In: *ACM Symposium on Principles of Database Systems (PODS)*. 2013.

[20] Barceló, Pablo and Libkin, Leonid and Reutter, Juan L. "Querying Graph Patterns". In: *ACM Symposium on Principles of Database Systems (PODS)*. 2011.

[21] Doug Beaver et al. "Finding a Needle in Haystack: Facebook's Photo Storage". In: *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2010.

[22] Hal Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. 1995, pp. 1–10.

[23] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. "Concurrency control and recovery in database systems". In: (1987).

[24] Bharat, Krishna and Broder, Andrei and Henzinger, Monika and Kumar, Puneet and Venkatasubramanian, Suresh. "The connectivity server: Fast access to linkage information on the web". In: *Computer networks and ISDN Systems* 30 (1998).

[25] *Bing*. URL: https://bing.com.

[26] Philip Bohannon et al. "Automatic Web-scale Information Extraction". In: *ACM International Conference on Management of Data (SIGMOD)*. 2012.

[27] P. Boldi and S. Vigna. "The Webgraph Framework I: Compression Techniques". In: *ACM WWW*. 2004.

[28] Falk Brauer et al. "Enabling Information Extraction by Inference of Regular Expressions from Sample Entities". In: *ACM International Conference on Information and Knowledge Management (CIKM)*. 2011.

[29] *Breadth First Search*. URL: https://en.wikipedia.org/wiki/Breadth-first_search.

[30] Nathan Bronson et al. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *USENIX Technical Conference (ATC)*. 2013.

[31] *Building a follower model from scratch*. URL: https://bit.ly/2nDwvJU.

[32] Michael Burrows and David J Wheeler. "A block-sorting lossless data compression algorithm". In: (1994).

[33] Diego Calvanese et al. "Rewriting of Regular Expressions and Regular Path Queries". In: *ACM Symposium on Principles of Database Systems (PODS)*. 1999, pp. 194–204.

[34] Chee-Yong Chan, Minos Garofalakis, and Rajeev Rastogi. "RE-tree: An Efficient Index Structure for Regular Expressions". In: *Proceedings of the VLDB Endowment* (2003).

[35] Badrish Chandramouli et al. "FASTER: A Concurrent Key-Value Store with In-Place Updates". In: *ACM SIGMOD*. 2018.

[36] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006.

[37] Surajit Chaudhuri. "An Overview of Query Optimization in Relational Systems". In: *ACM Symposium on Principles of Database Systems (PODS)*. 1998.

[38] Flavio Chierichetti et al. "On Compressing Social Networks". In: *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 2009.

[39] Laura Chiticariu et al. "The SystemT IDE: An Integrated Development Environment for Information Extraction Rules". In: *ACM International Conference on Management of Data (SIGMOD)*. 2011.

[40] Junghoo Cho and Sridhar Rajagopalan. "A Fast Regular Expression Indexing Engine". In: *IEEE International Conference on Data Engineering (ICDE)*. 2001.

[41] Brian F Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *ACM Symposium on Cloud Computing (SoCC)*. 2010.

[42] James Corbett et al. "Spanner: Google's Globally-distributed Database". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.

[43] Thomas H Cormen. *Introduction to Algorithms*. 2009.

[44] *CouchDB*. URL: http://couchdb.apache.org.

[45] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "A Graphical Query Language Supporting Recursion". In: *ACM International Conference on Management of Data (SIGMOD)*. 1987, pp. 323–330.

[46] Carlo Curino et al. "Schism: a Workload-Driven Approach to Database Replication and Partitioning". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 48–57.

[47] Nilesh Dalvi, Ravi Kumar, and Mohamed Soliman. "Automatic Wrappers for Large Scale Web Extraction". In: *Proceedings of the VLDB Endowment* (2011).

[48] Daniel J. Abadi and Samuel R. Madden and Nabil Hachem. "Column-Stores vs. Row-Stores: How Different Are They Really?" In: *ACM International Conference on Management of Data (SIGMOD)*. 2008.

[49] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[50] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2007.

[51] *Deep Learning Meets Heterogeneous Computing*. URL: http://on-demand.gputechconf.com/gtc/2014/presentations/S4651-deep-learning-meets-heterogeneous-computing.pdf.

[52] *Delta Encoding*. URL: http://en.wikipedia.org/wiki/Delta_encoding.

[53] *Demining the "Join Bomb" with graph queries*. URL: https://bit.ly/2m2kuNA.

[54] *Distributive Property*. URL: https://en.wikipedia.org/wiki/Distributive_property.

[55] Aleksandar Dragojević et al. "FaRM: Fast Remote Memory". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[56] Ayush Dubey et al. "Weaver: A High-Performance, Transactional Graph Store Based on Refinable Timestamps". In: *CoRR* abs/1509.08443 (2015).

[57] *Elasticsearch*. URL: http://elasticsearch.org.

[58] Orri Erling et al. "The LDBC Social Network Benchmark: Interactive Workload". In: *ACM International Conference on Management of Data (SIGMOD)*. 2015, pp. 619–630.

[59] Robert Escriva, Bernard Wong, and Emin Gün Sirer. "HyperDex: A Distributed, Searchable Key-value Store". In: *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 2012.

[60] *Extended Regular Expressions*. URL: http://pubs.opengroup.org/onlinepubs/9699919799.

[61] *Facebook*. URL: https://facebook.com.

[62] Ronald Fagin et al. "Spanners: A Formal Framework for Information Extraction". In: *ACM Symposium on Principles of Database Systems (PODS)*. 2013.

[63] Bin Fan, David G. Andersen, and Michael Kaminsky. "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013.

[64] Wenfei Fan. "Graph Pattern Matching Revised for Social Network Analysis". In: *ACM International Conference on Database Theory (ICDT)*. 2012.

[65] Wenfei Fan et al. "Adding regular expressions to graph reachability and pattern queries". In: *IEEE International Conference on Data Engineering (ICDE)*. 2011.

[66] Wenfei Fan et al. "Query Preserving Graph Compression". In: *ACM International Conference on Management of Data (SIGMOD)*. 2012.

[67] Paolo Ferragina and Giovanni Manzini. "An Experimental Study of a Compressed Index". In: *Information Sciences* 135.1 (2001), pp. 13–28.

[68] Paolo Ferragina and Giovanni Manzini. "An Experimental Study of an Opportunistic Index". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2001.

[69] Paolo Ferragina and Giovanni Manzini. "Indexing Compressed Text". In: *Journal of the ACM (JACM)* 52.4 (2005), pp. 552–581.

[70] Paolo Ferragina and Giovanni Manzini. "Opportunistic Data Structures with Applications". In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 2000.

[71] Robert D Finn et al. "Pfam: The protein families database". In: *Nucleic Acids Research* (2013).

[72] *FlockDB*. URL: https://github.com/twitter/flockdb.

[73] *Function Shipping: Separating Logical and Physical Tiers*. URL: https://docs.oracle.com/cd/A87860_01/doc/appdev.817/a86030/adx16nt4.htm.

[74] Alexandre Gattiker, Elisabeth Gasteiger, and Amos Marc Bairoch. "ScanProsite: a reference implementation of a PROSITE scanning tool". In: *Applied Bioinformatics* (2002).

[75] Gattiker, Alexandre and Gasteiger, Elisabeth and Bairoch, Amos Marc. "ScanProsite: a reference implementation of a PROSITE scanning tool". In: *Applied Bioinformatics* (2002).

[76] Davide Gianfelice et al. "Modificatory Provisions Detection: A Hybrid NLP Approach". In: *ACM International Conference on Artificial Intelligence and Law (ICAIL)*. 2013.

[77] *gMark Queries for LDBC Social Network Benchmark*. URL: https://github.com/graphMark/gmark/tree/master/demo/social/social-translated.

[78] Robert R. Goldberg. "Finite state automata from regular expression trees". In: *The Computer Journal* (1993).

[79] Joseph E Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[80] *Google*. URL: https://google.com.

[81] Roberto Grossi, Ankur Gupta, and Jeffrey Vitter. "High-order Entropy-compressed Text Indexes". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2003.

[82] Roberto Grossi and Jeffrey Scott Vitter. "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching". In: *SIAM Journal on Computing* 35.2 (2005), pp. 378–407.

[83] Varun Gupta et al. "Analysis of Join-the-Shortest-Queue Routing for Web Server Farms". In: (2007).

[84] Cecilia Hernández and Gonzalo Navarro. "Compressed Representations for Web and Social Graphs". In: *Knowledge and Information Systems* 40.2 (2014).

[85] Cecilia Hernández and Gonzalo Navarro. "Compression of Web and Social Graphs supporting Neighbor and Community Queries". In: *ACM Workshop on Social Network mining and Analysis (SNAKDD)*. 2011.

[86] Wing-Kai Hon et al. "Practical aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences". In: *Workshop on Algorithm Engineering and Experiments and Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALC)*. 2004.

[87] *How Google Search works*. URL: https://google.com/search/howsearchworks/crawling-indexing.

[88] Cheng Huang et al. "Erasure Coding in Windows Azure Storage." In: *USENIX Annual Technical Conference (ATC)*. 2012.

[89] *Hyperdex Bug*. URL: https://bit.ly/2mSipnN.

[90] *Introducing FlockDB*. URL: https://blog.twitter.com/2010/introducing-flockdb.

[91] *Introducing Graph Search Beta*. URL: https://bit.ly/2ogy4O7.

[92] *IUPAC One letter codes for Amino Acids*. URL: http://bioinformatics.org/sms/iupac.html.

[93] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014. URL: http://snap.stanford.edu/data.

[94] Juha Kärkkäinen, Dominik Kempa, and Simon J Puglisi. "Parallel external memory suffix sorting". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2015, pp. 329–342.

[95] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. "BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2016.

[96] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. *Sprint: Regular Expression Queries on Compressed Data*. Tech. rep. 2019.

[97] Anurag Khandelwal et al. "ZipG: A Memory-efficient Graph Store for Interactive Queries". In: *SIGMOD*. 2017.

[98] Rajasekar Krishnamurthy et al. "SystemT: A System for Declarative Information Extraction". In: *ACM SIGMOD Record* (2009).

[99] Stefan Kurtz. "Reducing the Space Requirement of Suffix Trees". In: *Software: Practice and Experience* 29.13 (1999), pp. 1149–1171.

[100] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. "GraphChi: Large-Scale Graph Computation on Just a PC". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.

[101] Julian Labeit, Julian Shun, and Guy E Blelloch. "Parallel lightweight wavelet tree, suffix array and FM-index construction". In: *Journal of Discrete Algorithms* 43 (2017), pp. 2–17.

[102] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[103] Andrew Lamb et al. "The Vertica Analytic Database: C-store 7 Years Later". In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1790–1801.

[104] Kevin Lang. "Finding good nearly balanced cuts in power law graphs". In: *Preprint* (2004).

[105] Ben Langmead et al. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome Biology* 10.3 (2009), pp. 1–10.

[106] Jure Leskovec et al. "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters". In: *Internet Mathematics* (2009).

[107] *Levenshtein distance*. URL: https://en.wikipedia.org/wiki/Levenshtein_distance.

[108] Haoyuan Li et al. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *ACM Symposium on Cloud Computing (SoCC)*. 2014.

[109] Yunyao Li et al. "VINERy: A Visual IDE for Information Extraction". In: *Proceedings of the VLDB Endowment* ().

[110] Leonid Libkin and Domagoj Vrgoč. "Regular Path Queries on Graphs with Data". In: *ACM International Conference on Database Theory (ICDT)*. 2012, pp. 74–85.

[111] Hyeontaek Lim et al. "MICA: A Holistic Approach to Fast In-memory Key-value Storage". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[112] Hyeontaek Lim et al. "SILT: A Memory-Efficient, High-Performance Key-Value Store". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2011.

[113] *LinkBench*. URL: https://github.com/facebookarchive/linkbench.

[114] *Linkedin*. URL: https://linkedin.com.

[115] *Longhair: Fast Cauchy Reed-Solomon Erasure Codes*. URL: https://github.com/catid/longhair.

[116] Yucheng Low et al. "GraphLab: A New Framework For Parallel Machine Learning". In: *arXiv preprint arXiv:1408.2041* (2014).

[117] *LZ4*. URL: https://lz4.github.io/lz4.

[118] Antonio Maccioni and Daniel J. Abadi. "Scalable Pattern Matching over Compressed Graphs via Dedensification". In: *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 2016.

[119] Grzegorz Malewicz et al. "Pregel: A System for Large-scale Graph Processing". In: *ACM International Conference on Management of Data (SIGMOD)*. ACM. 2010.

[120] Udi Manber and Gene Myers. "Suffix Arrays: A New Method for On-line String Searches". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1990.

[121] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. "Cache Craftiness for Fast Multicore Key-value Storage". In: *ACM European Conference on Computer Systems (EuroSys)*. 2012.

[122] Norbert Martinez-Bazan, Sergio Gómez-Villamor, and Francesc Escalé-Claveras. "DEX: A high-performance graph database management system". In: *IEEE Data Engineering Workshops (ICDEW)*. 2011.

[123] Hossein Maserrat and Jian Pei. "Neighbor Query Friendly Compression of Social Networks". In: *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. Washington, DC, USA, 2010.

[124] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. 2010.

[125] *MemCached*. URL: http://memcached.org.

[126] Maged M Michael. "High Performance Dynamic Lock-free Hash Tables and List-based Sets". In: *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 2002.

[127] *Microsoft GraphView*. URL: https://github.com/facebookarchive/linkbench.

[128] *Microsoft SQL Server*. URL: https://www.microsoft.com/en-us/sql-server/sql-server-2016.

[129] Jeffrey C Mogul et al. "Potential benefits of delta encoding and data compression for HTTP". In: *ACM SIGCOMM Computer Communication Review*. 1997.

[130] *MongoDB*. URL: http://mongodb.org.

[131] Y Mori. *libdivsufsort: A lightweight suffix-sorting library*. 2010.

[132] Mulder, Michael and Nezlek, GS. "Creating Protein Sequence Patterns Using Efficient Regular Expressions in Bioinformatics Research". In: *IEEE International Conference on Information Technology Interfaces (ITI)*. 2006.

[133] Subramanian Muralidhar et al. "f4: Facebook's Warm BLOB Storage System". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[134] *MySQL*. URL: https://mysql.com.

[135] Gonzalo Navarro and Eliana Providel. "Fast, Small, Simple Rank/Select on Bitmaps". In: *Experimental Algorithms*. Vol. 7276. Lecture Notes in Computer Science. 2012, pp. 295–306.

[136] *Neo4j*. URL: http://neo4j.com.

[137] *Neo4j Pushes Graph DB Limits Past a Quadrillion Nodes*. URL: https://datanami.com/2016/04/26/neo4j-pushes-graph-db-limits-past-quadrillion-nodes.

[138] Yasuhiro Ogawa, Shintaro Inagaki, and Katsuhiko Toyama. "Automatic Consolidation of Japanese Statutes Based on Formalization of Amendment Sentences". In: *Conference on New Frontiers in Artificial Intelligence (JSAI)*. 2008.

[139] *openCypher*. URL: http://www.opencypher.org.

[140] *Oracle Database*. URL: https://oracle.com/index.html.

[141] *OrientDB*. URL: http://orientdb.com/.

[142] John Ousterhout et al. "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM". In: *ACM SIGOPS Operating Systems Review* 43.4 (2010), pp. 92–105.

[143] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. "Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems". In: *ACM International Conference on Management of Data (SIGMOD)*. 2012.

[144] *Pizza&Chili Corpus: Compressed Indexes and their Testbeds*. URL: http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array.

[145] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. 2010.

[146] Raphael Polig et al. "Compiling text analytics queries to FPGAs". In: *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. 2014.

[147] *PostgreSQL*. URL: https://postgresql.org.

[148] *Presto*. URL: http://prestodb.io.

[149] *Property Graph Model*. URL: https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model.

[150] Puntervoll, Pål and Linding, Rune and Gemünd, Christine and Chabanis-Davidson, Sophie and Mattingsdal, Morten and Cameron, Scott and Martin, David MA and Ausiello, Gabriele and Brannetti, Barbara and Costantini, Anna and others. "ELM server: A new resource for investigating short functional sites in modular eukaryotic proteins". In: *Nucleic Acids Research* (2003).

[151] K. V. Rashmi et al. "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster". In: *USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*. 2013.

[152] KV Rashmi et al. "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers". In: *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 2014.

[153] *Recommender System with Mahout and Elasticsearch*. URL: https://bit.ly/2ojbRyV.

[154] *Redis*. URL: http://redis.io.

[155] *Regular Expressions in MySQL*. URL: https://dev.mysql.com/doc/refman/5.7/en/regexp.html.

[156] *Regular Expressions in Oracle*. URL: https://bit.ly/1PUA96R.

[157] Paul Resnick and Hal R Varian. "Recommender systems". In: *Communications of the ACM* 40.3 (1997), pp. 56–59.

[158] Francesco Ricci, Lior Rokach, and Bracha Shapira. "Introduction to recommender systems handbook". In: *Recommender systems handbook*. Springer, 2011, pp. 1–35.

[159] Daniel Robenek, Jan Platos, and Vaclav Snasel. "Efficient In-memory Data Structures for n-grams Indexing." In: *DATESO*. 2013.

[160] Rodrigo González and Szymon Grabowski and Veli Mäkinen and Gonzalo Navarro. "Practical implementation of rank and select queries". In: *Workshop on Efficient and Experimental Algorithms (WEA)*. 2005.

[161] Kunihiko Sadakane. "Compressed Suffix Trees with Full Functionality". In: *Theory of Computing Systems* 41.4 (2007), pp. 589–607.

[162] Kunihiko Sadakane. "Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array". In: *International Conference on Algorithms and Computation (ISAAC)*. 2000.

[163] Kunihiko Sadakane. "New Text Indexing Functionalities of the Compressed Suffix Arrays". In: *Journal of Algorithms* 48.2 (2003), pp. 294–313.

[164] Kunihiko Sadakane. "Succinct Representations of Lcp Information and Improvements in the Compressed Suffix Arrays". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2002.

[165] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. 1989.

[166] *SAP HANA*. URL: http://saphana.com.

[167] Maheswaran Sathiamoorthy et al. "XORing Elephants: Novel Erasure Codes for Big Data". In: *International Conference on Very Large Data Bases (VLDB)*. 2013.

[168] *Scalable Recommender Systems: Where Machine Learning Meets Search!* URL: https://goo.gl/g6eFf7.

[169] *SDSL*. URL: https://github.com/simongog/sdsl-lite.

[170] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. "Automated Discovery of Valid Test Strings from the Web Using Dynamic Regular Expressions Collation and Natural Language Processing". In: *IEEE International Conference on Quality Software (QSIC)*. 2012.

[171] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *ACM International Conference on Management of Data (SIGMOD)*. ACM. 2013.

[172] Julian Shun and Guy E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoP)*. 2013.

[173] Julian Shun, Laxman Dhulipala, and Guy Blelloch. "Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+". In: *IEEE Data Compression Conference (DCC)*. 2015.

[174] Christian JA Sigrist et al. "New and continuing developments at PROSITE". In: *Nucleic Acids Research* (2012).

[175] Swaminathan Sivasubramanian. "Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service". In: *ACM International Conference on Management of Data (SIGMOD)*. 2012.

[176] *snappy*. URL: http://google.github.io/snappy.

[177] Yee Jiun Song et al. "RPC Chains: Efficient Client-server Communication in Geodistributed Systems". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2009.

[178] *Sparksee by Sparsity Technologies*. URL: http://sparsity-technologies.com.

[179] PierLuigi Spinosa et al. "NLP-based Metadata Extraction for Legal Text Consolidation". In: *ACM International Conference on Artificial Intelligence and Law (ICAIL)*. 2009.

[180] *Sprint*. URL: https://github.com/amplab/sprint.

[181] Michael Stonebraker et al. "C-Store: A Column-Oriented DBMS". In: *International Conference on Very Large Data Bases (VLDB)*. 2005.

[182] *Succinct*. URL: https://github.com/amplab/succinct-cpp.

[183] *Succinct on Apache Spark*. URL: https://github.com/amplab/succinct.

[184] *Suffix Array*. URL: http://en.wikipedia.org/wiki/Suffix_array.

[185] *Suffix Tree*. URL: http://en.wikipedia.org/wiki/Suffix_tree.

[186] James W Thatcher. "Tree Automata: An Informal Survey". In: (1973).

[187] *The Infrastructure Behind Twitter: Scale*. URL: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html.

[188] *TimescaleDB: SQL made scalable for time-series data*. URL: https://timescale.com/papers/timescaledb.pdf.

[189] *Titan*. URL: http://thinkaurelius.github.io/titan.

[190] *Titan Data Model*. URL: http://s3.thinkaurelius.com/docs/titan/current/data-model.html.

[191] *TPC-H*. URL: http://tpc.org/tpch/.

[192] Dominic Tsang and Sanjay Chawla. "A Robust Index for Regular Expression Queries". In: *ACM Conference on Information and Knowledge Management (CIKM)*. 2011.

[193] *Twitter*. URL: https://twitter.com.

[194] Esko Ukkonen. "On-Line Construction of Suffix Trees". In: *Algorithmica* 14 (1995), pp. 249–260.

[195] *Vertica Does Not Compute on Compressed Data*. URL: http://tinyurl.com/l36w8xs.

[196] Sebastiano Vigna. "Broadword Implementation of Rank/Select Queries". In: *Workshop on Efficient and Experimental Algorithms (WEA)*. 2008.

[197] *Virtuoso Universal Server*. URL: http://virtuoso.openlinksw.com.

[198] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. "Towards Elastic Transactional Cloud Storage with Range Query Support". In: *Proc. VLDB Endow.* 3.1-2 (2010), pp. 506–514.

[199] Christopher B Walton, Alfred G Dale, and Roy M Jenevein. "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins". In: *International Conference on Very Large Data Bases (VLDB)*. 1991.

[200] Rui Wang, C Conrad, and S Shah. "Using Set Cover to Optimize a Large-Scale Low Latency Distributed Graph". In: *Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2013.

[201] Peter Weiner. *Linear Pattern Matching Algorithms*. 1973.

[202] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. "Autoscaling Tiered Cloud Storage in Anna". In: *Proc. VLDB Endow.* 12.6 (2019), pp. 624–638.

[203] *Yahoo! Search*. URL: https://yahoo.com.

[204] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2010.

[205] Dong Zhou, David Andersen, and Michael Kaminsky. "Space Efficient, High Performance Rank and Select Structures on Uncompressed Bit Sequences". In: *Experimental Algorithms*. Vol. 7933. Lecture Notes in Computer Science. 2013, pp. 151–163.